WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2011

Distributed Monte Carlo Simulation

Aruna Sri Bommagani
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
Bommagani, Aruna Sri, "Distributed Monte Carlo Simulation" (2011). *Graduate Theses, Dissertations, and Problem Reports*. 675.
https://researchrepository.wvu.edu/etd/675

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Distributed Monte Carlo Simulation

Aruna Sri Bommagani

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Matthew C. Valenti, Ph.D., Chair
Vinod K. Kulathumani, Ph.D.
Yenumula V. Reddy, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2011

Keywords: Monte Carlo simulation, distributed computing, scheduling

Copyright 2011 Aruna Sri Bommagani

Abstract

Distributed Monte Carlo Simulation

Aruna Sri Bommagani

Monte Carlo simulation is an effective way to analyze models of sophisticated problems, but often suffers from high computational complexity. Distributed computing is an effective technology that can be used for compute-intensive applications, such as Monte Carlo simulation. The goal of this thesis is to combine the concepts of Monte Carlo simulation and distributed computing in an effort to develop an efficient system capable of rapidly executing computationally-demanding simulations.

When distributed computing is used to support the simulations of multiple users, a scheduling algorithm is required to allocate resources among the users' jobs. In this thesis, a scheduling algorithm is developed that is suitable for Monte Carlo simulation and utilizes the available distributed-computing resources. The unified framework for scheduling is capable of accommodating classic scheduling algorithms such as equal job share, first-in first-out (FIFO), and proportional fair scheduling. The behavior of the scheduler can be controlled by just three parameters. By choosing appropriate parameter values, individual users and their jobs can be assigned different priorities. By introducing an appropriate analytical model, the role of these parameters on system behavior is thoroughly investigated. Using insights obtained by studying the analytical model, a complete distributed Monte Carlo system is designed and presented as a case study.

Acknowledgements

Immense thanks to my advisor and committee chair, Dr. Matthew C. Valenti. This thesis is possible because of the guidance and support provided by Dr. Valenti, through numerous technical discussions with the freedom to deliver thoughts, attention to detail, and constructive feedback since its inception to the end. I am also thankful to Dr. Valenti for being cordial all the time and for guiding me through my Master's study apart from the research work.

I thank Dr. Vinod K. Kulathumani and Dr. Yenumula V. Reddy for being on my committee.

I thank WVU writing center for the help provided in writing my thesis report. I am thankful to my lab mate, Terry R. Ferrett for all the technical support and for the guidance in starting my thesis documentation. I thank all my lab mates in Wireless Communication Research Laboratory.

On a personal note, I thank all my dear friends for their love and support. I am blessed to have a wonderful family. I humbly thank my brother-in-law, Dr. Srinivas for being an inspiration to me and for the love and support. My wonderful niece, Harika and nephew, Nayan for all the smiles with their love and innocence. Words cannot express my gratitude to my family. Whatever I am, and all my achievements are possible because of the sacrifices, love, and patience of my mother, father, sister, and brother, who always stand by me. Their faith in me, guidance, and encouragement imparts in me the necessary confidence in all my endeavors. To them, I dedicate this thesis.

Contents

List of Figures

List of Tables

Notation

We use the following notation and symbols throughout this thesis.

X	:	A random variable
$f_X(x)$	:	pdf of X
$E\left[X\right], \hat{M_X}$	:	Mean of X
x_i	:	i^{th} Monte Carlo realization of X
N	:	Total number of trials used to estimate the mean of X
N_i	:	Number of trials used to estimate $E[X]$
k	:	Number of tasks required to complete a job
$\hat{k}$, $\hat{k_j}$	:	Expected number of tasks required to complete job j
$x_{n,j}$	:	n^{th} Monte Carlo realization of j^{th} task
p	:	Success probability of a trial
S	:	Total number of successes in N trials of X
$\mathcal{U}$	:	A set of users in the system
U_i	:	i^{th} user in the set $\mathcal{U}$
$\mathcal{J}$	:	A set of jobs in the queue
$J_{i,j,\ell}$	:	j^{th} job in the queue, that belongs to i^{th} user in time slot, ℓ
C	:	Number of computing resources available in a time slot
t_1	:	Time period to check arrival of new jobs by the job manager
t_2	:	Time period to execute a task
Γ_p	:	Total processor time required to complete a job
Γ_c	:	Total completion time of a job (includes time from submission to completion)
δ	:	Number of time slots required for job execution
$E_{i,\ell}$	:	Exponentially-weighted cumulative processing share of user, i, in time slot, ℓ
α	:	Forgetting factor, used in evaluating $E_{i,\ell}$
β	:	Weight factor that moderates $E_{i,\ell}$
$C_{i,j,\ell}$	:	Number of completed tasks of job, j, in time slot, ℓ, of user, i
$R_{i,j,\ell}$	:	Remaining number of tasks of job, j, in time slot, ℓ, of user, i
c_1	:	System defined value of remaining percent of job
$r_{i,j,\ell}$	:	Remaining percent of job, j, in time slot, ℓ, that belongs to user, i
$\hat{P_{i,j,\ell}}$	:	Expected priority of job, j, in time slot, ℓ, that belongs to user, i
$P_{i,j,\ell}$	:	Priority of job, j, in time slot, ℓ, that belongs to user, i

c_2	:	System defined value of percent of workers assigned to a new job
$\tau_{i,\ell}$	:	Number of tasks serviced for user, i, in time slot, ℓ
$W_{i,j,\ell}$	:	Weight of job, j, in time slot, ℓ, that belongs to user, i
T	:	Actual duration (number of time slots) of the simulation
ρ	:	Job submission probability of a user
$F_{i,j,\ell}$	:	Job factor of job, j, in time slot, ℓ, of user, i
γ	:	Weight factor that moderates $F_{i,j,\ell}$.

Chapter 1

Introduction

1.1 Monte Carlo Simulation

A Monte Carlo simulation is an experimental method for estimating the expected value of a random variable X. Random numbers are used to perform the simulation. Let X be a random variable with *probability density function* (pdf) $f_X(x)$. The goal is to estimate the mean $E[X]$. The mean of X is

$$E[X] = \int x f_x(x)\, dx \tag{1.1}$$

Finding the expected value using (1.1) is often not feasible because the pdf may not be available or the integral may not be easy to calculate. An alternative technique to find the expected value is to use the Monte Carlo integration technique [1]. Let x_i be the i^{th} Monte Carlo realization of X and N be the number of trials. An estimate of the expected value of X may be found using

$$\hat{M_X} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1.2}$$

A large number of trials are used to perform a simulation and the time required to compute the mean can be largely reduced by performing trials in parallel using distributed computing. Parallel simulation is discussed in section 3.1 of chapter 3.

Often the random variable X is Bernoulli. A Bernoulli random variable can take a value of either 1 or 0. Let p be the probability of a 1 and $1 - p$ be the probability of a 0. To generate a Bernoulli variable X, we generate a uniform variable Y, uniform over 0 to 1, and set $X = 1$, if $Y < p$. The event in which $X = 1$, is a *success* while the event that $X = 0$, is a *failure*. Bernoulli trials are important in the analysis of communication systems because they can be used to indicate a bit or packet error. It follows that if X represents a bit error, then $\mathrm{E}[X]$ is the bit error rate (BER).

1.2 Random Number Generation

A random number sequence is a sequence of numbers with no specific pattern among them. There is a significant body of literature on *random numbers*, *random number generators*, and the statistical tests that guarantee the standard of the random number generators [2]. However, before proceeding, we need to ask the question *Why do we need random numbers?* We cover the need for random numbers and some of their properties in the next section, followed by the two main approaches for generating random numbers using a computer, the methods for generating *pseudo random number generators* (PRNGs) and a brief section on the *Mersenne twister* random number generator.

1.2.1 Why Random Numbers?

Random numbers are needed in a variety of fields. Random numbers are required in scientific problems for the simulation of the stochastic processes, simulation in physical, biological, chemical, astrophysical problems, cryptography, VLSI testing, computer games and computational statistics. Random numbers are required for any Monte Carlo simulation, and are therefore important for the many diverse fields that use Monte Carlo simulation. To mention a few applications of Monte Carlo simulation, in digital communications it is useful to obtain bit error rates (BER), in finance it is useful for risk analysis, in physical chemistry it is useful for the simulations involving atomic clusters, and in geophysics to invert seismic data.

1.2.2 Random Number Generators

A *random number generator (RNG)* [2, 3, 1] is a device that generates a random sequence of numbers. The device used to generate a random number sequence can be either a computational (used for generating pseudo-random numbers) or a physical device (for generating true-random numbers).

The two main approaches for generating random numbers [1] are *pseudo random number generators* (PRNGs) and *true random number generators* (TRNGs). Random number generators should be able to produce genuine random numbers in a small amount of time, must have a large least period length (the length at which a sequence repeats), should be reproducible (possible for PRNGs) for testing a model, and preferably, amenable to parallel generation.

Pseudo random number generators are not truly random but have genuine enough randomness. They are generated by deterministic algorithms based on mathematical formulae using a computer. The sequence of numbers generated using PRNGs are *deterministic* as they can be reproduced by providing the initial state of the sequence, known as the *seed*. PRNGs are *periodic* and *efficient* as they can produce many numbers in a little time. Efficiency is an important feature to test the stability of the model. PRNGs are particularly useful where a large number of random numbers are required (e.g., in simulation and modeling applications). They are not very useful where the sequence of random numbers should be unpredictable (e.g., cryptography).

True random number generators use the randomness of the physical phenomenon (e.g., decay of a radioactive source, atmospheric noise) to generate random numbers. Unlike PRNGs, TRNGs are less efficient as they generally take long time to produce numbers (because of the source used to generate random numbers). Numbers generated using TRNGs are non-deterministic and aperiodic, though there might be a possibility that the same sequence of random numbers are generated by chance. TRNGs are best suited for cryptography and gambling machines.

[1] `http://www.random.org/randomness/`

1.2.3 Methods of Pseudo Random Number Generation

Pseudo random number generation involves generating a sequence of real numbers using a function of independent and identically distributed (iid) variables on $\mathbb{R}$. The two categories of random numbers are *uniform random numbers* and *non-uniform random numbers.* Uniform random numbers are generated using a uniform distribution, and *non-uniform random numbers* are generated using distributions (e.g.,Gaussian, gamma, poisson, binomial and exponential distributions) other than uniform distribution. Transformation methods are used to transform uniform random numbers to generate non-uniform random numbers (section 4.9 of [4]). Let X be a uniform random variable in the interval $[0, 1]$. To generate a random variable, Y, with the desired pdf, generate X and find the inverse of its cumulative distribution function (CDF), $Y = F_X^{-1}(x)$.

1.2.4 Mersenne Twister

The *Mersenne twister* (MT) is the default random number generator in Matlab. It was developed by Makoto Matsumoto and Takuji Nishimura during 1996-1997. MT [5] is a uniform PRNG with a long period, a *Mersenne prime* period of $2^{19937}-1$ and 623-dimensional equi-distribution with 32-bit accuracy. MT is a variant of *twisted generalized feedback shift register (TGFSR)* [6] with improvements to realize the Mersenne prime period. The TGFSR and the *generalized feedback shift register* (GFSR) can be obtained by using a particular set of parameter values of the MT recurrence function. The MT algorithm seem to be most suitable for the Monte Carlo simulations. Using MT, higher bit (64-bit) integers can be generated by concatenating the words. However, the algorithm is not suitable for use in cryptography and has the limitation of *zero-excess initial state* [7].

Variants of Mersenne Twister

SIMD-oriented fast Mersenne twister (SFMT)[2] is a *linear feedbacked shift register* (LFSR) [6] generator of a 128-bit pseudo-random integer using parallelism features like multi-stage pipelining and SIMD instructions. Compared to MT, SFMT is much faster and has a faster

[2]`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html`

recovery from zero-excess initial state. *CryptMT*[3] is suitable for use in cryptography. It's a stream cipher with a combination of LFSR like MT and non-linear filter based on multiplication. CryptMT has a period of a non-zero multiple of $2^{19937} - 1$, and the 1241-dimensional equi-distribution of 8-bit output sequence.

1.3 Confidence Intervals

A *confidence interval* provides an interval estimate of a statistical parameter (e.g., mean and variance). The values in the interval have a high reliability to contain the true value of the parameter. Observed data is used to find the interval estimate, and the estimate may differ based on the data in consideration. A narrow interval implies the estimate of the parameter is more accurate. *Confidence level* or *confidence coefficient* is the degree of consistency with which the parameter value lies within the interval.

Mathematically, for a random variable X the probability of observing a true value of a parameter for instance, the mean, μ, within a confidence interval ($u(X)$, the upper bound, and $l(X)$, the lower bound) of $[l(X), u(X)]$ and confidence level (1 - α) is given by

$$P[l(X) \leq \mu \leq u(X)] = (1 - \alpha) \tag{1.3}$$

The Monte Carlo simulation in the system, designed considering the analytical model in chapter 3, an assumption is made such that a certain number of trials, N, (divided into multiple tasks) result in the required successes, S. The simulation is stopped when N trials are completed. In this case, S is random and N is the binomial random variable. A more interesting and challenging case is when the simulation is performed until the number of observed successes is S. This can be obtained by not fixing N, in the simulation i.e., N is random and S is fixed. In the following sub-sections we go through the Gaussian approximation of the binomial distribution to find the confidence interval of the mean of a random variable and the confidence interval for the variables that can be approximated to Gaussian random variables.

[3]`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/CRYPTMT/index.html`

1.3.1 Confidence Interval: Gaussian Approximation

Let X be a random variable to estimate mean, μ, and variance, σ^2.

Confidence interval for a normal random variable (Gaussian approximation of binomial distribution) can be defined using the *Wilson score function* [8]. The Wilson score function is based on the set:

$$\left\{ p : \left| \frac{\hat{p} - p}{\sqrt{p(1-p)/N}} \right| \leq d_\alpha \right\} \tag{1.4}$$

where $\hat{p}$ is the *maximum likelihood* estimate of probability of a success and d_α is the solution to

$$1 - \alpha = \int_{-d_\alpha}^{+d_\alpha} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt \tag{1.5}$$

The confidence interval can be obtained by squaring both sides of (1.4) and solving the quadratic equation for p

$$\left[\frac{\hat{p} + \frac{d_\alpha^2}{2N} - d_\alpha \sqrt{\frac{\hat{p}(1-\hat{p})}{N} + \left(\frac{d_\alpha}{2N}\right)^2}}{1 + \frac{d_\alpha^2}{N}}, \frac{\hat{p} + \frac{d_\alpha^2}{2N} + d_\alpha \sqrt{\frac{\hat{p}(1-\hat{p})}{N} + \left(\frac{d_\alpha}{2N}\right)^2}}{1 + \frac{d_\alpha^2}{N}} \right] \tag{1.6}$$

The other way to find the confidence interval is by using the expected mean and the sample variance estimate [4]. The expected sample mean of X, a Gaussian random variable is given by

$$E[X] = \overline{X}_N = \frac{1}{N} \sum_{j=1}^{N} X_j \tag{1.7}$$

and the sample variance estimate

$$\begin{aligned} \hat{\sigma}^2 &= E\left[(X - \mu)^2\right] \\ &= E\left[X^2\right] - \left(E[X]\right)^2 \end{aligned} \tag{1.8}$$

Using the equivalence that will be described below, the upper and lower bounds can be determined by considering a normally distributed sample mean, $\overline{X}_N$, from a normally distributed sample with mean, μ, and standard error, $\frac{\sigma}{\sqrt{N}}$. By standardizing we get, $\frac{\overline{X}_N-\mu}{\sigma/\sqrt{N}}$ and we can find the upper bound $-a$ and lower bound a between which, $\frac{\overline{X}_N-\mu}{\sigma/\sqrt{N}}$ lies with probability, $(1-\alpha)$

$$\begin{aligned}\left\{-a \leq \frac{\overline{X}_N-\mu}{\sigma/\sqrt{N}} \leq a\right\} &= \left\{\frac{-a\sigma}{\sqrt{N}} \leq \overline{X}_N-\mu \leq \frac{a\sigma}{\sqrt{N}}\right\} \\ &= \left\{-\overline{X}_N-\frac{a\sigma}{\sqrt{N}} \leq -\mu \leq -\overline{X}_N+\frac{a\sigma}{\sqrt{N}}\right\} \\ &= \left\{\overline{X}_N-\frac{a\sigma}{\sqrt{N}} \leq \mu \leq \overline{X}_N+\frac{a\sigma}{\sqrt{N}}\right\}\end{aligned}$$

Confidence Interval for mean: For Gaussian random variables with unknown mean and known variance

For a set of Gaussian random variables with an unknown mean, μ, and a known variance, σ^2, the confidence interval of mean, μ, is found as follows:

$$\begin{aligned}1-2Q(z) &= P\left[-z \leq \frac{\overline{X}_N-\mu}{\sigma/\sqrt{N}} \leq z\right] \\ &= P\left[\overline{X}_N-\frac{z\sigma}{\sqrt{N}} \leq \mu \leq \overline{X}_N+\frac{z\sigma}{\sqrt{N}}\right] \qquad (1.9)\end{aligned}$$

For $\alpha = 2Q(z_{\alpha/2})$ where $z_{\alpha/2}$ is the critical value, the $(1-\alpha)\times 100\%$ confidence interval for the parameter μ using (1.9) is given by

$$\left[\overline{X}_N-\frac{z_{\alpha/2}\sigma}{\sqrt{N}}, \overline{X}_N+\frac{z_{\alpha/2}\sigma}{\sqrt{N}}\right] \qquad (1.10)$$

Confidence Interval for mean: For Gaussian random variables with unknown mean and variance

For a set of Gaussian random variables with an unknown mean, μ, and variance, σ^2, the confidence interval of mean, μ, can be found by replacing the variance, σ^2, in the equation (1.10) with its estimate sample variance, $\hat{\sigma}^2$.

$$\left[\overline{X}_N-\frac{t\hat{\sigma}}{\sqrt{N}}, \overline{X}_N+\frac{t\hat{\sigma}}{\sqrt{N}}\right] \qquad (1.11)$$

The random variable here,

$$T = \frac{\overline{X}_N-\mu}{\hat{\sigma}_N/\sqrt{N}} \qquad (1.12)$$

has Student's t-distribution with $N-1$ degrees of freedom.

For a critical value, $t_{\alpha/2,N-1}$, the $(1-\alpha)\times 100\%$ confidence interval for mean, μ, is given by

$$\left[\overline{X}_N - \frac{t_{\alpha/2,N-1}\hat{\sigma}}{\sqrt{N}}, \overline{X}_N + \frac{t_{\alpha/2,N-1}\hat{\sigma}}{\sqrt{N}}\right] \tag{1.13}$$

For larger samples, the Student's t pdf is observed to approach the pdf of standard Guassian random variable N(0, 1).

For non-Gaussian random variables, X with unknown mean and known variance the equation (1.13) can be modified using the *method of batch means* (section 8.4 of [4]).

Confidence Interval for variance: For Gaussian random variables with unknown mean and variance

The estimate of sample variance is given by the equation (1.8) The below equation is used to develop confidence intervals for the variance of a Guassian random variable. It has a chi-square distribution with $N-1$ degrees of freedom.

$$\chi^2 = \frac{(N-1)\hat{\sigma}^2}{\sigma^2} = \frac{1}{\sigma^2}\sum\sum_{j=1}^{N}(X_j - \overline{X}_N)^2 \tag{1.14}$$

For a critical value $\chi^2_{\alpha/2,N-1}$, the $(1-\alpha)\times 100\%$ confidence interval for variance, σ^2, is given by

$$\left[\frac{(N-1)\hat{\sigma}^2}{\chi^2_{\alpha/2,N-1}}, \frac{(N-1)\hat{\sigma}^2}{\chi^2_{1-\alpha/2,N-1}}\right] \tag{1.15}$$

1.4 Thesis Outline

The objective of the thesis is to combine the concepts of Monte Carlo simulation and distributed computing to efficiently execute compute-intensive problems. Comparing various choices of distributed computing, we can conclude that cluster computing is an appropriate form of computing for academic research. To realize the thesis objective, a scheduling algorithm is proposed and an analytical model is created to investigate the effect of the parameters in the algorithm on the system. Using insights obtained by studying the analytical model, a complete distributed Monte Carlo system is designed and presented as a case study.

The purpose of chapter 2 is to understand distributed computing and its various types. The types of distributed computing covered in the chapter are cluster computing, grid computing, volunteer computing, cloud computing, and utility computing. The different types of computing are presented in a hierarchical order of resources usage, where a cluster is formed by combining together a set of computers; a collection of distributed computers, clusters can be a grid; computing power of a collection of computers donated by anonymous users is volunteer computing; and cloud and utility computing require grid. For each type of computing, we cover its characteristics, issues and challenges involved, architecture, and the difference between the types of computing. Single system image (SSI) concept, where a cluster can be represented as a single resource is presented in cluster computing. Middleware management systems provide services to manage any specific type of computing. We provide a brief description of available middleware management systems for each type of computing. The chapter also includes a section on Google App Engine, a cloud computing technology developed by Google which provides platform as a service to develop and deploy web applications on Google's infrastructure.

Chapter 3 introduces *parallel Monte Carlo simulation*, a mechanism to quickly execute jobs by dividing a job into multiple tasks, using distributed computing. An analytical model is presented, which is characterized by a few critical parameters. Next, a scheduling algorithm is presented, and the behavior of the scheduler can be controlled by just three parameters of the algorithm. The scheduling algorithm includes the forgetting factor, α, and the weight factor, β, which moderates the cumulative processing share of a user. Following this, the influence of the parameters on the system behavior is presented. The next part of the chapter introduces a refined algorithm that adds a third parameter, γ, which moderates the priority assigned to job through the job factor parameter. The refined algorithm is followed by a discussion on the effect of α, β, and γ parameters on the system. Towards the end of the chapter, we present two classic algorithms, *equal job share* and *FIFO*, which can be realized by varying the parameter values in the scheduling algorithm. Finally, we conclude the chapter by presenting the recommended parameter values, obtained from the data resulted in investigating the system behavior.

Chapter 4 presents the design of the system which is developed using the insights from the analytical model. It includes the implementation details, and handling of the files and data. A brief discussion is presented on how the actual system implementation differs from the analytical model. Matlab programming is used to coordinate workers, schedule, and execute jobs on the cluster. A user-friendly *graphical user interface* (GUI) web application is presented which is developed for the easy access to the cluster. Using the web application users can sign in to their account from virtually anywhere to manage their jobs. Administrators have extra accessibility features to manage all the users and jobs. The chapter also includes few screen shots of the web application. Chapter 5 presents conclusions and makes suggestions for future work.

Chapter 2

Distributed Computing

Distributed computing is a concept involving a systems of multiple networked computers each with a separate local memory, which communicate and coordinate to solve compute-intensive and data-intensive problems. Distributed computing offers several advantages when compared to a single high-performance computer (e.g., a super computer). Distributed computing is cost-effective as it is easy to have several low-end computers networked together to solve a large problem. It is more reliable as a failure in a single computer in the distributed system negligibly impacts the performance of the overall system. It is easy to expand the system by adding more computers and also easy to eliminate unused or outdated computers from the system. Distributed computing is effective in solving large scientific problems. It is globally popularized by allowing individuals to participate by donating their computing power through *volunteer computing.*

In this chapter, we will cover different types of distributed computing. In section 2.1 we cover *cluster computing*, different categories of cluster computing, such as high-availability clusters, load-balancing clusters, and high-performance clusters, and distinct characteristics of cluster computing that provide advantages over other alternative options for solving large problems. We also cover the concept and the advantage of representing a cluster as a *single system image* (SSI), implementation of SSI at various levels of cluster architecture, and some of the existing resource management software. In section 2.2 we cover *grid computing*, its characteristics, architecture and grid middleware, which provides software libraries for management and accessibility of the resources. We list some of the issues and challenges in grid computing and close the section with a note on the comparison between cluster computing and grid computing. In section 2.3 we cover *volunteer computing* and review the two major middleware management softwares *BOINC*, an open source software for computing using volunteered resources and *Frontier*, a software solution for grid-computing. Later in the section we review some of the major issues and challenges such as security, reliability and close the section with a comparison between grid computing and volunteer computing. In section 2.4 we cover *cloud computing*, characteristics of cloud computing, the hierarchical architecture that include IaaS, PaaS, and SaaS defined in the section, issues and challenges and a comparison between grid computing and cloud computing. A brief section on the working and the components of Google App Engine, a technology developed by Google is covered at the end of the cloud computing section. In section 2.5 we cover *utility computing*, its characteristics, which might be both an advantage or a disadvantage depending on the context (such as provider level or client level), and major concerns and improvements that may popularize the usage of utility computing are discussed. We close the section with a comparison between cloud computing and utility computing.

2.1 Cluster Computing

Cluster computing is a form of distributed computing of networked computers located in a close proximity and operating as an integrated computing resource. A cluster of computers are usually at a single physical location. Typically a cluster is homogeneous, i.e., all the cluster computers have the same hardware and operating system. All the computers in a cluster work co-operatively managed by a centralized resource manager. The key components of a cluster include: multiple standalone computers (such as PCs, workstations, or symmetric multiple processors (SMPs)), operating systems, high performance interconnects, middleware, parallel programming environments, and applications.

2.1.1 Characteristics of Cluster Computing

The characteristics that make cluster computing suitable for certain tasks are high availability, high scalability, low network latency, reduced cost of maintenance, and easy manageability. A cluster provides *high availability* by eliminating any points of failure and is *highly scalable* as nodes in a cluster can be added or removed without affecting the service provided by the cluster. As all the nodes in a cluster are located at one physical location, they are connected with high bandwidth connections leading to *low network latency.* The nodes in a cluster are represented as a single system image providing *easy manageability.* A cluster is *cost-effective* compared to super computers used in solving high-end problems. A cluster is a combination of multiple computers, and it is easy to get multiple computers with less configuration and combine them together to perform large problems. It is also easy to maintain a cluster compared to a single high end configuration computer.

2.1.2 Cluster Categories

Cluster computing can be categorized based on the availability of clusters, the type of load balancing, and the performance. The three major cluster categories are high-availability clusters, load-balancing clusters and high-performance clusters. *High availability clusters* provide high access to the services by maintaining redundant nodes in a cluster. This is useful in providing a reliable service in case any nodes fail. *Load balancing clusters* improve performance by sharing the workload among all the nodes in a cluster. *High performance clusters* exploit parallel processing power of the nodes in a cluster in order to provide high performance of the tasks running on the cluster. A cluster with a dedicated network and homogeneous nodes is known as a 'Beowulf cluster'[1].

2.1.3 Single System Image

A *single system image* [9] represents a heterogeneous and distributed system as a single unified computing resource and is useful to effectively manage a group of computers. SSI ensures high system availability and load balancing. The key SSI features are single entry point, user-interface, process and I/O space, process migration and check pointing, file system, job management system, virtual networking, and control point and management. SSI can be implemented as an additional layer at the hierarchical levels of hardware, operating

[1]`http://www.beowulf.org/`

system, middleware and application levels of cluster architecture [10].

SSI in the *hardware level* allows the user to view a cluster as a shared-memory system (e.g., systems such as *digital/compaq memory channel* and hardware *distributed shared memory* (DSM)). Hardware level SSI has the highest level of transparency but because of its rigid architecture it does not offer the flexibility required for the extension and enhancement of the system.

SSI at the *operating system level* supports process migration to provide dynamic load balancing, device handling, process/thread co-ordination and fast inter-process communication for both the system and user-level applications. This level offers full SSI to all users (includes application developers and end users) but is expensive to develop, maintain and it is difficult to keep pace with technological innovations emerging into mass-market operating systems [11].

Major cluster management systems are implemented as an extension to the Linux operating system. *MOSIX* [12] is a proprietary software implemented as an OS virtualization layer on top of the Linux kernel providing a single system image to the users and applications. *OpenMosix* is an open source cluster management software, and its development is currently inactive. *Linux Process Migration Infrastructure* (LinuxPMI)[2] is an open source Linux kernel extension for multi-system-image clustering and it is a continuation of the halted openMosix project. *Kerrighed* [13] is an open source single-system image project implemented as an extension to the Linux operating system.

SSI implementation in the *middleware level* includes cluster file system, for a single storage system with each node of the cluster having the same view of the data. It includes job management and scheduling systems such as CODINE [14], and cluster-enabled Java virtual machine.

The *application level* SSI is the highest level and is accessible to the end users. End users can view multiple components of an application as a single application at this level (e.g., a GUI-based tool PARMON [15] and Linux virtual server [16] with a single virtual server view). The advantage of the application level is that it is not required to develop all the components of this level to be SSI aware, only the required components may be developed

[2]`http://linuxpmi.org/trac/`

to be SSI aware and the users can benefit from it.

2.1.4 Middleware Management Systems

Middleware is a software layer that provides services for applications in a heterogeneous environment. Middleware includes *resource management* and *workload management* software to administer the available cluster resources to the jobs in the system. Job scheduling software schedules jobs and administers the job scheduling policies using process and job prioritization. Below, are some of the cluster middleware management systems software.

Maui Cluster Scheduler[3] is an open source job scheduler maintained and supported by Cluster Resources, Inc. for cluster and supercomputers. This is a job scheduler that supports multiple scheduling policies and fair share capabilities.

TORQUE[4] is an open source resource manager, which controls batch jobs and distributed compute nodes. TORQUE can be integrated with *moab workload manager* to improve the performance and usage on a cluster. The key features of TORQUE include tolerance to failure conditions in a cluster, scalability, usability and an user-interface for easy management of the jobs on the cluster.

Oracle Grid Engine (previously known as *sun grid engine* (SGE)) [17] is a *distributed resource management (DRM)* system. SGE is mainly used in a high-performance computing (HPC) cluster and is responsible for scheduling and managing the execution of different types of user jobs. Some of the key features include topology-aware scheduling and thread binding, multi-clustering, job check pointing, resource reservation and fault tolerance. The scheduling policies of SGE include *first-come first-served* (FCFS) and an optional administrator set function of *equal share* scheduler, a fair-share scheduler that distributes resources equally among all users and groups.

Moab Cluster Suite[5] is a cluster workload management that simplifies and unifies the cluster management across multiple environments. Its development was based on the Open Source Maui job scheduling package. Its key capabilities include performance utilization,

[3]`http://www.clusterresources.com/products/maui-cluster-scheduler.php`
[4]`http://www.clusterresources.com/products/torque-resource-manager.php`
[5]`http://www.clusterresources.com/products/moab-cluster-suite.php`

gain of control over automated tasks, policies and reporting, fair share of resources, increase in user productivity by allowing users to submit jobs from anywhere using an interface, and easy management across clusters.

The *Portable Batch System (PBS)* is a workload management system for Linux clusters that performs job scheduling. The three major components of PBS include a *job server*, also called *pbs_server*, which provides the batch services such as receiving and creating a batch job, modifying the job, protecting the job against system crashes and running the job. *Job executor* is a daemon, *pbs_mom* that actually places the job into execution when a job is received from the job server. The daemon creates a new session and returns the job's output to the user. *Job Scheduler* is a daemon that controls the running jobs based on job scheduling policies. PBS is available as *OpenPBS*, an unsupported open source version for small clusters and *PBS Professional (PBS Pro)*, a commercial version. PBS Pro[6] allows preemption between different priority jobs. The default scheduler in both the versions of PBS is *shortest job first* (SJF) and includes starvation prevention mechanism and backfilling of jobs.

2.2 Grid Computing

Grid computing [18] is a form of distributed computing composed of a loosely coupled computers that are geographically distributed to solve large compute or data intensive problems. The resources in grid computing belong to and are shared within an *organization(s)* or *virtual organization(s).*

2.2.1 Characteristics of Grid computing

Distinctive characteristics of grid computing [19] are collaboration, heterogeneity, aggregation, scalability, and decentralized control. The resources in a grid are a *collaboration* of resources within an organization or by more than a single organization forming *virtual organizations*. As the resources come from multiple geographical locations they are *heterogeneous* i.e., have different configuration with varied hardware and software components and

[6]`http://www.pbsworks.com/Product.aspx?id=1`

are connected with low-bandwidth connections. The collaborated resources are *aggregated* and shared among the organizations increasing the utilization, providing high performance, better quality of service, and easier access to resources and data. Resources in the grid have *high scalability* as the resources can be added or removed without affecting the services provided by the grid because of the large number of resources. Grid is a *decentralized system.* The resources have different ownership control mechanisms as they come from various administrative domains or organizations and do not have a single system view. Each node in a grid is autonomous and behaves as an independent entity.

2.2.2 Grid Architecture

Typical grid architecture [18, 20] consists of four layers: fabric layer, core middleware layer, user-level middleware, and applications and portal layer. Each layer is built upon services offered by the lower layers. Grid *fabric layer* consists of computational resources (such as PCs, clusters, and supercomputers), networks, storage devices, and scientific instruments (such as telescopes, sensor networks). *Core middleware layer* abstracts the complexity and heterogeneity of the fabric layer using the services of remote process management, quality of service, storage access, allocation of resources, information registration, discovery, and security. *User-level middleware* provides abstractions and services for the application development environments, programming tools, resource brokers, and application task scheduling. In *applications and portal layer* grid applications and portals are developed using the *user-level middleware.*

2.2.3 Grid Middleware

Grid Middleware provides software libraries for the management of the grid and for accessing the resources on the grid. Two main grid middleware systems are the *globus toolkit* and the *gridbus middleware.* Globus toolkit provides core-grid services, and Gridbus middleware provides more of the user-level services to the grid. We shall see some of the details of the systems in the sections below.

Globus Toolkit

Globus toolkit[7] [21, 22] includes software services and libraries required to build distributed system services and applications. It includes tools for building new web services, security infrastructure, service implementations, and client APIs and command line programs that provide access to various services. Service implementations include resource management, data movement, monitoring and discovery service, and other services. The *grid security infrastructure* (GSI) provides the authentication and authorization of grid users using proxies, certificates, and secure communication using a single sign on. The globus resource management package *globus resource allocation manager (GRAM)* provides remote job execution, monitoring, and reporting of status using the job managers. Job managers are created depending on the local scheduler (e.g., PBS). The data management is handled by the *GirdFTP* an extension to FTP protocol. GridFTP uses various services such as *replica location service* (RLS), for the creation and deletion of replicas of a file identified by the *logical file name (LFN).* The *monitoring and discovery service* (MDS) component of the toolkit facilitates collecting and querying of the resource information. The three levels of MDS consists of the *information providers* (IPs) in the bottom level, which gather and format resource data; *grid resource information service* (GRIS) in the second level, which query the IPs and update the cache with the resource data; and *grid information index service* (GIIS) in the top level, which indexes the resource information provided by other registered GRISs and GIISs.

Gridbus Middleware

Grid middleware such as *Globus toolkit* provides secure access and execution of the jobs in the grid. The technologies required to realize utility computing, such as the resource brokering technologies, scheduling data-driven applications, data management strategies, accounting of resource consumption, are provided in the *Gridbus* project. Gridbus hides the details of resources and the low level middleware technologies from the application developers, thus providing the user-level grid services. Gridbus provides various software technologies such as

[7]`http://www.globus.org/toolkit/`

alchemi, for the enterprise grid middleware; *gridbus broker*, for grid resource brokering and scheduling; *gridbus workflow engine*, for grid workflow management; and *visual parametric modeller*, for the grid application programming interface.

2.2.4 Issues and Challenges

Issues of concern in grid computing include security, availability, and reliability. Resources of a grid can come from various organizations and the security policies may differ, so the *integrity* and *confidentiality* of the data processed on the grid is a *security* concern. *Authentication*, *authorization*, *encryption*, and *confidentiality* communication mechanisms, and *redundant computing* can be used to handle the security concerns in the grid. *Availability* of the grid resources can be increased by maintaining redundant resources in the grid. *Reliability* on a grid can be improved by *autonomic computing*, a self-management software that helps in improving the reliability by re-submitting the jobs to other machines in the grid in case of inconsistencies.

2.2.5 Cluster Computing vs Grid Computing

A cluster and a grid are formed by combining together multiple computers but they differ in many ways. Nodes on a cluster are tightly-coupled, homogeneous, and dedicated. The nodes on a grid can be loosely-coupled, heterogeneous, and they can be either dedicated or non-dedicated. Nodes on a grid can make use of spare computing power of a desktop computer. Resources of a grid are geographically distributed, whereas the nodes in a cluster in a close proximity generally at a single physical location. A grid is more scalable compared to a cluster as the nodes in the grid are heterogeneous and any machine (with any configuration and platform) can be added. *Fine-grained* parallel problems can be efficiently run on a cluster in contrast to the *coarse-grained* parallel problems and problems composed of independent tasks run on a grid. A cluster has a single system view and resources are managed by a centralized resource manager, whereas each node in a grid is an autonomous independent entity.

2.3 Volunteer Computing

Volunteer computing [23] is a form of distributed computing where the computing resources are volunteered by anonymous people from different geographical locations. The computing power gathered using the volunteer computing is huge and it is also cost-effective as the resources (computing power) are voluntarily donated by anonymous donors from around the globe. Example volunteer computing projects are *Search for Extraterrestrial Intelligence (SETI)* and *Folding@home*. SETI@home[8] project analyzes the data gathered by radio telescopes in search of evidence for extraterrestrial communications, and Folding@home[9] project studies the way proteins take certain shapes (called folds) and how the folds relate to the work of the proteins.

2.3.1 Middleware Management Systems

A Middleware system provides services to schedule jobs, resource management, and workload management. The two major players in providing the middleware software solution to volunteer and grid computing are *BOINC* and *Frontier*. BOINC and Frontier have paved a path to volunteer computing by enabling any regular user to donate their idle CPU cycles in executing large data and compute intensive tasks.

BOINC

BOINC (Berkeley Open Infrastructure for Network Computing)[24] is an open-source software developed at University of California, Berkeley. BOINC is used for compute intensive projects in scientific research.

BOINC software consists of server software and client software. *Server software* is used to create projects. It includes a relational database to store the details of the applications such as the application descriptions, platforms, versions, work units, results, accounts, teams etc. *Client software* is a simple downloadable software to run BOINC on the volunteers computers on all the major platforms. Installing the BOINC client software makes a computer eligible

[8]`http://setiathome.berkeley.edu/`
[9]`http://folding.stanford.edu/`

to volunteer its CPU cycles for an available project. The software can operate either as a screen saver application or a service on the volunteer computer. BOINC provides tools to manage projects. The web services and daemon processes handle the server functions, and the *scheduling servers* issues work and handles results from the clients. The *data servers* handles the valid file uploads.

Redundant computing is used in BOINC to validate the results computed by the volunteer computers. *Failure and backoff* is used to manage the connection overload by the participants. Volunteers have the option to choose on how and when their resources can be used by *participant preferences*. Resources donated and the credit earned by the volunteers is tracked using the *accounting system*. A *local scheduling policy* is used to decide on which client to volunteer and execute the tasks for a project.

Frontier

Frontier[10] is a software solution for grid-computing by Parabon Computaion, Inc. Using Frontier, virtual-private grids and public grids can be built using the dedicated and idle resources. Data and compute intensive jobs can be executed on the Frontier grid. Frontier allows organizations to build their own private grids by utilizing the computing resources available in the organization. The resource management can be handled either by the organization itself or the Parabon Computation, Inc.

An organization can build their own grid using the *Frontier Enterprise* grid software. *Frontier Grid Server* is the heart of the grid, which manages the scheduling of the jobs and the management of the resources. It can be installed on a single or multiple servers. *Frontier Compute Engine* a downloadable software, can be installed on the provider machines to execute the tasks of jobs. *Frontier Dashboard* a web-interface, allows to launch jobs, view results, and monitor usage of resources. Frontier allows any computer user to donate their CPU cycles to execute the tasks of jobs. This can be simply done by downloading the *Frontier Compute Engine* software and installing on the computer. Frontier also provides tools to create Frontier-enabled applications that can be run on Frontier.

Though the scheduling algorithm or the policies utilized in executing the tasks of jobs

[10]`http://www.parabon.com/`

or how the resources are managed in a grid is unknown because of the limited available information, Frontier makes sure it ensures the security and the confidentiality of the user jobs and the resources. It uses various security mechanisms like encryption of the user jobs (tasks), authentication of resources, and following a *double blind* procedure. In double blind procedure, the provider of the resources is unaware of the source of the tasks that are running on the resource, and the users of the jobs are unaware of the nodes on which their task is executed. Massive parallelism and redundancy are used to ensure availability of the resources (nodes) to the user jobs. Result tampering is avoided by the redundancy checks of the results obtained by running a task of a job on multiple nodes.

2.3.2 Issues and Challenges

Volunteer computing is a very flexible and cost-effective form of computing among the various distributed computing. With the advantage of using the computing power donated by individuals, comes some of the vulnerabilities. Challenges of volunteer computing include the heterogeneity, reliability of the results, scalability, and security. As the resources are donated by anonymous volunteers they are *heterogeneous* with respect to hardware, software, and network speed. This is an issue because the resources are not dedicated for the purpose and it is difficult to estimate the number of resources required for a job and the time for the completion of a job. This issue can be overcome by the use of *virtual machines*. The results given by the volunteer computers may not be trustworthy because of the anonymity of the resources. This can be reduced by *task replication* i.e., executing job on multiple computers and comparing the results to validate and provide *reliable* results. Volunteer computing is *highly scalable* as resources are volunteered by anonymous people from around the world. *Scalability* can be handled using a server architecture distributed across multiple machines. *Privacy and security* is a major concern as the project tasks run on machines of anonymous donors who are not accountable to the data-integrity. *Account sandboxing* can be used to handle the issue, where the volunteers will not have access to the data files of the tasks that are run on their donated resources.

2.3.3 Grid Computing vs Volunteer Computing

The resources on a grid belong to an organization(s) or a *virtual organization(s)* which regulate the resources according to the organizational policies. Therefore, the resources are accountable for the results returned by them. In volunteer computing, the resources are not accountable for the results returned for the tasks run on them, as anonymous volunteers donate participate by donating their resources for compute intensive projects. Resources in grid computing are monitored, maintained, and updated by the participating organizations, whereas the resources in volunteer computing are maintained and updated by the volunteers eliminating the maintenance cost.

2.4 Cloud Computing

Cloud computing [25, 26, 27, 28] is a form of distributed computing where the resources required by applications i.e., platform to develop the application, software, and infrastructure can be accessed as an on-demand service. A *cloud* usually uses a *grid.* In cloud computing, the functioning, handling of the devices and the resources of the cloud are abstracted from the end users. End users can request and avail of the services depending on the demand of the application using a user-interface (e.g., a web browser).

2.4.1 Characteristics of Cloud Computing

The characteristics of cloud computing include [29] high availability, scalability, on-demand metered service, ubiquitous network access, resource pooling, and elasticity. In cloud computing the consumers can avail of the required computing resources *on-demand* as a *metered service* based on the service agreement and the payment policies. It is an advantage to the client companies as they can pay only for the amount of resources the company uses. With cloud computing, the power of computing can be realized by all the internet-enabled devices (e.g., mobile phones, laptops, and PDAs), on-the-go i.e., *ubiquitous computing.* The resources in a cloud are *pooled* to serve multiple users' computing demands. The resources can be dynamically varied (assigned or released) based on the user-demand using various models such as the *multi-tenancy* or the virtualization model. Thus, a cloud is *resource elastic* in nature.

2.4.2 Hierarchical Architecture

The bottom-up hierarchical architecture of cloud computing comprises of data centers, infrastructure, platform, application, and cloud clients. *Data centers* consist of the computer hardware and software designed for the delivery of cloud services. *Infrastructure as a service (IaaS)* offers a usage-based pricing model of the computer infrastructure i.e., hardware, software and equipments required to deliver software application environments. *Platform as a service (PaaS)* provides high-level integrated environments to design, build, test, deploy, and update online applications without the need to purchase or manage the hardware/software required for the applications. Using *software as a service (SaaS)*, software is rendered as a service over the internet . SaaS eliminates the capital costs required to purchase, install, and maintain the software. *Cloud clients* consist of the end user computer hardware and software, that is specifically designed for the receipt of the cloud services for the application delivery.

2.4.3 Issues and Challenges

Security, interoperability costs, privacy, and reliability are some of the issues and challenges in cloud computing [30, 31].

The idea of putting one's data and deploying applications on a third-party resource provider is intimidating. The *security* concerns include the organization's loss of data, phishing, and botnet (network of computers using distributed computer software) [32].

The cloud computing reduces the capital and maintenance costs for a client as the clients need not purchase, install, and update the required infrastructure. But, in order to integrate cloud computing services the client company may need additional infrastructure which may increase the client company's costs, and such *interoperability costs* should be minimized.

In cloud computing there is a risk of privacy of data as the data resides on the service provider resources, which may be geographically distributed with varied privacy laws and standards. In a cloud, depending on the user-demand the resources are allocated to different users. There is a risk of data-theft as the data of a user can be recovered at a later time when the resources are accessed for another job of another user. Tasks of jobs are run on the service provider's resources where the client's data resides. Service providers have access to the data of jobs on the resources, which they can monitor and control impacting the *privacy and security.*

The resources in a cloud are heterogeneous and come from various sources (organizations or anonymous volunteers). The resources (nodes) are located at different geographical locations and are connected over a network, thus *network latency* is an issue. It is possible that the job might take more than an estimated time to complete because of network latency and *unavailability* of the resources. Thus, the *reliability* of the access of the service can be improved by maintaining redundant resources and *redundant computing.*

2.4.4 Grid Computing vs Cloud Computing

Grid computing links autonomous heterogeneous computers to form one large infrastructure to solve computationally intensive problems. Cloud computing evolves from Grid computing and provides on-demand provisioning of the resources (software, platform, and infrastructure) on the cloud [33, 34].

2.4.5 Google App Engine

Google App Engine (GAE)[11] is a cloud computing technology developed by Google. It provides the PaaS for developing and deploying the web applications on Google's infrastructure. GAE includes the Java runtime environment and the Python runtime environment. It supports applications written in standard Java technologies and Python environment. GAE includes APIs for user authentication, sending email using Google accounts, data store (a distributed data storage service with a query engine and transactions consisting of *data objects or entities* with *properties.*), task queues for performing work outside the scope of a web request, and scheduling tasks for triggering events.

[11]`http://code.google.com/appengine/`

Application Instances

Applications run on Google servers. Servers load balance the traffic and the data storage. The requests to the application are served by the instances dynamically. The number of *instances* running can vary dynamically depending on the traffic to the application. Instances include a security layer to separate from one another and each instance includes the language runtime, the App Engine APIs, application's code, and memory. A minimum of three instances can be allowed to run always for an application as *always on* irrespective of the traffic demands.

Application Management

Users can launch their applications using their Google account and manage it on the GAE. *Admin console* can be used to create and manage the applications. Data store and application's traffic can be monitored, and new versions of the application can be tested and added.

Quotas and Billing

GAE billing is controlled by *quotas*. GAE is free for limited storage providing enough CPU and bandwidth to efficiently run an application. Billing for an application can be enabled to set a maximum daily budget depending on the application needs and charging of resources is done per the billing policies when extra resources are required.

2.5 Utility Computing

Utility computing [35] is a form of distributed computing where the computing resources like hardware, computer processing power, and data storage are provided as an utility similar to the other regular utilities such as electricity. Utility computing is an advantage to companies which prefer to rent the computing resources on-demand and pay only based on their usage. Utility computing provides computing resources to the client companies and grid computing is required to gather the computing resources to provide as an utility.

2.5.1 Characteristics of Utility Computing

The main characteristics of utility computing from the client prospective include the convenience, cost, and reliability. Utility computing is not yet a well-established service and so each of the characteristics have its advantages and concerns. Using utility computing a client has the advantage of *convenience* to not worry about purchasing and maintaining the computing resources that are required by the client company. All the background work required for using the computing resources is handled by the utility computing company. Though it might seem advantageous for a company to use utility computing, it might not be a right choice in all cases. The infrastructure to take advantage of utility computing might surpass the capital and maintenance costs of the resources the client company intends to get service from the utility company. Thus, *cost* can be either an advantage or disadvantage depending on the client company requirements, policies, and the quality of service of the utility computing company. *Reliability* depends on how effectively the utility computing company can provide service without major interruptions to the service. An interruption may lead to loss to time, resources, and data.

2.5.2 Issues and Challenges

Major issues and challenges of utility computing include billing, quality management and technology capabilities, interoperability costs, operational costs, and deployment time. To stabilize the utility computing, companies need to come up with *policies* and *standards* for *billing* the computing resources. The utility computing companies also need to establish certain *quality of service standards* in the business and reevaluate the standards with time and follow *cutting-edge technology standards.* An advantage of the utility computing is, the client companies pay only for the amount of resources utilized. However, for the efficient use of the resources the client company might need to procure *additional infrastructure* (hardware/software) and these costs need to be minimized. Also, the *operating costs* need to be minimized as the businesses of the client companies may grow and the amount of computing resources utilized by the client companies may grow with time. Using utility computing the *cost*, *deployment time*, and the use of *additional infrastructure* must be minimized compared to the costs when the client companies internal resources are utilized.

2.5.3 Cloud Computing vs Utility Computing

Utility computing is a business model to provide computing resources on pay-by-use basis. Cloud computing is a wider concept which provide services (hardware or software) required for the development cycle of software applications and the resources can be changed depending on the requirement and the availability.

2.6 Conclusion

There are many choices available for distributed computing. Many of these choices are well suited for parallel Monte Carlo simulation. For the academic computing environment envisioned by this research, a combination of cluster and volunteer computing is a reasonable choice. A cluster computer of modest size can be used to provide the computing power required for most applications, while volunteer computing can be used to freely assemble a grid of additional resources when they are needed.

Chapter 3

Scheduling for a Multi User Environment

3.1 Theoretical Background

Let X be a random variable, N be the total number of trials, and S be the number of successes in N, trials. The goal of the simulator is to estimate the mean of X. In a simulation, a trial is a Bernoulli trial with outcomes 0 and 1. Success outcome is 1 with probability, p, and failure outcome is 0 with probability, $1 - p$. The mean of X can be calculated as below:

$$\hat{M_X} = \frac{1}{N}\sum_{i=1}^{N} x_i \quad = \quad \frac{S}{N} \tag{3.1}$$

Parallel Monte Carlo Simulation

Using distributed computing, a simulation can be quickly performed by dividing a job into multiple tasks and executing them in parallel. A *task* is a set of trials that may be run in parallel with other tasks. Of the large number of trials required to complete a simulation, a set of trials are run independently on each of the available machines. Let $x_{n,j}$ be the n^{th} realization of the j^{th} task associated with X, N_j be the number of trials in the j^{th} task, and k is the total number of tasks required to complete a job(k, is unknown until the job is completed). Using the parallel Monte Carlo simulation method, the expected value of mean can be calculated as

$$\begin{aligned} \hat{M_X} &= \frac{1}{\sum_{j=1}^{k} N_j} \left(\sum_{n=1}^{N_1} x_{n,1} + \sum_{n=1}^{N_2} x_{n,2} + \cdots + \sum_{n=1}^{N_k} x_{n,k} \right) \\ M_X &= \frac{1}{\sum_{j=1}^{k} N_j} \left(\sum_{j=1}^{k} \sum_{n=1}^{N_j} x_{n,j} \right) \end{aligned} \tag{3.2}$$

3.2 System Operation

Jobs in a distributed computing system can be scheduled using many existing scheduling algorithms such as stride scheduling [36], lottery scheduling [37], charge-based proportional scheduling [38], fair share scheduler [39], and other proportional share algorithms [40] that consider the processing time utilized by the users and proportionally allocate resources relative to their share. The execute a Monte Carlo simulation in parallel making use of distributed computing a scheduling algorithm is developed to schedule and execute jobs efficiently. An analytical model is created to test the system behavior when the parameters of the algorithm are varied. In this section we discuss the system operation of the analytical model and then present the algorithm in the following section.

A set of users in the system is represented as, $\mathcal{U} = \{U_1, U_2, \ldots, U_d\}$. Users submit jobs to the system, and the jobs are enqueued in the jobs queue. A single user can have multiple jobs. *Job manager* is a process, which checks for the newly arrived jobs in the queue, checks for the resources in a time slot, assigns priorities to jobs, combines tasks results of a job, and updates the jobs. Each time period, t_1, *job manager* checks for new jobs in the queue. Let t_2 be the time period of one *time slot*. As each worker executes a task in a time slot, the time period of execution of a task is t_2. However, a task is an integer number of trials, so at the end of time period, t_2, if the task is midway of a trial then the task ends after the completion of that particular trial. Thus, the actual time period, t_2, is random for each task. Let n be the number of jobs in queue in time slot, ℓ. A set of jobs in the queue is represented by, $\mathcal{J} = \{J_{i,1,\ell}, J_{i+1,2,\ell}, \ldots, J_{p,n,\ell}\}$ $(p \leq d)$, where $J_{i,j,\ell}$ is the j^th job in the queue of time slot, ℓ, and it belongs to user, i.

At the beginning of each time slot, job manager assigns a *priority*, a value in the interval $[0, 1]$ to each job in the queue. The priority is assigned using the scheduling algorithm defined in the following sections. The total number of computing resources (workers) available to service jobs in a time slot be, C. A *worker* is a computing resource that services/executes a task. A worker can service a single task in a time slot.

A worker picks a job for execution as follows; for each job using its priority, a priority interval (a lower and an upper bound of the priority in the range $[0, 1]$) is constructed. Each worker generates a uniform random number between $[0, 1]$, if the random number is in the priority interval of a job then that particular job is picked to be serviced by the worker. As workers independently choose a job to service by generating a random number, the number of workers servicing a job (i.e., the number of tasks executed for a job) is random in a time slot. At the end of execution of a task each worker saves the results.

Job manager keeps track of the execution results of all the tasks of a job and saves the consolidated data at the end of each time slot. For each user, job manager keeps track of the number of tasks executed. This data is necessary in calculating the priorities of jobs in the queue, discussed as part of the scheduling algorithm. The *total processor time* required to complete a job is, $\Gamma_p = kt_2$, where k is the number of tasks required to complete a job (k is unknown until job completion) and the *duration* of a job is, $\Gamma_c = \delta t_2$, where δ is the number of time slots required to complete the execution of a job (from the time slot the job is submitted to the system to the time slot the job completes execution).

In the analytical model, the number of successes, S, is required to complete a job (S, is an input parameter). Number of trials generated in i^{th} task of a job, N_i, is fixed. The number of successes observed in a task is pN_i where, p, is the success probability of a Bernoulli trial. Thus, the number of expected tasks required to observe S, successes for a job is, $\hat{k} = \frac{S}{pN_i}$. Assuming that the job can be completed in $\hat{k}$, tasks simplifies the calculation of all the intermediate values in the algorithm (e.g., remaining percent of the job can now be obtained using the number of remaining tasks, unlike calculating it using total number of required successes, S; number of remaining successes; and the number of trials required to observe the completed successes, if the above approximation of using $\hat{k}$, is not done). The actual number of tasks, k, required to complete a job can differ from $\hat{k}$, as the number of tasks serviced for a job in a time slot is random.

3.3 Proportionally Fair Scheduling Algorithm Design

In the scheduling algorithm discussed in the following sections, the *exponentially-weighted window* is used to calculate the *cumulative processing power* utilized by a user. An exponentially-weighted window is an efficient method to keep track of processing power utilized by the users as it do not have to keep track of the data in each time slot. It is sufficient to keep track of the data in the previous time slot i.e., the number of computing resources and the exponential cumulative processing power value of a user in the previous time slot.

Parameter α, the *forgetting factor* is used in evaluating the exponential cumulative processing power utilized by the users. $\alpha = 0$ implies only the number of computing resources utilized in the previous time slot are considered to calculate the exponential cumulative processing share of a user (i.e., the exponential processing power utilized through the previous time slots is not considered). For $\alpha = 1$, exponential cumulative processing power utilized is the sum of the resources used by a user until the current time slot. The preferable range of values of α is $0 < \alpha < 1$.

The second parameter, β, moderates the processing power utilized by a user. The preferable range of values for β is $[-1, 0]$. For $\beta = -1$, the weight assigned to a job of a user is inversely proportional to the processing power utilized by a user, i.e., if a user has utilized more processing power then a job of that particular user gets less weight. Thus, the job gets a low priority. For $\beta = 1$, the weight assigned to a job of a user is directly proportional to the processing power utilized by the user and so the job is assigned a high priority. The priority assigned to a job reverses from $\beta = -1$ to $\beta = 1$. If $\beta = 0$, then processing power utilized by a user is not considered in evaluating the weight of a job. All the jobs in the queue are assigned an equal priority and this is *equal job share.* The exponential cumulative processing power utilized by each user is moderated with value β, to assign a weight to each job in the queue. A normalized weight assigned to each job gives the priority of the job, a value in the interval $[0, 1]$. The number of workers servicing a job depends on the priority of the job, more workers service a job with high priority.

A job can be over done, if more than the required number of workers service a job. This is often observed in the following two cases. Case 1 is observed when a job gets high priority towards the end of its completion, and case 2, when a job is small and it gets an initial high priority. The second case arises because the system is ignorant of the number of workers required to complete a job unless a part of the job is serviced. The first case is handled by determining if a job is towards the end of its completion. Let c_1, be the the remaining percent of a job at which it can be labeled as case 1. $\hat{k}_j$, be the expected number of tasks required to complete a job j in the queue, and $C_{i,j,\ell}$, be the number of completed tasks of a job, j, in time slot, ℓ, of a user, i, then the remaining number of tasks of the job is, $R_{i,j,\ell} = \hat{k_j} - C_{i,j,\ell}$. The remaining percent of job is given by, $r_{i,j,\ell} = \frac{R_{i,j,\ell}100}{\hat{k}_j}$. The algorithm avoids assigning a high priority to a job by comparing the assigned priority of a job (calculated using the weight of a job in the time slot) to the expected priority (calculated using $r_{i,j,\ell}$) to avoid over work. The expected priority is given by, $\hat{P_{i,j,\ell}} = \frac{R_{i,j,\ell}}{C}$, normalized with the total number of available computing resources, C, in a time slot. If the assigned priority, $P_{i,j,\ell}$, of a job is greater than the expected priority, $\hat{P_{i,j,\ell}}$, then adjust the priority of the job to the expected priority. In case 2, i.e., when the number of completed tasks of a job is zero, a predetermined percentage of workers, c_2, (c_2 value converted into $[0,1]$ range) is assigned as the priority of the job.

3.3.1 Scheduling Algorithm

The following steps explain the process of assigning priorities to the jobs in the queue.

1. An exponentially-weighted window is used to calculate the processing power utilized by the users. Through time slot, ℓ, the computing resources utilized by an user, i, is given by

$$E_{i,\ell} = \alpha E_{i,\ell-1} + \tau_{i,\ell-1}, \tag{3.3}$$

   where, $E_{i,\ell}$, is the exponentially-weighted cumulative processing power utilized by user, i, through time slot, ℓ; $\tau_{i,\ell}$, is the number of tasks executed for user, i, in time slot, ℓ; and α, is the forgetting factor.

   For $\ell = 1, E_{i,1} = \alpha 1 + 0$, i.e., $E_{i,0} = 1$ and $\tau_{i,0} = 0$.

2. The weightage of the available resources that can be assigned to a job, $J_{i,j,\ell}$ i.e., j^{th}

job in queue of user, i, in time slot, ℓ, is

$$W_{i,j,\ell} = E_{i,\ell}^{\beta} \tag{3.4}$$

In equation (3.4), for $\beta = 0$ all the jobs in the queue get equal weight, *equal job share.*

3. The priority of a job, j, of user, i, in time slot, ℓ, is

$$P_{i,j,\ell} = \frac{W_{i,j,\ell}}{W_{1,1,\ell} + W_{2,2,\ell} + \cdots + W_{d,n,\ell}}, 0 \leq P_{i,j,\ell} \leq 1 \tag{3.5}$$

If $C_{i,j,\ell} > 0$, $r_{i,j,\ell} \leq c_1$, and $P_{i,j,\ell} > \hat{P_{i,j,\ell}}$ then assign $P_{i,j,\ell} = \frac{R_{i,j,\ell}}{C}$. If $C_{i,j,\ell} = 0$, then assign $P_{i,j,\ell} = c_2$, (c_2 value converted into $[0, 1]$ range).

The above statement is executed if the number of jobs in queue is greater than the sum of the new jobs and the jobs that are about to complete (otherwise, the above statement is not executed because in a time slot we do not anticipate to have workers that does not execute any task, as we want to maximize the cluster utilization; and also processing more than the required number of tasks for a job gives more precise results). As over work is handled, the sum of the priorities of the jobs can be less than 1, however, as cluster utilization should be maximized, the priorities of jobs with $C_{i,j,k} > 0$, and $r_{i,j,k} \geq c_1$, are recalculated and assigned.

4. In each time slot, for each job a priority interval (a lower and an upper bound of the priority in the range $[0, 1]$) is constructed. Each worker independently generates a uniform random number in the interval $[0, 1]$. Worker checks the priority interval of each job and picks a job to service for which the generated random number is in the range of the priority interval.

5. The number of tasks serviced (including the time required to complete each task) and the number of successes observed for each job in a time slot are tracked.

6. (a) At the end of each time slot i.e., at the end of time period, t_2 a check is performed to make sure if the number of tasks serviced for each job were sufficient to complete the execution of the job.

(b) If the number of completed tasks are sufficient, corresponding jobs' status are marked as *Done*.

(c) Jobs with status *Done* are not included in the list of jobs that need to be scheduled in subsequent time slots.

(d) A check is performed to determine the arrival of new jobs into the jobs queue. All the new jobs are taken into consideration to be scheduled in the next time slot.

(e) For all the jobs in the queue, priorities are recalculated, scheduled, and executed using steps 1 through 4.

This algorithm assigns weight to a job of a user depending on the β value. If $\beta > 0$, then a job of a user who utilized more cumulative processing power is assigned a greater weight compared to a job of a user with less cumulative processing share. If $\beta < 0$, then a job of a user with less cumulative processing share is assigned a greater weight compared to a job of a user with more cumulative processing share. Thus, the priority assigned to a job in a time slot depends on the exponential cumulative processing share (moderated using the parameter, α) and the value of the parameter, β.

3.3.2 Effect of α and β Parameters on the System

An analytical model is created to observe the effect of the parameters, α, and β, on the system. Number of users in the system are, $d = 10$, and the number of available computing resources in a time slot, $C = 100$. Among 10 users, user, U_1, submits only one big job, $J_{1,1,1}$, which is 20 timer bigger than a regular job. The job submission probability of a user (other than user, U_1) is , $\rho = 0.008$. The total expected duration to analyze the system behavior is 10000 time slots, where the actual duration may take a little longer, as the analysis is complete for a set of parameters when all the jobs in the system complete execution. The actual duration, $T > 10000$, for the above parameters. Using, ρ, and 10000 time slots, a matrix of users and time slots is created. This matrix provides information on which user submits a new job to the system in which time slot. From the matrix, the total number of jobs submitted to the system are 539 (538 regular jobs and 1 big job of user, U_1).

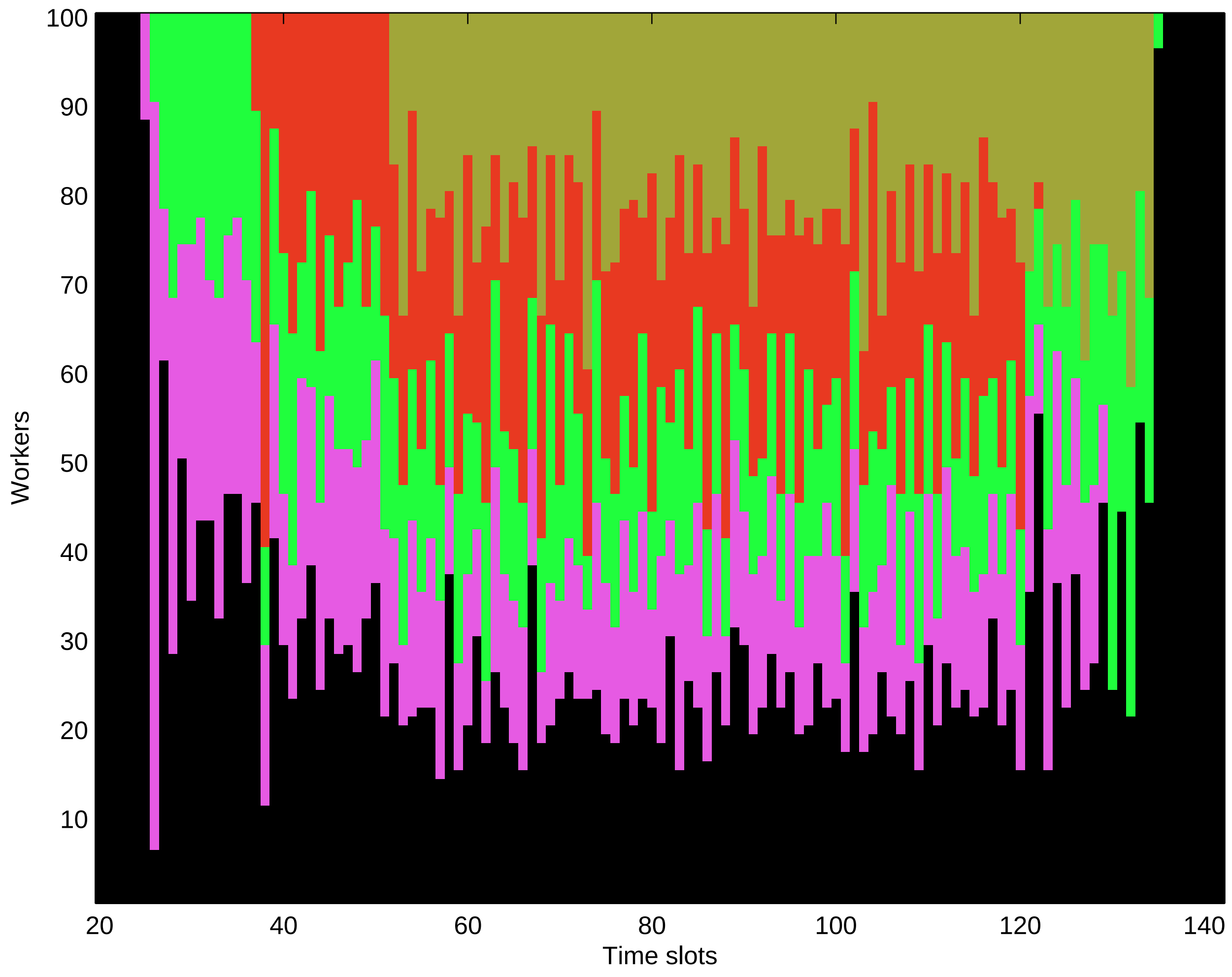


Figure 3.1: Number of workers executing each of the jobs over the time slots, $alpha = 0.25$ and $\beta = -1$

The input parameters for each job include the success probability of a trial, $p = 0.01$, the number of trials per task (task i) of a job, $N_i = 1000$, and the number of successes required for the completion of the job, $S = 20000$. Therefore, $\hat{k} = \frac{S}{pN_i}$, for a regular job, $\hat{k} = 2000$; and $\hat{k}$, of $J_{1,1,1}$ (of user U_1) with $S = 400000$, is, $\hat{k} = 40000$. Let $t_2 = 1$ (a unit time period).

In order to understand how the jobs are executed by the workers in the system, we shall examine Figure 3.1. The duration (number of time slots) required to complete a job depends on the priority assigned to a job in each of the time slots. In each time slot, the tasks of a job may be serviced by different workers. The number of workers executing a job in a time slot is proportional to the priority assigned to the job. In Figure 3.1, we examine the execution of each of the jobs, Job 1 - 5 from time slots 20 - 140 with parameters $\alpha = 0.25$ and $\beta = -1$.

Figure 3.1, shows the tasks of each job sorted together in a time slot. x-axis represents *time slots*, y-axis is *workers*, and each of the jobs (tasks of the jobs) serviced by the workers in the time slots is represented with a different color. In Figure 3.1, color black represents Job 1, pink for Job 2, green for Job 3, red for Job 4, and brass for Job 5. Job 1 execution starts in time slot 1, and as it is the only job in the system all the workers service it until Job 2 is added in time slot 25. As more jobs are added by the users into the system, the number of workers servicing a job varies over the time slots depending on the job priority. Jobs 2 - 5 start in between time slot 25 and end in time slot 134. In time slot 135 and in the later time slots, all the workers service Job 1 as it is the only job in the system, until more jobs are added to the system.

Effect of α on the System

First, let's take the parameter, α, and observe the system behavior when α, value varies. In the algorithm, we have two parameters, α, and β. To observe the effect of α, we set β, to -1, i.e., the priority assigned to a job is proportional to the exponential cumulative processing share of the user. We chose $\beta = -1$, as proportional share is preferred in most of the scheduling mechanisms.

α, is used in calculating the exponential cumulative processing share of a user. For user, i, in time slot, ℓ, a small value of, α, assigns less weight to $E_{i,\ell-1}$, decreasing the value of exponential cumulative processing share, $E_{i,\ell}$, of a user in current time slot. A small value of $E_{i,\ell}$, gives a high priority to a job (with $\beta = -1$), that means the job should complete in a small number of time slots. Similarly, a high value of α, gives more weight to $E_{i,\ell}$, and so a low priority is assigned to the job, which means job requires longer duration to complete. The system behavior is observed for $0.25 \leq \alpha \leq 1$. The duration of big job, $J_{1,1,1}$, and the mean duration of the regular jobs are provided in Table 3.1. For a small value of α, $J_{1,1,1}$, is executed in a short duration and the duration is observed to increase with an increase in the value of α.

The mean duration of the regular jobs is observed to be reverse of the duration of the big job, $J_{1,1,1}$, shown in Fig. 3.2. With an increase in α, $J_{1,1,1}$, is assigned a low priority over the time, and more workers service the regular jobs decreasing the mean duration of the regular

Table 3.1: Duration of big job and the mean duration of regular jobs with $\beta = -1$, and different values of α

α	β	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.25	-1	11164	40000	3385	2000.7	1923.8
0.5	-1	11165	40000	3538	2000.9	1923.1
0.75	-1	11164	40000	3590	2000.7	1920.4
0.85	-1	11164	40001	3665	2000.7	1919.8
0.95	-1	11164	40000	3793	2000.7	1920.2
0.98	-1	11164	40001	4098	2000.7	1916.6
0.99	-1	11165	40000	4474	2000.9	1914.2
0.9999	-1	11165	40001	9821	2000.9	1809.2
1	-1	11165	40001	10564	2000.9	1791.4

jobs.

Effect of β on the System

To observe the effect of β, we fix the value of α, and vary the values of β. The preferred range of values of β is, $[-1, 0]$. Weight of a job is proportional to the β power of the exponential cumulative processing share of a user, equation (3.7). We did not consider positive values of β, because, in most scheduling scenarios users' with high processing power are not preferred over users' with low processing power.

For a fixed set of α and β values, the exponential cumulative processing share value, $E_{i,\ell}$, is high for a user, i, in time slot, ℓ, with a large number of executed tasks. If an user's jobs are not serviced by the workers over certain number of time slots, then $E_{i,\ell}$, value decreases. The weight of a job is not dependent on the value of β, alone, but is also affected by the value of α. The priority of the job in a time slot is dependent on the weight of the other jobs, equation (3.5). The value of the parameter, α, and the weight of the other jobs influence the system behavior, and the number of tasks serviced for the big job alternates over the time slots, i.e., the number of tasks serviced increase with an increase in β for few time slots, and then the number of tasks serviced decreases. However, for each of the β values, about 43% of the job is observed to complete in the first $330 - 350$ time slots (initial 4% of time slots) and then the behavior stops fluctuating as described above, shown in Fig. 3.3 for $\alpha = 0.95$, and in Fig. 3.4 for $\alpha = 1$. From the data we can conclude that with an increase in the value of β more number of time slots are required to complete the remaining job ($J_{1,1,1}$) i.e.,

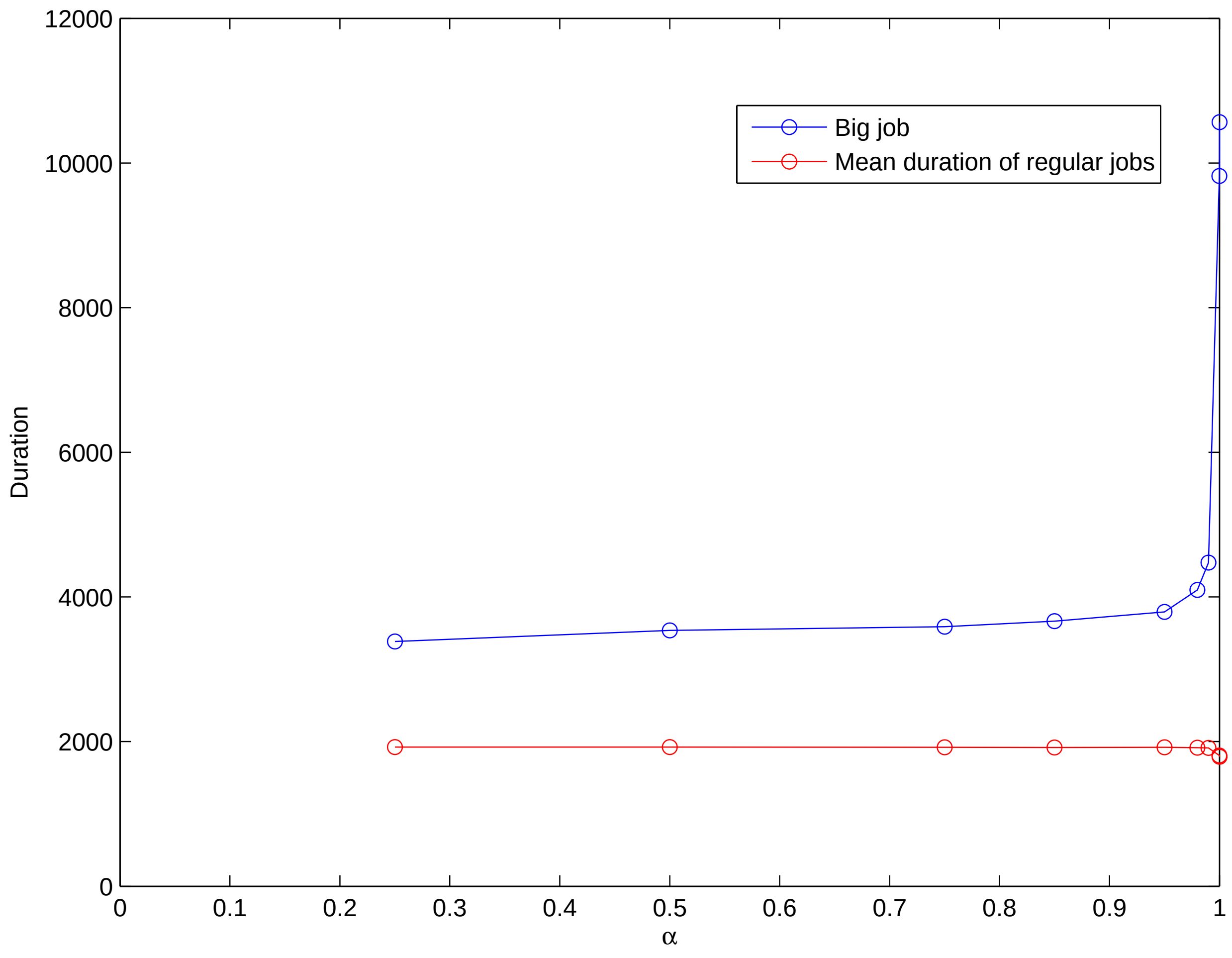


Figure 3.2: Duration of big job and the mean duration of regular jobs with $\beta = -1$, and different values of α

Table 3.2: Duration of big job and the mean duration of regular jobs with $\alpha = 0.5$, and different values of β

α	β	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.5	-1	11165	40000	3538	2000.9	1923.1
0.5	-0.75	11164	40000	4239	2000.5	1921.5
0.5	-0.5	11164	40000	5567	2000.7	1904.5
0.5	0	11168	40063	11168	2001.4	1786.6

Table 3.3: Duration of big job and the mean duration of regular jobs with $\alpha = 0.95$, and different values of β

α	β	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.95	-1	11164	40000	3793	2000.7	1920.2
0.95	-0.75	11164	40000	4528	2000.7	1912.3
0.95	-0.5	11165	40000	5936	2000.9	1902.1
0.95	0	11168	40026	11168	2001.4	1791.2

Table 3.4: Duration of big job and the mean duration of regular jobs with $\alpha = 1$, and different values of β

α	β	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
1	-1	11165	40001	10564	2000.9	1791.4
1	-0.75	11165	40002	11012	2000.9	1786.2
1	-0.5	11164	40050	11164	2000.7	1780.9
1	0	11167	40062	11167	2001.2	1787.5

remaining 57% of the big job take a large number of time slots (about 96% of time slots).

We also observed the system behavior by varying α value for a set of β values (in the range [-1, 0]). Increasing the value of α, for a fixed value of β, we observed that big job takes long duration to complete. This is expected and follows the reasoning given in section 3.3.2. Data for this system behavior is shown in Tables 3.2 - 3.4. For α very close to 1 i.e., $\alpha = 0.9999$, 1, the duration of big job, $J_{1,1,1}$, is observed to increase by a large fold, data provided in Table 3.4. The reason for this behavior is, now the exponential cumulative processing share of a user is the sum of the tasks serviced for an user (exponential cumulative processing share value, $E_{i,\ell}$, never decreases in this case unlike when $\alpha < 1$).

For regular jobs, the mean duration of the jobs is observed to decrease with an increase in the value of β. As the duration of big job, $J_{1,1,1}$, increases with an increase in the β, more workers service the regular jobs decreasing the mean duration of the regular jobs, shown in Fig. 3.5.

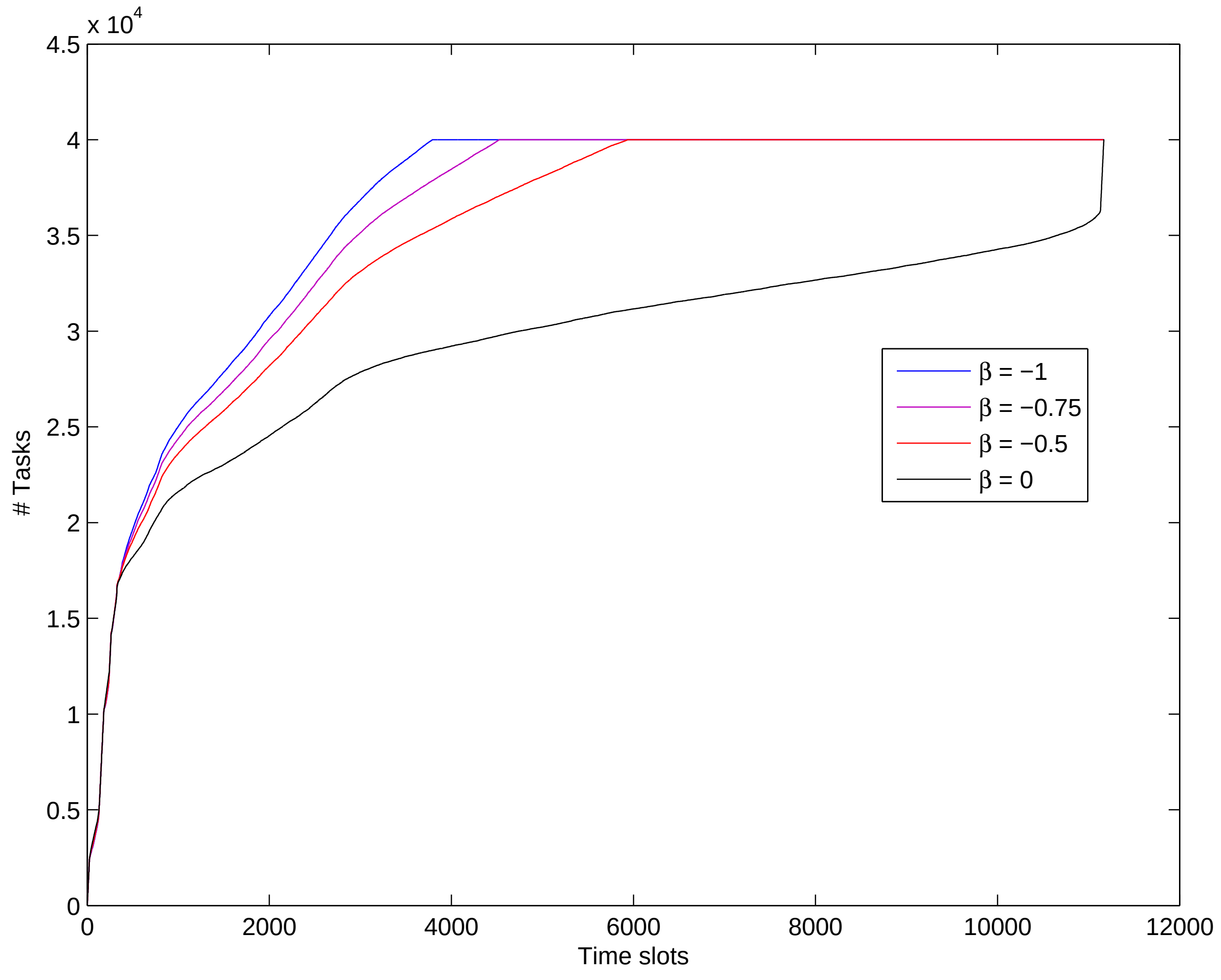


Figure 3.3: Cumulative tasks of big job for $\alpha = 0.95$, and different values of β

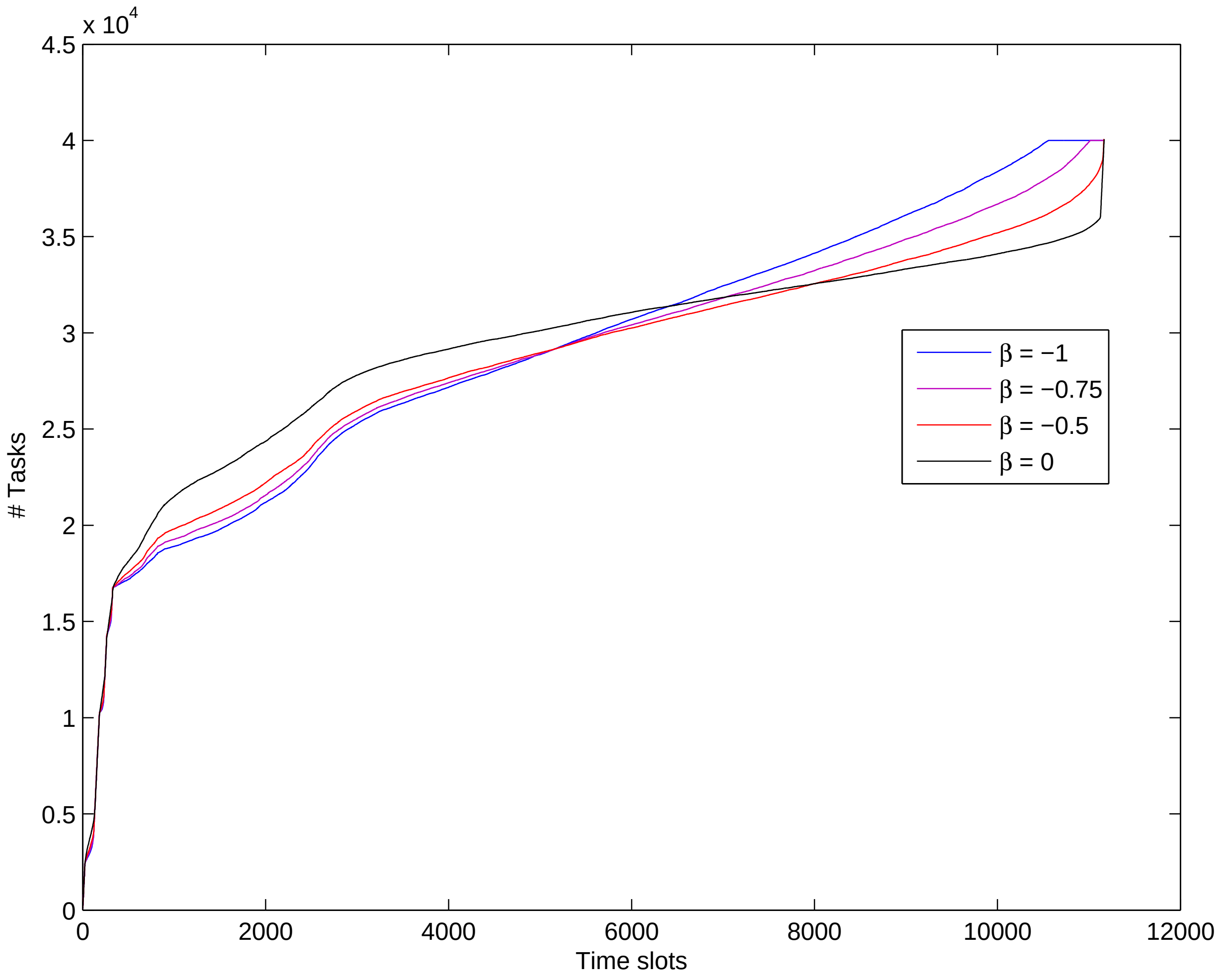


Figure 3.4: Cumulative tasks of big job for $\alpha = 1$, and different values of β

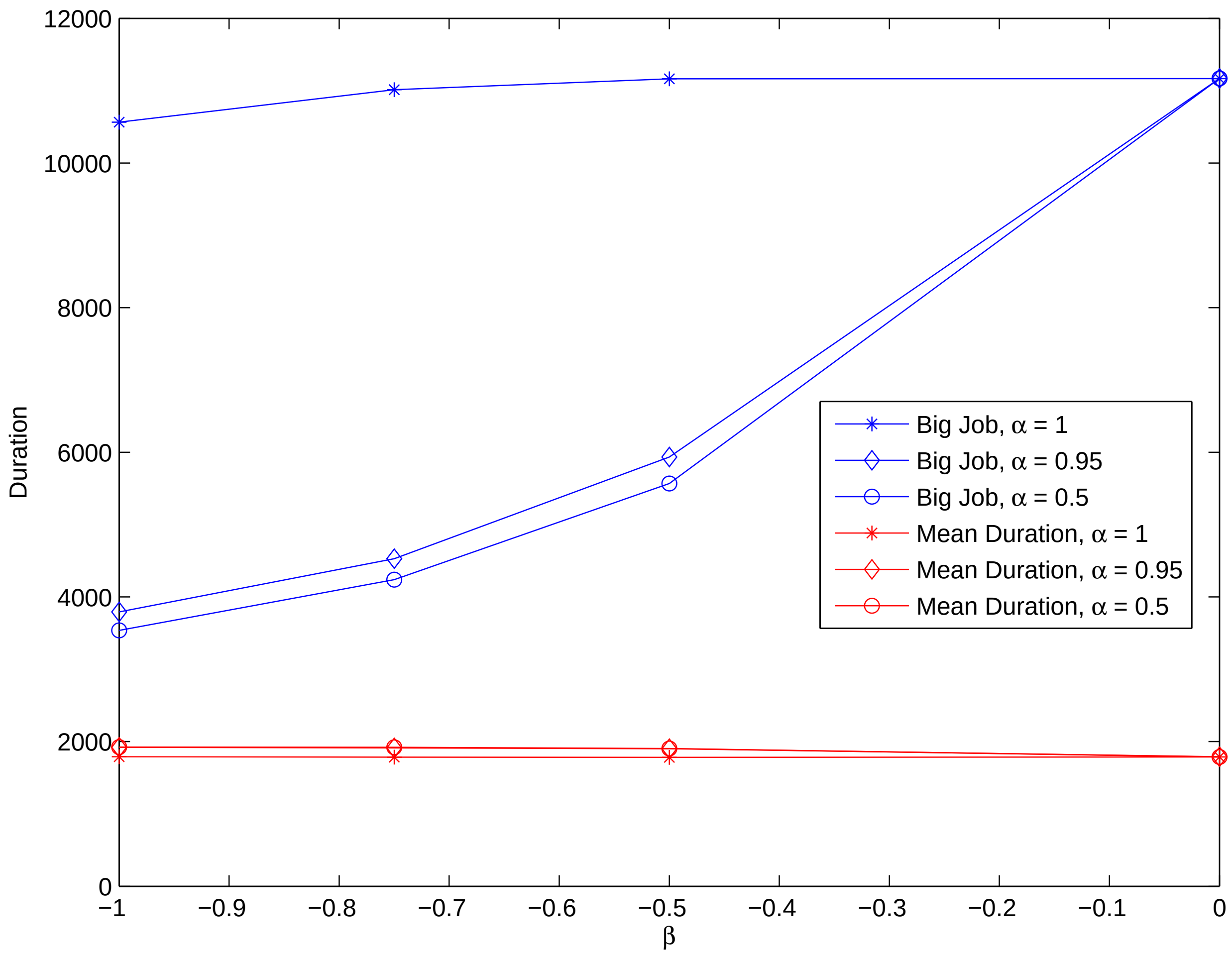


Figure 3.5: Duration of big job and the mean duration of regular jobs with $\alpha = 0.5, 0.95, 1$; and different values of β

3.4 Refined Algorithm with Job Factor

A variant of the algorithm described in section 3.3.1 adds another parameter, the job factor. *Job factor* is the expected number of remaining tasks required to complete a job. Job factor of job, j, in time slot, ℓ, of user, i, is represented by $F_{i,j,\ell}$. If the number of completed tasks of the job is $C_{i,j,\ell}$, and $R_{i,j,\ell}$, the remaining number of tasks required to complete, then the job factor is given by

$$F_{i,j,\ell} = R_{i,j,\ell} \tag{3.6}$$

when $C_{i,j,\ell} = 0$, a pre-determined percentage of workers, c_2, is used to assign the priority of the job.

Here, job factor is $R_{i,j,\ell}$, because of our assumption that in each task a certain number of successes are observed. But if we don't make this assumption, we need the knowledge of the number of tasks completed, number of completed trials, remaining successes to observe (we need to keep track of this data), and the total number of required successes to observe for a job.

Weight contributed to a job by job factor is moderated by a third parameter of the algorithm, γ. Job factor raised to the power of γ (along with α and β parameters) is used in assigning a weight to the job. Using γ, jobs of users with high cumulative exponential processing share, $E_{i,\ell}$ are not ignored. For $\gamma > 0$, the weight contributed by the job factor is high, if there are a large number of remaining tasks. For $\gamma < 0$, the weight contributed by the job factor is less, if there are a large number of remaining tasks. $\gamma = 0$, implies job factor is not considered in calculating priority of the job. For $\gamma = 0$, and $\beta = 0$, all jobs in the queue get equal priorities, equal job share scenario.

3.4.1 Refined Scheduling Algorithm

The algorithm with the addition of third parameter is as follows:

1. Step 1 of the algorithm is same as step 1 in the 3.3.1 algorithm.

2. The weightage of the available resources that can be assigned to a job, $J_{i,j,\ell}$, j^{th} job in

queue of user, i, in time slot, ℓ, is given by

$$W_{i,j,\ell} = F_{i,j,\ell}^{\gamma} E_{i,\ell}^{\beta} \tag{3.7}$$

In equation 3.7 for $\beta = 0$, and $\gamma = 0$, gives *equal job share.*

3. Steps 3-6 are same as steps 3-6 in 3.3.1 algorithm.

3.4.2 Effect of α, β, and γ Parameters on the System

To observe the effect of the parameters, α, β, and γ, in the algorithm, described in section 3.4.1 we used the analytical model described in section 3.3.2. In this section, the system behavior is observed by varying α, and β, values when the third parameter, γ, (fixed value) is considered in calculating the priority of a job. We also observe the effect of γ, by fixing the values of α, and β. Also, the combined effect of the parameters α, and β; and α, and γ, on the system is also observed.

Effect of α on the System

The system behavior is analyzed to observe the effect of α, in the presence of the third parameter, γ (used with job factor). A high value of γ, implies more weight is assigned to the job if it requires a large number of expected tasks to complete (a big job). To observe the effect of α, we set $\beta = -1$, i.e., the priority assigned to a job is proportional to the exponential cumulative processing share of a user, and $\gamma = 1$. With $\gamma = 1$, a big job gets more weight compared to other jobs. Thus, big job, $J_{1,1,1}$, completes in a short duration compared to the duration when job factor is not considered in assigning priority of a job.

Comparing the duration of the big job for different values of α, we conclude that the duration to complete a big job increases with increase in the value of α. This behavior is expected, for a fixed β, and γ, values a small value of α, implies less exponential cumulative processing share value, thus a high priority is assigned to a job in a time slot. Similarly, a high value of α, gives a high cumulative processing share value, thus a low priority is assigned to a job. Table 3.5 provides data for, $0.25 \leq \alpha \leq 1$, the duration of big job, $J_{1,1,1}$, and the mean duration of the regular jobs.

Table 3.5: Duration of big job and the mean duration of regular jobs with $\beta = -1$, and $\gamma = 1$, and different values of α

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.25	-1	1	11162	40001	1227	2000.4	3539.4
0.5	-1	1	11161	40000	1257	2000.2	3580.7
0.75	-1	1	11161	40000	1354	2000.2	3590.2
0.85	-1	1	11161	40000	1348	2000.2	3624.2
0.95	-1	1	11161	40000	1439	2000.2	3590.3
0.98	-1	1	11161	40000	1488	2000.2	3648.3
0.99	-1	1	11161	40000	1728	2000.2	3677.7
0.9999	-1	1	11161	40000	4685	2000.2	4040.5
1	-1	1	11161	40000	5409	2000.2	4008.8

Similar to when the job factor is not considered in assigning weight to jobs, with increase in α, $J_{1,1,1}$, gets low priority over time and more workers service the regular jobs decreasing the mean duration of the regular jobs, shown in Fig. 3.6.

Effect of β on the System

The system behavior is analyzed when the value of β, varies in the presence of the third parameter, γ. For a job, the expected number of remaining tasks decrease over the time slots, consequently the weight contributed by the job factor in assigning the priority of a job decreases. The exponential cumulative processing share, $E_{i,\ell}$, of user, i, in time slot, ℓ, increases if more number of tasks are serviced and decreases if a less number of tasks are serviced. We know (from equation 3.7) that the weight of a job contributed by β depends on $E_{i,\ell}$, (α moderates $E_{i,\ell}$) and the value of β. For the reasons explained above, for a job, more number of tasks are serviced in the initial few time slots and less number of remaining tasks are serviced when a small percent of the job is remaining. Small number of remaining tasks take a longer time because, weight contributed by job factor decreases over time and also it is observed that $E_{i,k}$, decreases predominantly i.e., $E_{i,k} < 1$, leading to a very small weight being assigned to the job. Thus, job completion is delayed towards the very end of completion of the job (increasing the duration of the job), shown in Fig. 3.7 and Fig. 3.8.

Fixing the values of α, γ, and varying values of β, we expect (from the algorithm equations) that an increase in the β, value should decrease the duration of completion of a job, but we observe otherwise from the data in table 3.6 i.e., the duration increases with an increase

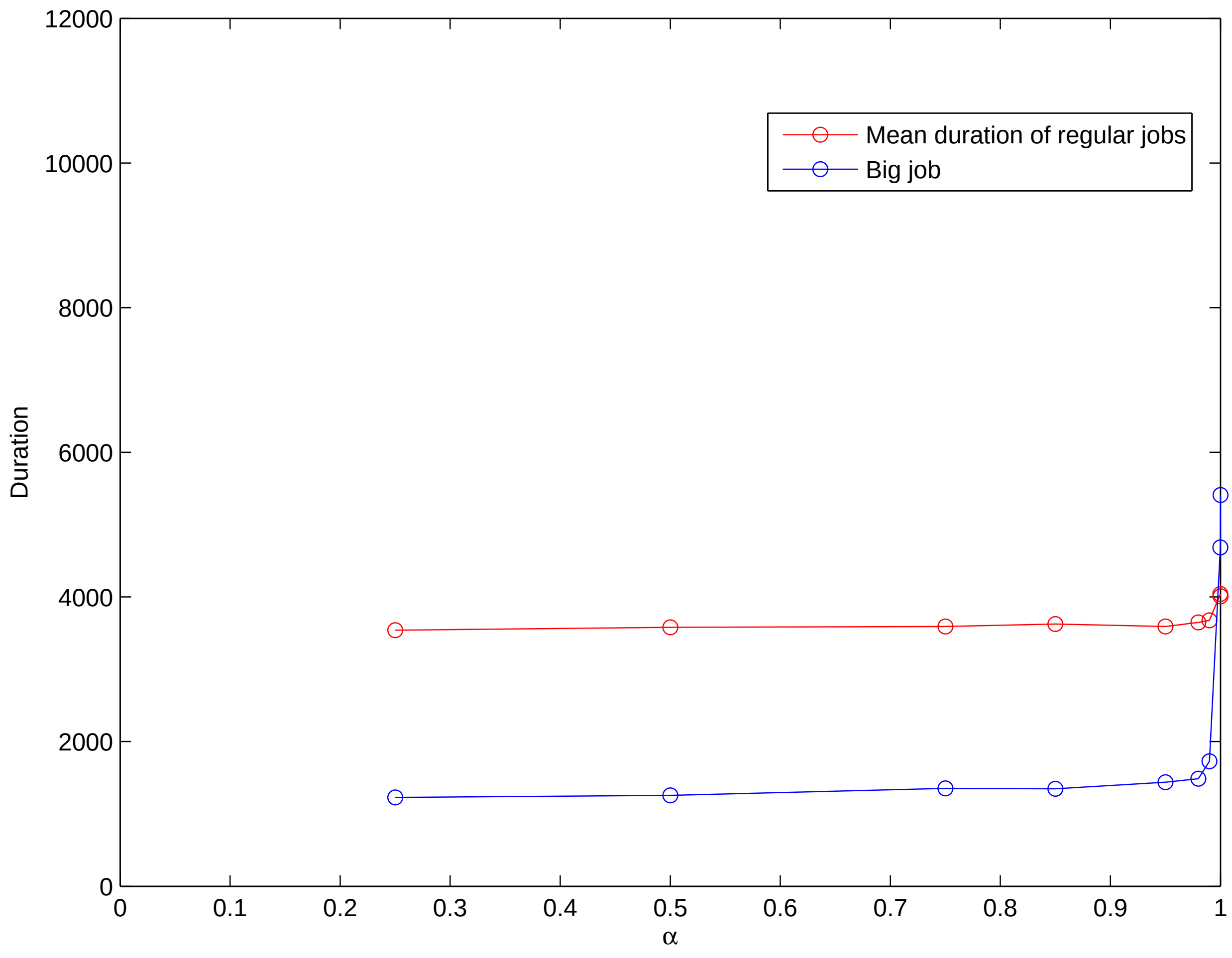


Figure 3.6: Duration of big job and the mean duration of regular jobs with $\beta = -1$, and $\gamma = 1$, and different values of α

Table 3.6: Duration of big job and the mean duration of regular jobs with $\alpha = 0.5$, $\gamma = 1$, and different values of β

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.5	-1	1	11161	40000	1257	2000.2	3580.7
0.5	-0.75	1	11161	40000	1281	2000.2	3707.3
0.5	-0.5	1	11161	40000	1348	2000.2	3747.2
0.5	0	1	11162	40000	2282	2000.4	4033.8

in the β value. The reason is explained as follows, analyzing the data for different values of β, we observed that more percent of job is completed in an initial small number of time slots as the weight contributed by the job factor (using γ) and $E_{i,\ell}$, is high. Towards the end of the job, a small percent of job takes long duration because the job gets less weight from job factor (because of the less number of remaining tasks) and it is observed that $E_{i,\ell} < 1$, so the weight contributed by β raised to the power of $E_{i,\ell}$, decreases with an increase in the value of β. The priority of the job is remarkably decreased because of the reasons explained above when a small amount of the job is remaining, thus increasing the duration of the job.

The system behavior is also observed by increasing the value of α, for $\gamma = 1$, and $\beta = -1, -0.75, -0.5, 0$ values. The values of α used to observe the system behavior are $\alpha = 0.5, 0.85, 0.95, 0.99$. For a fixed value of β, with an increase in α, job takes longer duration to complete, refer Tables 3.6 and 3.7. For α, very close to 1 i.e., $\alpha = 0.9999, 1$, the exponential cumulative processing share of a user, is never less than 1, $E_{i,\ell} \not< 1$. Thus, the duration to complete the job is observed to decrease with an increase in the value of β, unlike for other values of α, $(0 < \alpha < 1)$, the data can be referred in Table 3.8.

For the set of parameters considered to observe the system behavior, regular jobs are assigned a low priority as big job receives high priority because of the parameter, γ (initially, job factor is high for big job and then starts decreasing). The mean duration of regular jobs is observed to increase when compared to the mean duration if the parameter, γ, is not considered in assigning priorities to jobs. The mean duration increases with an increase in the value of β, but the increase is more predominant for the values of α, close to 1, shown in Fig. 3.9.

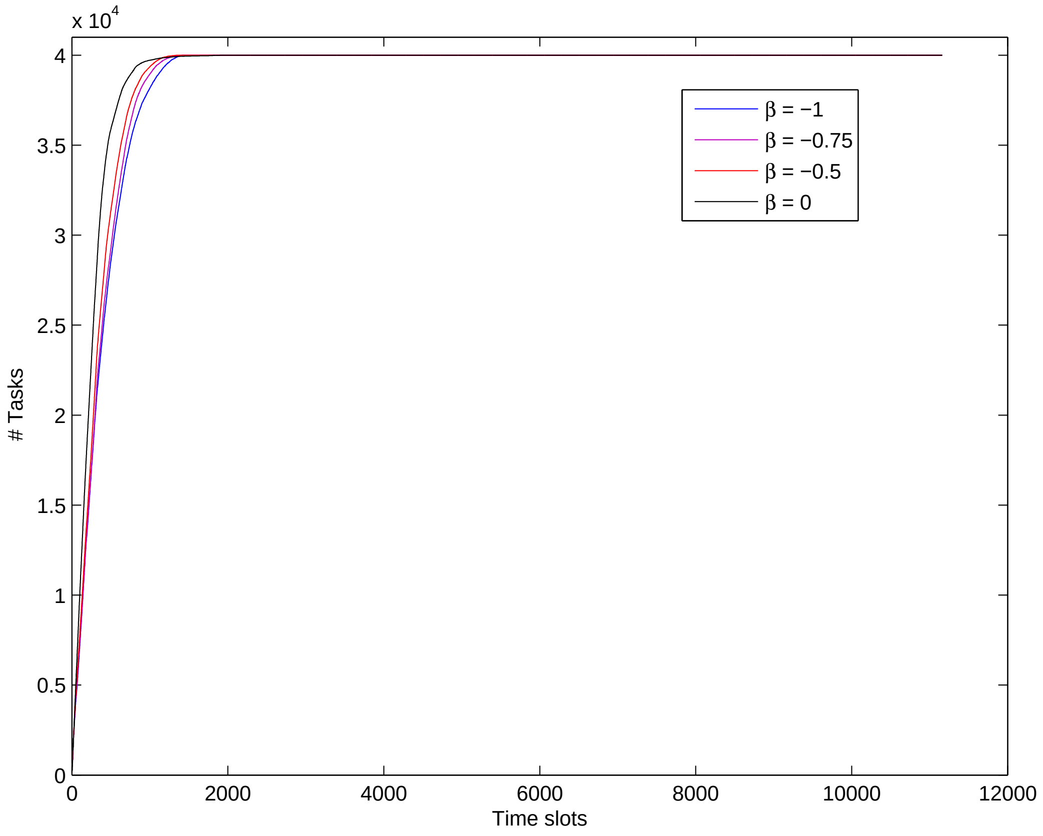


Figure 3.7: Cumulative tasks of big job for $\alpha = 0.95$, $\gamma = 1$, and different values of β

Table 3.7: Duration of big job and the mean duration of regular jobs with $\alpha = 0.95$, $\gamma = 1$, and different values of β

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.95	-1	1	11161	40000	1439	2000.2	3590.3
0.95	-0.75	1	11161	40000	1451	2000.2	3722.1
0.95	-0.5	1	11161	40000	1412	2000.2	3764.3
0.95	0	1	11162	40000	2613	2000.4	4033.5

Table 3.8: Duration of big job and the mean duration of regular jobs with $\alpha = 1$, $\gamma = 1$, and different values of β

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
1	-1	1	11161	40000	5409	2000.2	4008.8
1	-0.75	1	11161	40000	4135	2000.2	3971.6
1	-0.5	1	11161	40000	4567	2000.2	3983.7
1	0	1	11161	40000	2502	2000.2	4078.1

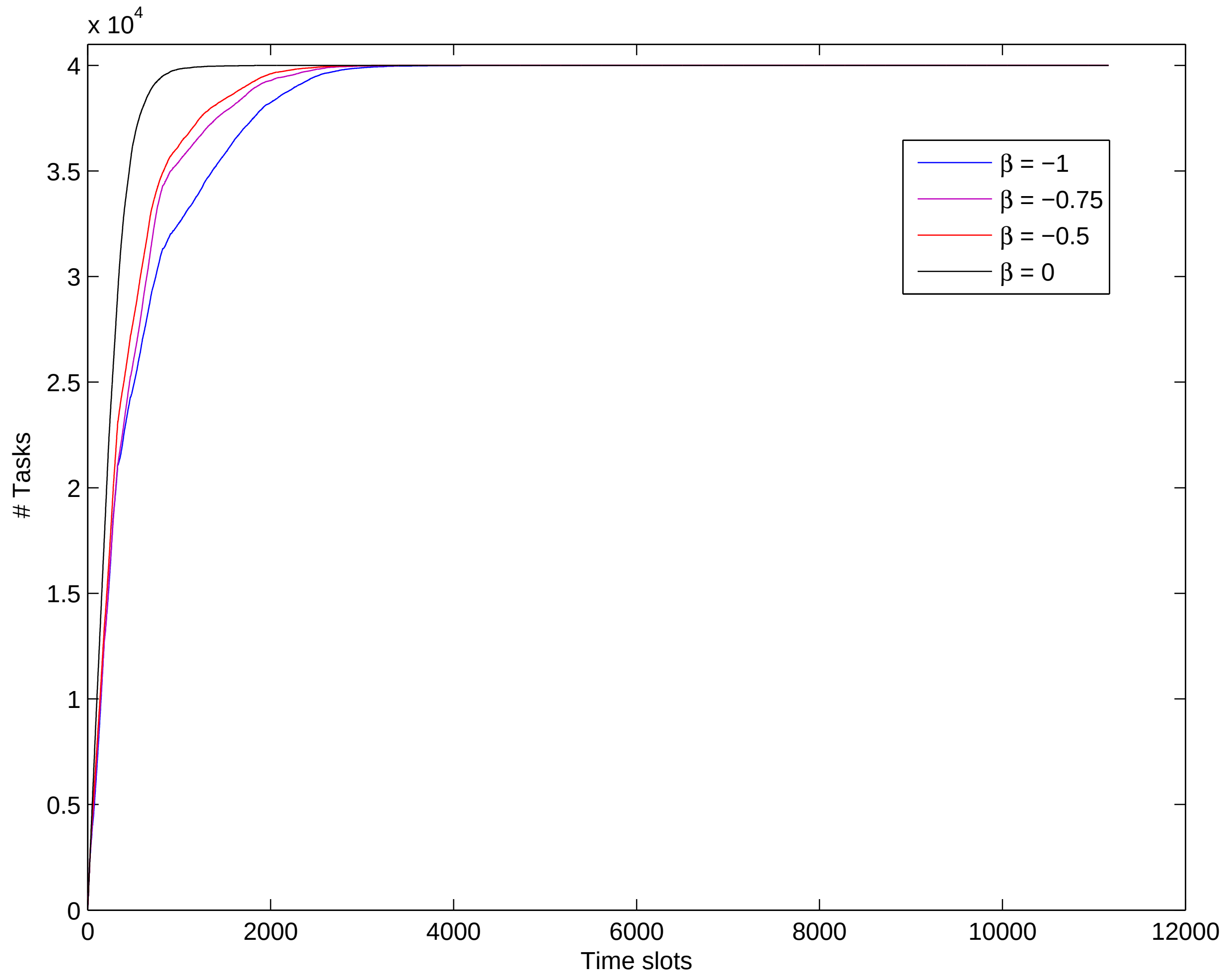


Figure 3.8: Cumulative tasks of big job for $\alpha = 1$, $\gamma = 1$, and different values of β

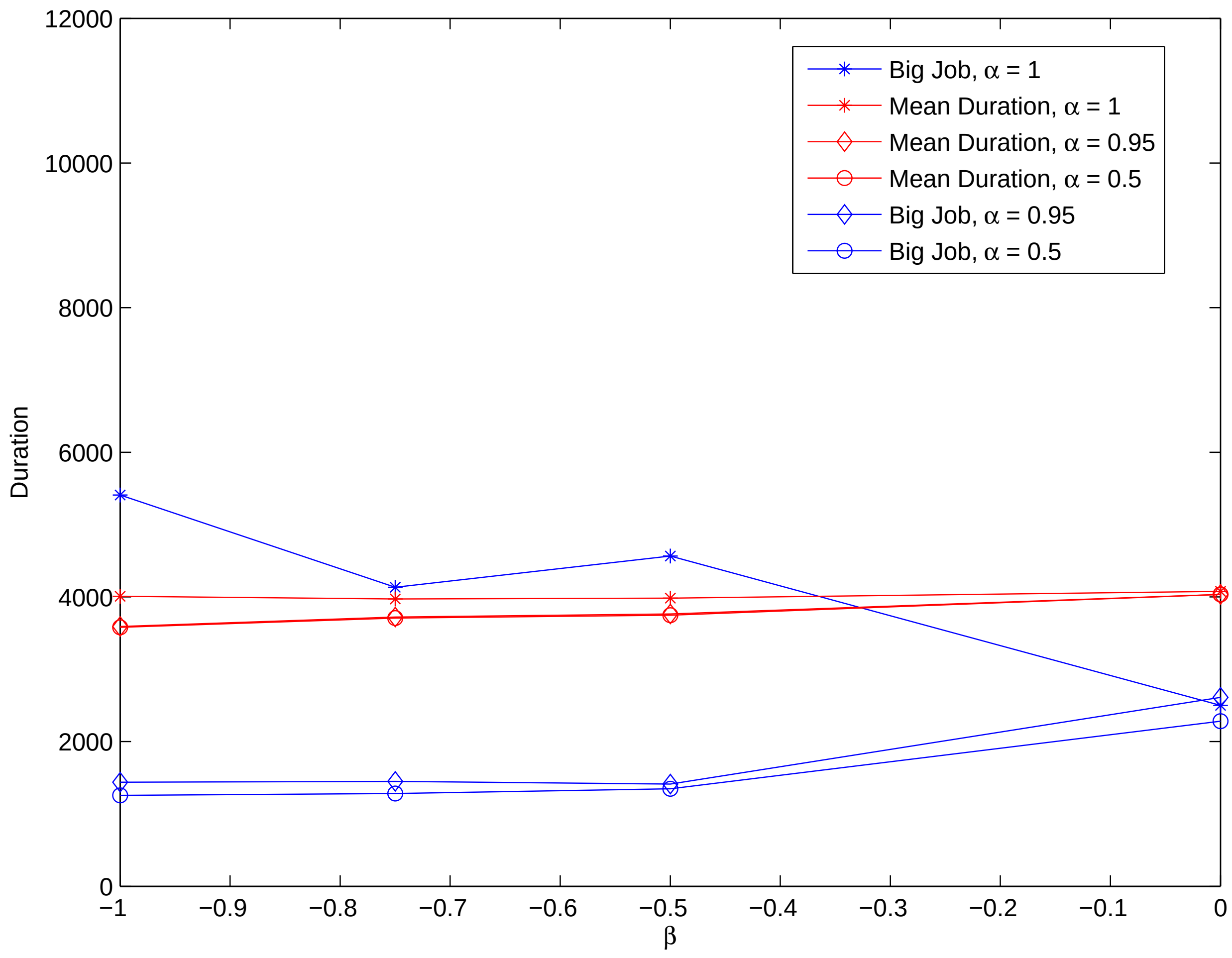


Figure 3.9: Duration of big job and the mean duration of regular jobs with $\alpha = 0.5, 0.95, 1$; $\gamma = 1$; and different values of β

Table 3.9: Duration of big job and the mean duration of regular jobs with $\alpha = 0.5$, $\beta = -1$, and different values ofγ

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.5	-1	-1	11173	40053	11173	2002.3	1350.4
0.5	-1	-0.5	11170	40001	7012	2001.9	1596.7
0.5	-1	0	11165	40000	3538	2000.9	1924.1
0.5	-1	0.5	11161	40000	2311	2000.2	2628.8
0.5	-1	1	11161	40000	1257	2000.2	3580.7

Effect of γ on the System

The value of the parameter, γ moderates the weight of a job through job factor. If $\gamma < 0$, a job with more number of remaining expected tasks is assigned a less weight and more weight is assigned for a job with a small number of expected tasks to complete (opposite to the behavior when $\gamma > 0$). To observe the effect of γ, the values of α, and β, are fixed and the value of γ is varied in the range $[-1, 1]$. $\gamma = 1$, results in providing preference to jobs with more number of remaining expected tasks, and $\gamma = -1$, gives least preference to jobs with more number of remaining tasks to complete. Observing the data provided in table 3.9, we can conclude that the duration to complete the big job decreases with increase in the value of γ.

The system behavior is also observed by increasing the value of α, with $\beta = -1$, and a set of values of γ (in the range $[-1, 1]$). For a fixed set of values of β, and γ, an increase in α, is observed to have an increase in the duration of the big job (e.g., the duration taken for $J1, 1, 1$, when $\alpha = 0.95$, is greater than the duration when $\alpha = 0.5$), the data is provided in Tables 3.9 - 3.13, and the behavior can be observed in Fig. 3.10. Values considered to analyze the system behavior are, $\alpha = 0.5, 0.85, 0.99, 0.9999, 1$. If the value of α, is really close to 1, i.e., for $\alpha = 0.9999, 1$; and $\gamma \leq 0$; we observe that the big job, $J_{1,1,1}$, takes long duration to complete refer data in Tables 3.12 and 3.13, the behavior can be observed in Fig. 3.11. The duration to complete a big job does not vary to a great extent when the value of α, is not close to 1, $\alpha \leq 0.99$.

Table 3.10: Duration of big job and the mean duration of regular jobs with $\alpha = 0.85$, $\beta = -1$, and different values of γ

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.85	-1	-1	11173	40091	11173	2002.2	1339.1
0.85	-1	-0.5	11169	40001	7243	2001.7	1590.3
0.85	-1	0	11164	40000	3673	2000.7	1919
0.85	-1	0.5	11161	40000	2353	2000.2	2633.7
0.85	-1	1	11161	40000	1348	2000.2	3624.2

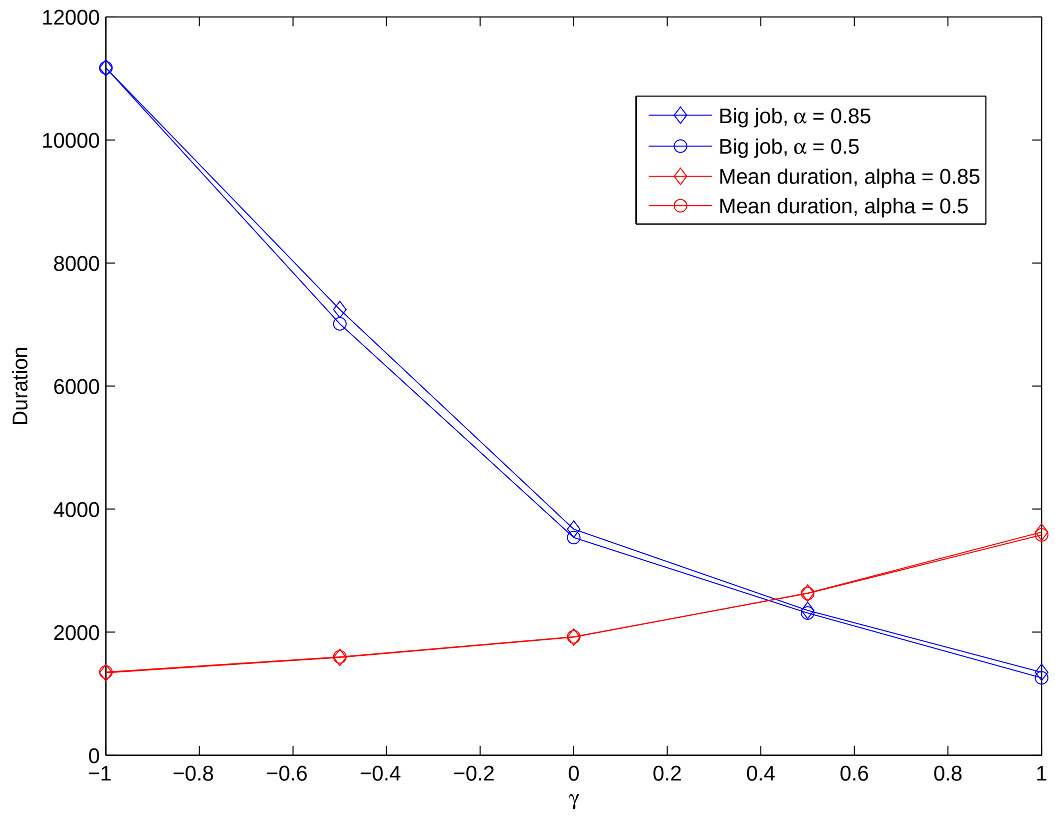


Figure 3.10: Duration of big job and the mean duration of the regular jobs with $\alpha = 0.85, 0.5$, $\beta = -1$, and different values of γ.

Table 3.11: Duration of big job and the mean duration of regular jobs with $\alpha = 0.99$, $\beta = -1$, and different values of γ

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.99	-1	-1	11170	40006	11170	2001.8	1307.4
0.99	-1	-0.5	11169	40000	8986	2001.7	1554.4
0.99	-1	0	11164	40000	4457	2000.7	1909.6
0.99	-1	0.5	11161	40000	2667	2000.2	2648.8
0.99	-1	1	11161	40000	1728	2000.2	3677.7

Table 3.12: Duration of big job and the mean duration of regular jobs with $\alpha = 0.9999$, $\beta = -1$, and different values of γ

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.9999	-1	-1	11171	40075	11171	2001.9	1203.9
0.9999	-1	-0.5	11168	40004	11168	2001.5	1392.9
0.9999	-1	0	11165	40002	9815	2000.9	1812
0.9999	-1	0.5	11161	40000	6293	2000.2	2695.8
0.9999	-1	1	11161	40000	4685	2000.2	4040.5

Table 3.13: Duration of big job and the mean duration of regular jobs with $\alpha = 1$, $\beta = -1$, and different values of γ

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
1	-1	-1	11171	40077	11171	2001.9	1198.4
1	-1	-0.5	11168	40079	11168	2001.3	1380.4
1	-1	0	11165	40002	10534	2000.9	1794.3
1	-1	0.5	11161	40000	7001	2000.2	2687.1
1	-1	1	11161	40000	5409	2000.2	4008.8

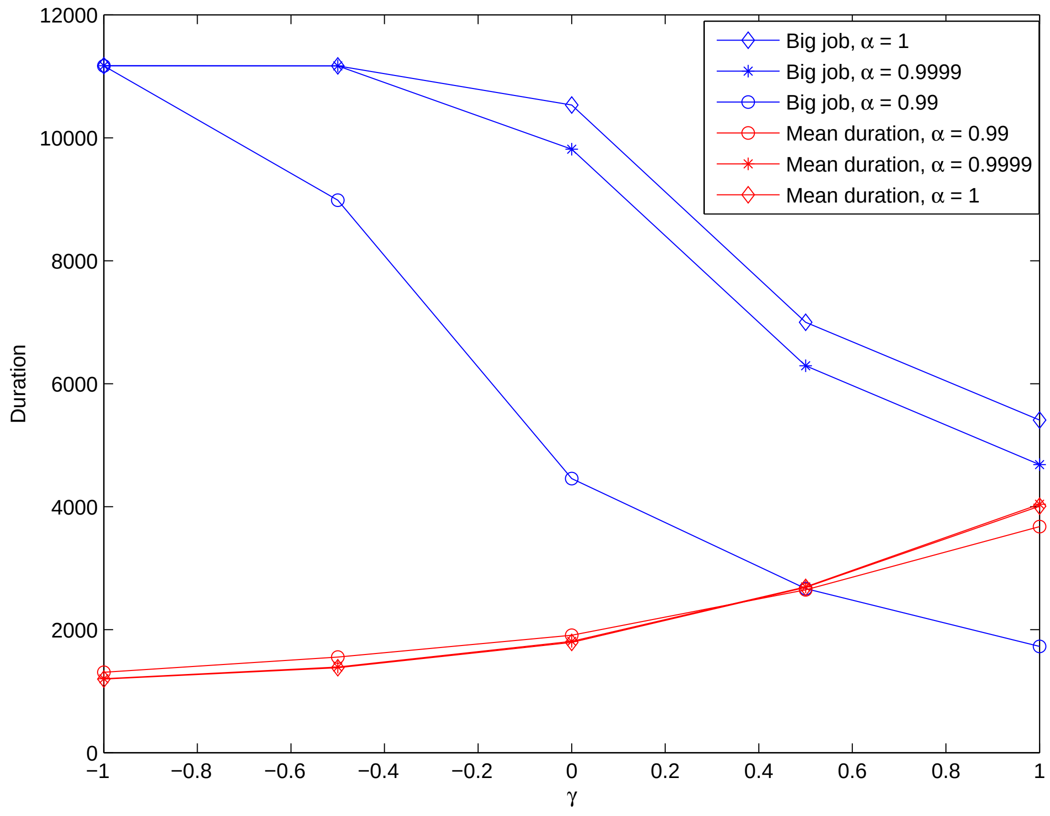


Figure 3.11: Duration of big job and the mean duration of regular jobs with $\alpha = 0.99, 0.9999, 1$; $\beta = -1$; and different values of γ.

Table 3.14: Duration of big job and the mean duration of regular jobs with $\beta = 0$, $\gamma = 0$, and $\alpha = 0.5, 0.85, 0.95, 0.99, 1$

α	β	γ	T	k of $J_{1,1,1}$	Γ_c of $J_{1,1,1}$	Mean # tasks	Mean duration
0.5	0	0	11167	40008	11167	2001.3	1788.2
0.85	0	0	11167	40043	11167	2001.2	1788.9
0.95	0	0	11168	40073	11168	2001.4	1788.1
0.99	0	0	11167	40040	11167	2001.2	1789
1	0	0	11167	40022	11167	2001.3	1791.8

3.5 Algorithm Special Cases

Using algorithms in section 3.3.1 and 3.4.1, by altering the parameters values we can schedule jobs in a number of possible ways. Two common and known algorithms are *equal job share* and *first-in first-out* FIFO. The parameter values to schedule jobs as equal job share and FIFO, duration of the big job, and the mean duration of the regular jobs for these values of the parameters are discussed in the following sub-sections.

3.5.1 Equal Job Share

Using the algorithm in section 3.3.1, we can schedule jobs as *equal job share* with $\beta = 0$. Using the algorithm in section 3.4.1, we can schedule jobs as equal job share with the parameter values $\beta = 0$, and $\gamma = 0$. In equal job share, each job in the queue gets an equal priority, i.e., in a time slot the probability that a worker chooses a particular job for execution is equal for all the jobs. The system is analyzed by fixing the values of β, and γ, and varying the value of α. The data in Table 3.14 shows that, for any $\alpha \leq 1$, big job, $J_{1,1,1}$, executes for the entire duration, T, used to analyze the system behavior for a single set of parameter values. The duration of $J_{1,1,1}$, is in the range $[11167, 11168]$ time slots, and the mean duration of the regular jobs is in the range $[1788, 1792]$, shown in Fig. 3.12. Observing the data we can conclude that *equal job share* is not preferred for a system where a combination of a small number of big jobs and a large number of small jobs are anticipated.

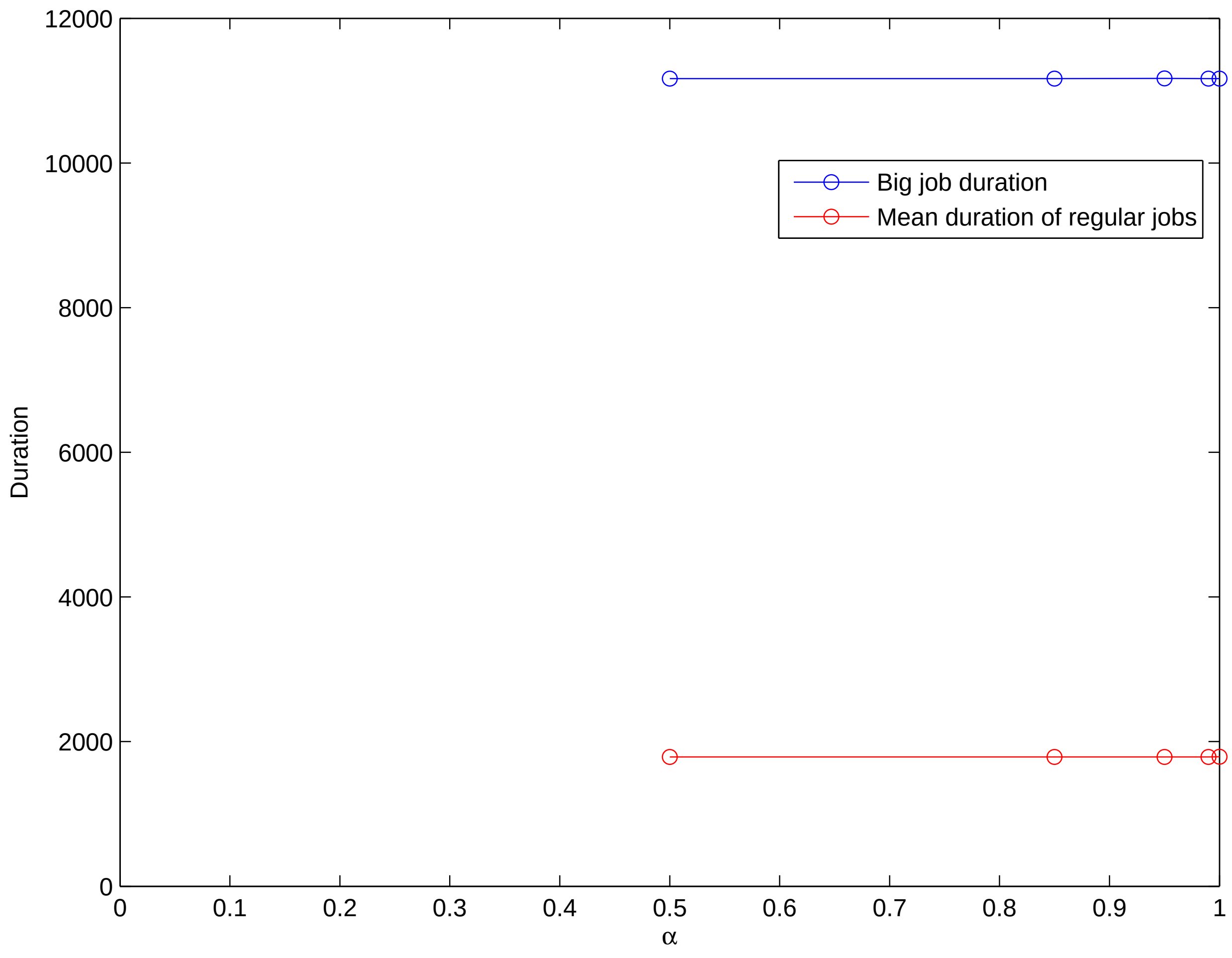


Figure 3.12: Duration of big job and the mean duration of regular jobs with $\beta = 0$, $\gamma = 0$, and $\alpha = 0.5, 0.85, 0.95, 0.99, 1$.

Table 3.15: FIFO with $\alpha = 0$, $\beta = 0$, and $\gamma = 0$

Case	α	β	γ	T	Mean # tasks	Mean duration
Case 1	0	0	0	11000	2013.9	963.692
Case 2	0	0	0	10927	2000	771.2226

3.5.2 First-In First-Out (FIFO)

Varying the parameter values, jobs in the queue can be executed in FIFO order. A variant of FIFO and FIFO can be implemented using the algorithm discussed in section 3.4.1. In case 1, all the jobs of a user with one or more incomplete jobs get equal priority with other users' jobs priority set to zero. Case 2 is absolute FIFO. The values of the parameters to realize FIFO (case 1 and case 2) are, $\alpha = 0$, $\beta = 0$, and $\gamma = 0$. In general the preferred values of α, are $0 < \alpha < 1$, however here we have $\alpha = 0$, this implies that, in case 1, we assign a priority to a job when the exponential cumulative processing power share of a user, i, in time slot, ℓ, $E_{i,\ell} \neq 0$. For the reason mentioned above, a user with an incomplete job gets equal priority among all of its jobs (refer step 1 of algorithm 3.3.1). Case 2 can be obtained by setting $E_{i,\ell} = 0$, for all the users in a time slot. The disadvantage in using FIFO is, a job has to wait until all the jobs submitted prior to it are completed (e.g., if a big job is submitted prior to small jobs they will be delayed until the big job is completed). System behavior with the parameter values for FIFO is observed with equal sized jobs for all the users. The mean duration of the jobs as observed in Table 3.15, is 963.692 time slots for the case 1, and 771.2226 time slots for case 2.

3.6 Conclusion

Using the parameters α, β, and γ the scheduling system behavior can be altered in accordance with the required scheduling policies. In a system with different types of users, e.g., type1 and type2, if users of type1 should be given preference over type2 users, then choosing a small value of α, and a big value of β for type1 users compared to type2 users gives preference to jobs of type1 users.

Using algorithm 3.3.1, with $0 < \alpha < 1$, and $\beta = 0$, jobs will be scheduled as *equal job share* and if $\beta = -1$, jobs are scheduled proportional to the exponential cumulative processing share of a user. $\beta < -1$, can be used if very less preference is required to be given to users who utilized more cumulative processing power. If it is required, per the scheduling policies that users who utilized more processing power be given preference over other users (a less common scenario in real time), then the following values can be set for the parameters in the algorithm, $0.5 \leq \alpha < 1$, and $\beta > 0$. Using γ, along with α, and β, parameters, jobs can be scheduled as *equal job share* with $0 < \alpha < 1$, and $\beta = \gamma = 0$. Jobs can be serviced in FIFO by not considering the exponential cumulative processing share of a user, with $\alpha = \beta = \gamma = 0$. Also, preference may be given to big jobs by setting $\gamma > 0$, though $\gamma = 1$, is preferred. When both big and small jobs exists in the jobs queue, preference can be given to small jobs by setting $\gamma < 0$.

Observing the data obtained from the various combinations of the parameters, we can say that using algorithm 3.3.1, the preferable values of the parameters are $0 < \alpha \leq 0.5$, and $\beta = -1$, as both the big job and the regular jobs do not take very long time to complete, i.e., neither of the jobs are starved or delayed indefinitely. Using the algorithm 3.4.1, the preferable values of the parameters are $0 < \alpha \leq 0.5$, $\beta = -1$, and $0 < \gamma \leq 0.5$.

Chapter 4

Case Study

4.1 Introduction

In an effort to make use of the scheduling algorithm described in chapter 3, we designed a system to run Monte Carlo simulations in parallel on the cluster using the insights obtained by examining the analytical model. A web application is developed to provide easy access of the cluster, using which any registered user can log in, upload, submit a job, and get back the results. Jobs are run on the cluster, so the users get back their results in a very short time compared to when the jobs are run on a PC. The technology discussed in this chapter has been used at WVU in several courses. In particular, a student project in the course EE 561 (Communication Theory) required students to design an optimal communication signal constellation, and a 200-core computing cluster was used to execute parallel simulations of the student's designs. A web interface was provided to give easy access to the cluster. A similar web interface was used in CPE 462 (Wireless Networking) to allow students to analyze cellular network designs.

4.2 Software Environment

Using the web application, a user can upload an input file (Matlab, .mat file), submit the job, monitor the status (Queued, Running or Done), progress (% of the job done), and download the output result files of the simulation. The complete system is deployed and runs on Linux environment. Web application is developed using *Google Web Toolkit* 2.2.0 in Java environment and is deployed in Apache Tomcat 5.5 server. All the work required for the implementation of the scheduling algorithm, processing of the job simulation and handling of the workers is written in Matlab. Unix shell scripts are used to start and stop the workers on the cluster and are handled from within the Matlab program. Data of all the registered users, details of the jobs (job name, upload time, status, progress, number of input and output files (input and output file names) are stored in the data base. MySQL $5.1.49-3$ Debian server is used as the data base server in the development of the application. The handling of the input and the output data of the jobs (files uploaded by the user and the result files) is done using the file system.

4.3 Analytical Model vs System Implementation

The actual system implementation differs in a few aspects from the analytical model. In the analytical model, workers have complete access to the global data structure. Workers check the random number they generated with the priority interval of each job, pick a job for execution, and update results when they complete execution of a task. In the actual system implementation, workers do not have access to the global table, `Jobs.mat` file. It is practically not a good choice to provide access to `Jobs.mat` to all the workers as it may lead to *race conditions*, where multiple workers manipulate data in `Jobs.mat` resulting in an inconsistent and invalid data. To handle this, task manager accesses the `Jobs.mat` and creates a number of tasks proportional to the job priority. In creating tasks, the buffer level of the `Tasks/input` directory is taken into consideration (i.e., maximum number of tasks that can be in the `Tasks/input` directory). Workers only have access to the `Tasks/input` directory, from where they can randomly pick a task and execute it.

The analytical model is synchronous, i.e., at the beginning of each time slot priorities of the jobs are calculated and tasks are serviced before the start of a new time slot. In the actual system implementation, job manager assigns priorities but tasks are created by the task manager when the number of tasks in the `Tasks/input` directory are below the buffer level i.e., actual system implementation is asynchronous. In the analytical model, we simplified simulation of a job by actually not generating trials with the assumption that i_{th} task of the job performs N_i trials and number of successes observed in each task is pN_i. In the actual system implementation, we do not make any assumptions of the successes observed in a task, we actually generate trials, observe and keep track of the successes.

4.4 System Operation

Using the web application, when a user submits a job, the uploaded input file is saved on the cluster file system by the Java program in `Jobs` directory (within the web interface directory) accessible to both Java and the Matlab programs. The directory structure of the file system is shown in Fig. 4.1. For each user, Java program creates a directory with `<username>`. Each time the user submits a job, a new directory using the `<jobid>` (unique key for each job) is created on the file system. Within this directory, an `input` directory (path: `Jobs/<username>/<jobid>/input`) is created and the input file is saved in this directory. For each new job submitted using the web application, Java program creates an empty file with filename of the format `<<jobid><username>>` and places it in the directory `Jobs/inputQueue`. Each file in `Jobs/inputQueue` directory indicates a new job is added to the system.

Every time period t_1, *Job manager* checks for new jobs in `Jobs/inputQueue` directory. For each file in the `Jobs/inputQueue` directory, job manager selects and parses the file (to get the JobId and the username, this is useful to get the information on the location of the input file), reads the input file, and then copies the input file(s) to matlab `Jobs/input` directory. All the new jobs are now in the `Jobs/input` directory. Initially, the status of jobs in the `Jobs/input` directory is *Queued*. All the jobs in the `Jobs/input` and `Jobs/running` directories (i.e., jobs with status queued and running respectively) are scheduled by the Job manager using the scheduling algorithm described in the section 3.4.1. The details of jobs i.e., the jobid, priority, status, expected number of tasks to complete, number of completed tasks, number of tasks in `Tasks` directory (i.e., in `Tasks/input` and `Tasks/running` directories, this is useful in calculating priorities of the jobs), and progress are saved into a Matlab file, `Jobs.mat`.

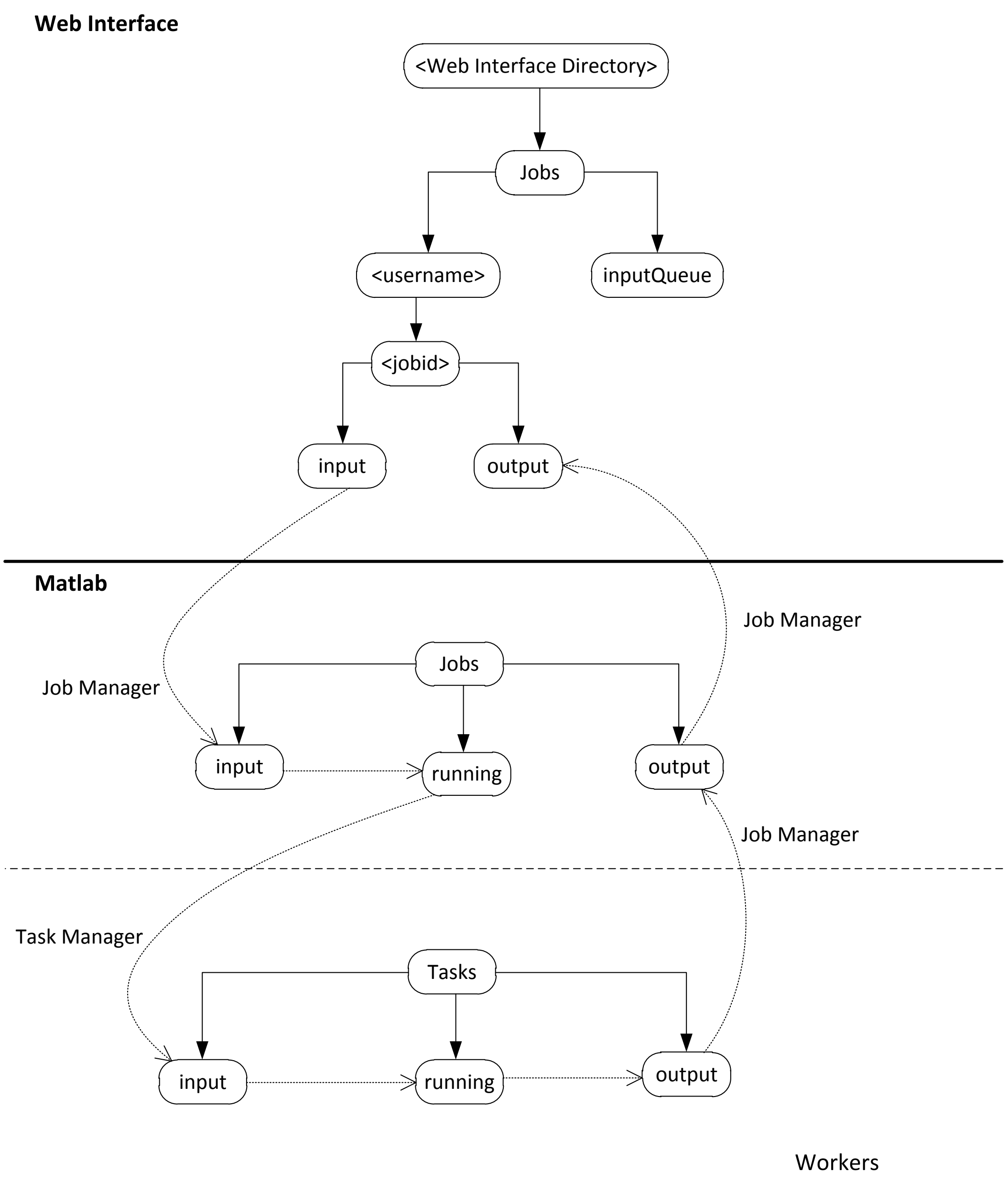


Figure 4.1: Directory structure and data flow in the file system

Each job with priority > 0 is moved into the `Jobs/running` directory and the status of the job is updated to *Running* in `Jobs.mat`. Using the priorities of the jobs in the `Jobs/running` directory (by reading file `Jobs.mat`), and the buffer level of the cluster task input directory `Tasks/input` the *Task manager* creates tasks proportional to the priority of job and places them in the `Tasks/input` directory. Each worker now checks the number of tasks available in the `Tasks/input` directory, generates a uniform random number between 1 and the maximum number of tasks. Using the number generated, worker chooses a task, moves the task from `Tasks/input` directory to `Tasks/running` directory, and services the task. A task is moved from `Tasks/input` directory to `Tasks/running` directory to avoid the conflict of executing a single task by multiple workers. Each worker executes a task for a period of $t_2 = 5$ minutes (taking care such that integer number of trials are performed i.e., a task is not stopped if it is in between a trial at the end of the time period, t_2). After a task is serviced, the task file in `Tasks/running` directory is deleted. The number of trials run and number of observed successes is written to an output file and is placed in the `Tasks/output` directory.

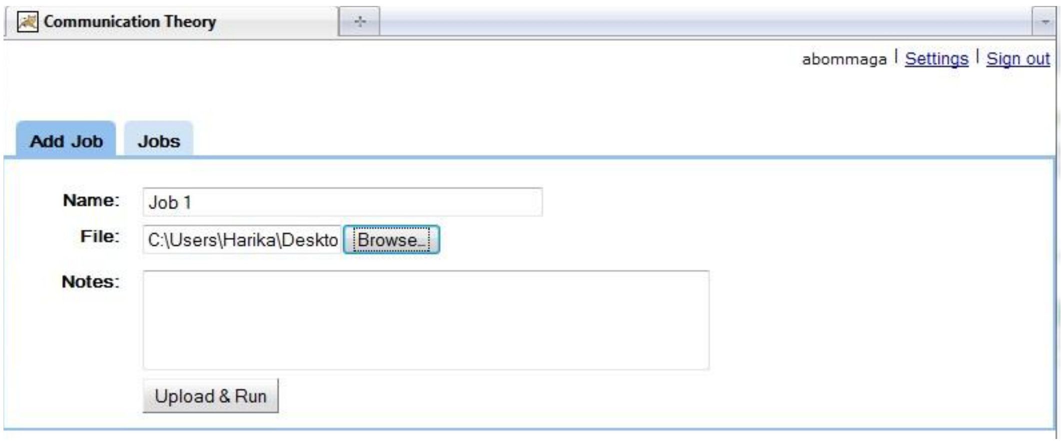
Every few minutes, job manager checks the `Tasks/output` directory, consolidates the results of each job and updates the results to `Jobs/output` directory. Each time the results are updated, job manager updates the `Jobs.mat`, and checks if any of the jobs are completed. For each job that is completed, its status is marked to *Done* in `Jobs.mat` file and the input file entry is deleted from the `Jobs/running` directory. We mark the status of the job to Done but do not delete job's entry as there might be some tasks running for the job. Entry in `Jobs.mat` with status Done acts as an indicator that the job's output file can be removed from `Jobs/output` directory when all the tasks (in `Tasks/running`) of the job are completed.

After updating `Jobs.mat` file, job manager triggers the task manager to check and delete tasks files of jobs with status Done from `Tasks/input` directory and return the job ids which have entries in `Tasks/running` directory (since we do not kill a task that is being executed by a worker). If the tasks in `Tasks/input` are deleted and if there are no tasks for the job in `Tasks/running` then job manger deletes that particular job entry from `Jobs.mat`, updates consolidated results to job output file in `Jobs/output` directory, (and subsequently to the web interface) and then deletes the output file from the `Jobs/output` directory (since further update of results to the output file is not required). If there are any tasks in `Tasks/running` directory, then job manager does not delete the entry in `Jobs.mat` (note that the job status is Done in `Jobs.mat`) and the output file in `Jobs/output` is not deleted. In further updates, when all the running tasks of the job (with status marked as Done) are executed, then the job's entry is removed from `Jobs.mat` and the output file is deleted from `Jobs/output` directory.

As it is not easy to read a Matlab (.mat) file using Java, the status and progress of a job is updated to the web application using a text file, `status.txt`. `status.txt` file contains the status, progress, and percent of the job completed. Job manager takes the result job files from the `Jobs/output` directory and update to the output directory of each job in the web application i.e., `Jobs/<username>/<jobid>/output` Task manager makes sure that a sufficient number of tasks are in the `Tasks/input` directory. Task manager checks the number of task files in the `Tasks/input` directory and then using the priorities of the jobs (from file `Jobs.mat`) creates new task files in `Tasks/input` directory if the number of task files is below the buffer level.

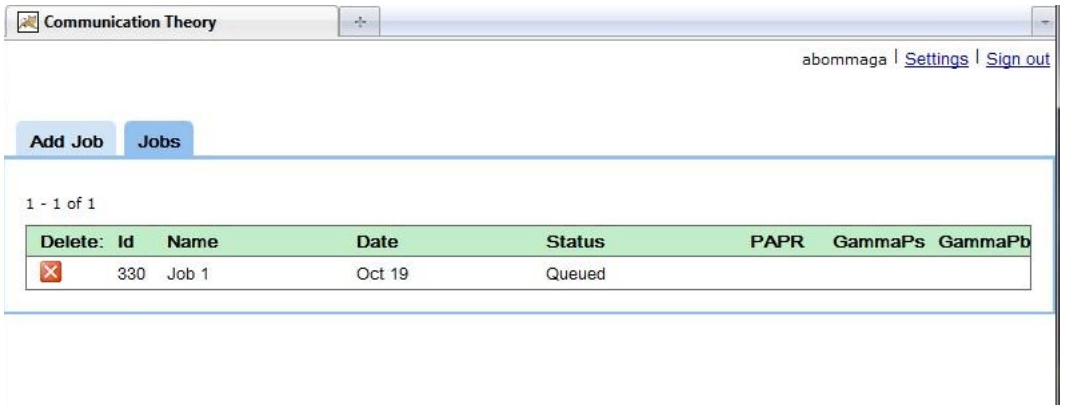
4.5 Web Application

A user friendly web application is developed for easy access of users to the cluster in order to submit and monitor jobs. A registered user can sign in to their account from virtually anywhere to manage their jobs. There are two types of users, regular users and administrators. A registered user can login to their account, add a new job using the *Add Job* tab Fig. 4.2, monitor jobs status in the list of jobs using *Jobs* tab Fig. 4.3. All the details of a job, such as the job upload time, input files, and output files can be viewed by clicking on a specific job Fig. 4.4 in the user account interface. Users can download input and output files of a job. Administrator user has more privileges than a regular user. An administrator can submit jobs as a regular user using *Add Job* tab, can create other users using *Add Users* tab, can reset and email passwords of the users, can assume identity of other users (by selecting a user from the list of users) using *Users* tab Fig.4.6. List of jobs



Name: Job 1
File: C:\Users\Harika\Deskto
Notes:

Figure 4.2: Add a new job



1 - 1 of 1

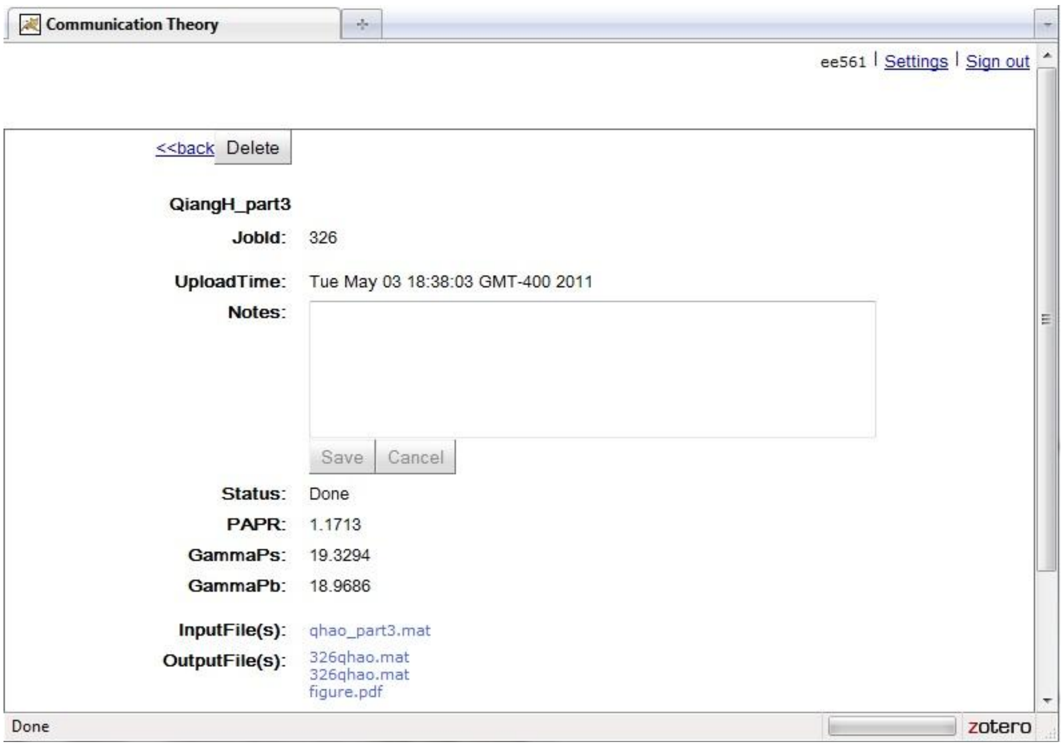
Delete:	Id	Name	Date	Status	PAPR	GammaPs	GammaPb
☒	330	Job 1	Oct 19	Queued			

Figure 4.3: List of jobs of a user

specific to an administrator can be monitored using the *My Jobs* tabs (this is similar to Jobs tab of regular users). Administrator has the privilege to monitor and edit all the jobs of the users, this is done using the *Jobs* tab Fig. 4.5 in the administrator interface. Assuming other users identity, an administrator can handle all the functions that a regular user can do such as submit, monitor jobs, and change settings.

4.6 Conclusion

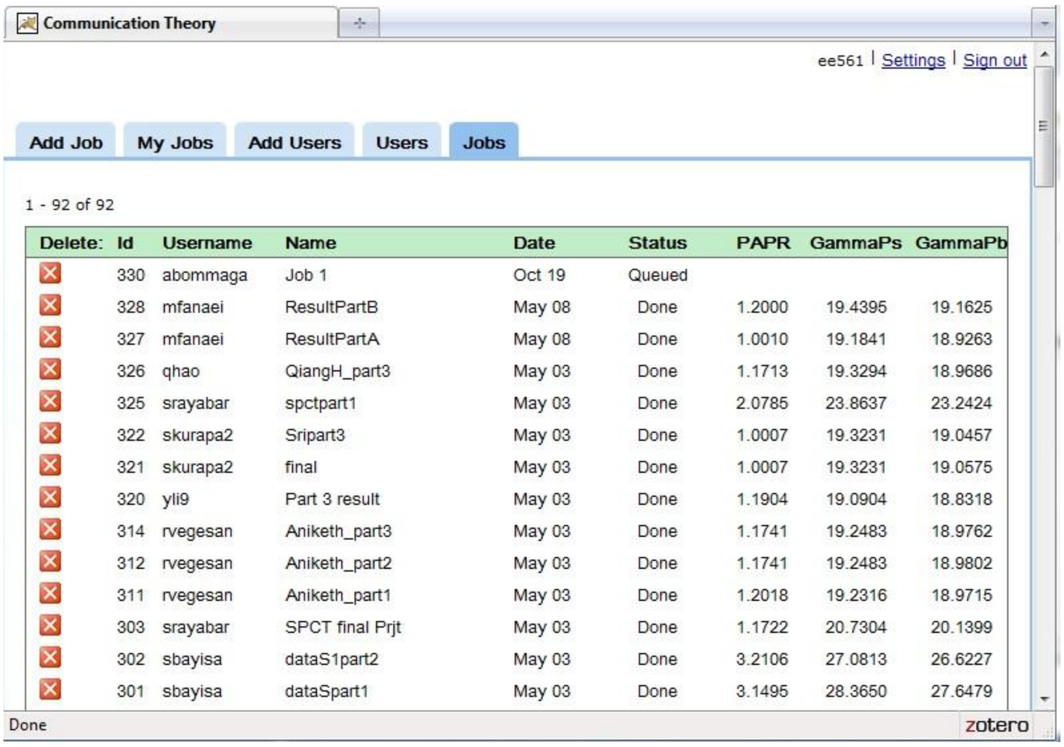
This chapter focused on the system implementation, to execute jobs in a distributed computing using the scheduling algorithm proposed in chapter 3. In this chapter, we discussed on how the analytical model differs from the actual system implementation. Later,



QiangH_part3

JobId:	326
UploadTime:	Tue May 03 18:38:03 GMT-400 2011
Notes:	
Status:	Done
PAPR:	1.1713
GammaPs:	19.3294
GammaPb:	18.9686
InputFile(s):	qhao_part3.mat
OutputFile(s):	326qhao.mat 326qhao.mat figure.pdf

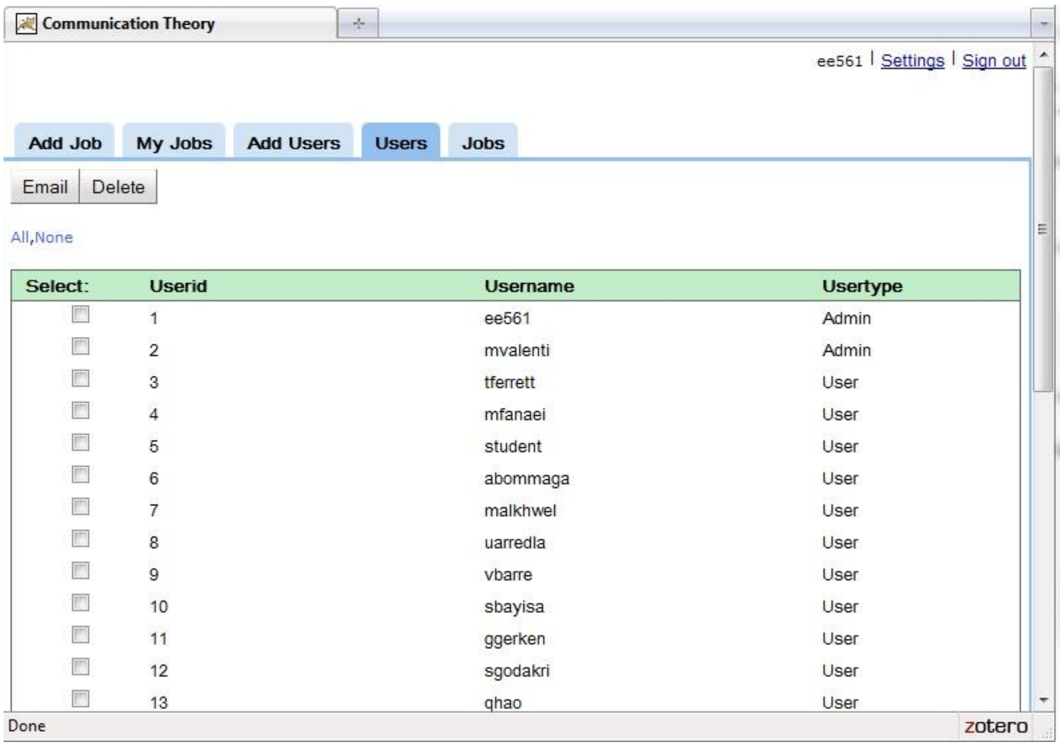
Figure 4.4: Details of a job



1 - 92 of 92

Delete:	Id	Username	Name	Date	Status	PAPR	GammaPs	GammaPb
☒	330	abommaga	Job 1	Oct 19	Queued			
☒	328	mfanaei	ResultPartB	May 08	Done	1.2000	19.4395	19.1625
☒	327	mfanaei	ResultPartA	May 08	Done	1.0010	19.1841	18.9263
☒	326	qhao	QiangH_part3	May 03	Done	1.1713	19.3294	18.9686
☒	325	srayabar	spctpart1	May 03	Done	2.0785	23.8637	23.2424
☒	322	skurapa2	Sripart3	May 03	Done	1.0007	19.3231	19.0457
☒	321	skurapa2	final	May 03	Done	1.0007	19.3231	19.0575
☒	320	yli9	Part 3 result	May 03	Done	1.1904	19.0904	18.8318
☒	314	rvegesan	Aniketh_part3	May 03	Done	1.1741	19.2483	18.9762
☒	312	rvegesan	Aniketh_part2	May 03	Done	1.1741	19.2483	18.9802
☒	311	rvegesan	Aniketh_part1	May 03	Done	1.2018	19.2316	18.9715
☒	303	srayabar	SPCT final Prjt	May 03	Done	1.1722	20.7304	20.1399
☒	302	sbayisa	dataS1part2	May 03	Done	3.2106	27.0813	26.6227
☒	301	sbayisa	dataSpart1	May 03	Done	3.1495	28.3650	27.6479

Figure 4.5: Administrative interface to list jobs of all users



Select:	Userid	Username	Usertype
☐	1	ee561	Admin
☐	2	mvalenti	Admin
☐	3	tferrett	User
☐	4	mfanaei	User
☐	5	student	User
☐	6	abommaga	User
☐	7	malkhwel	User
☐	8	uarredla	User
☐	9	vbarre	User
☐	10	sbayisa	User
☐	11	ggerken	User
☐	12	sgodakri	User
☐	13	qhao	User

Figure 4.6: Administrative interface to list all users

we presented the system implementation discussing the details on how the jobs submitted by the users are taken care by the job manager and the task manager, how the results of jobs can be viewed by the users, the directory structure, and the movement of the input and the output files of the jobs by the job manager and the task manager to handle the jobs. Finally, we discussed the features of the web interface, which is developed for easy use of the cluster. Submitting jobs to the system using the web application, users need not worry on the aspects to start the execution of a job and monitor its progress on the cluster. The chapter also includes few screen shots of the web interface to present the current implementation of the web interface.

Chapter 5

Conclusion

5.1 Conclusions and Summary

The work in this thesis combines the concepts of Monte Carlo simulation and distributed computing to efficiently execute compute-intensive problems. A scheduling algorithm is presented in order to execute Monte Carlo simulations in parallel in a distributed computing environment, more specifically the cluster computing. An analytical model is created under the assumption that the system behavior can be controlled by varying the values of the three parameters (α, β, and γ) of the algorithm. The proposed unified framework accommodates scheduling algorithms such as equal job share, FIFO, and proportionally fair scheduling. Variants of proportionally fair scheduling can be realized by varying the algorithm parameters.

The scheduling algorithm uses an exponentially-weighted window along with the forgetting factor, α, to keep track of resources used by the users. An exponential-weighted window has an advantage over the weighted window concept, as it only need to keep track of the exponential cumulative use of resources value in the past time slot (or time period) instead of the use of the computing resources by the users in each time slot over a number of time slots (defined by the window size) or time periods. From the data obtained by investigating the system behavior we can suggest that a good choice of parameter values to quickly execute jobs using algorithm 3.3.1 are $0 < \alpha \leq 0.5$ and $\beta = -1$, and $0 < \alpha \leq 0.5$, $\beta = -1$, and $0 < \gamma \leq 0.5$ using 3.4.1.

The analytical model is synchronous in nature, but in real time jobs are submitted and need to be scheduled asynchronously. As a case study, a sub-optimal actual system implementation is designed and presented to schedule and execute jobs on the cluster. A web application is provided to provide users with a medium to submit and monitor jobs on a cluster without worrying about the actual technology behind the handling of resources and scheduling of the jobs.

5.2 Future work

We now suggest possible improvements to the current system. In the current system, simulation performed for each input job is independent i.e., if two users submit jobs with same input parameters the system cannot identify that the two jobs are alike and performs the simulation twice, once for each of the jobs. To minimize such unnecessary use of the computing resources and to make the system smart, data mining and pattern recognition techniques can be used to identify and categorize jobs. Using these techniques, we can add capabilities to the system such that it checks if the input parameters of a new job exactly match with the input parameters of an existing job (which is already executed or is in the process of execution), and then provide the results without actually performing the simulation for the new job. The other case is when the input parameters of new job partially match the parameters of the existing jobs. In this case, the system can take the available SNR points data and simulate for the SNR points for which data is not readily available in the system. Only when the above two cases fail, simulation for the job can be performed, thus saving the computing resources.

Above we proposed an improvement to the system such that it utilize the existing data by matching the input parameters of new job, however, there is a possibility that the source code utilized to perform the simulation is updated, thus invalidating some of the factual data in the system. This can be handled by labeling jobs with the version of the source code used to simulate the jobs. This does not completely solve the problem since the sub-version repository is used to store the source code and each time any file is updated leads to a new version of the code. This is a hard problem and to an extent can be handled using the project and the version of the project source code, as the source code of all the projects might not be updated each time.

To handle high computational demand, jobs can be executed on the grid. However, it is a good process to start a simulation on the cluster and then transfer it to the grid depending on the necessity (if the resources required by the job is huge). Starting a simulation on a cluster gives an idea of the size of the job i.e., the approximate number of resources required and the amount of time it needs to complete. Directly submitting a job to the grid causes a delay in receiving the initial response of the job. The delay may be because of the network latency, since the server that manages the resources and scheduling of jobs, and the resources which actually execute tasks of the jobs are geographically distributed. Also, if we directly submit a job to grid it is possible that more than the required number of tasks might be created for the job (over work), and it is often difficult to kill the tasks once they are created. Starting a job on the cluster allows to maintain a consistent processing power credit system, since we have complete access and information of the cluster (configuration etc.,) unlike the resources on a grid. A form of consistent credit system is illustrated as follows; for instance we may define a credit to be 1000 trials/minute on the cluster. If a resource on the grid runs 500 trials/minute, then it implies only 0.5 credit of work is performed (because of its hardware configuration) as only 500 trials are run, and it doesn't mean one credit of work is performed as it ran for one minute.

The following improvements can be added to the web interface. The existing interface can handle only a single project. It would be better to have a single web interface that has the ability to handle multiple projects. Users can be provided with an option to subscribe to projects. This function can be added in the settings page, a new section, *program settings* can be added allowing users to list their preferred projects so that a newly submitted job is automatically run using the highest preferred project. Users may be provided the ability to categorize jobs into folders based on the project used to run the job. A further improvement to this idea can be, to add a function such that the system becomes smart enough to categorize jobs when the job is submitted. Also, users may be informed of their credit usage and they can be provided with the ability to request for more credits.

References

[1] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 2nd edition, 1992.

[2] G. Marsaglia, "A current view of Random Number Generators," in *Proc. Computer Science and Statistics: 16th Symposium on the Interface*. 1985, pp. 3–10, Elsevier Press.

[3] H. Bauke and S. Mertens, "Random numbers for large-scale distributed monte carlo simulations," *Physical review. E*, vol. 75, pp. 066701, Jun 2007.

[4] Leon-Garcia. A, *Probability, statistics, and random processes for electrical engineering*, Pearson Prentice Hall, 3rd edition, 2008.

[5] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[6] P. L'Ecuyer, *Handbook of Computational Statistics*, Springer, 2004.

[7] M. Saito, "An application of finite field: Design and implementation of 128-bit instruction-based fast pseudorandom number generator," M.S. thesis, Hiroshima University, Feb 2007.

[8] B. Mazzeo and M. Rice, "On Monte Carlo Simulation of Bit Error Rate," in *Proc. IEEE ICC*, 2011.

[9] R. Buyya, T. Cortes, and H. Jin, "Single System Image (SSI)," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 15, no. 2, pp. 124–135, 2001.

[10] C. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and F. Sommers, *Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers*, chapter 16, pp. 521—551, Springer, 2006.

[11] R. Lottaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot, "OpenMosix, OpenSSI and Kerrighed: A comparative study," in *Proceedings of CCGrid*, 2005, pp. 1016–1023.

[12] A. Barak and A. Shiloh, "The MOSIX Management System for Linux Clusters, Multi-Clusters, GPU Clusters and Clouds," A white paper, 2011.

[13] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J. Berthou, and I. Scherson, “Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems,” in *Proc. IEEE International Conference on Cluster Computing*, 2004, pp. 277–286.

[14] F. Ferstl, *Job and Resource Management Systems, High Performance Cluster Computing: Architectures and Systems*, vol. 1, Prentice Hall, NJ, USA, 1999.

[15] R. Buyya, “PARMON: A Portable and Scalable Monitoring System for Clusters,” *Software: Practice and Experience*, vol. 30, pp. 723–739, June 2000.

[16] W. Zhang, “Linux virtual server for scalable network services,” in *Proc. Linux Symposium*, July 2000.

[17] “Oracle grid engine: An overview,” A white paper, August 2010.

[18] R. Buyya and S. Venugopal, “A Gentle Introduction to Grid Computing and Technologies,” *CSI Communications*, vol. 29, no. 1, pp. 9–19, 2005.

[19] H. Stockinger, “Defining the grid: a snapshot on the current view,” *The Journal of Supercomputing*, vol. 42, no. 1, pp. 3–17, Mar. 2007.

[20] J. Joseph, M. Ernest, and C. Fellenstein, “Evolution of grid computing architecture and grid adoption models,” *IBM Systems Journal*, vol. 43, no. 4, pp. 624–645, 2004.

[21] I. Foster, “Globus toolkit version 4: Software for service-oriented systems,” in *Proceedings of IFIP International Conference on Network and Parallel Computing.* 2005, LNCS, pp. 2–13, Springer-Verlag.

[22] I. Foster and C. Kesselman, “Globus: A metacomputing infrastructure toolkit,” *International Journal of Supercomputer Applications*, vol. 11, pp. 115—128, 1996.

[23] D. Anderson, “Volunteer computing: The ultimate cloud,” *ACM Crossroads*, vol. 16, no. 3, pp. 7–10, Mar. 2010, ACM ID: 1734164.

[24] D. Anderson, “BOINC: A system for public-resource computing and storage,” in *Proc. IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 4–10.

[25] X. Dong, “Cloud computing: An emerging technology,” in *Proc. International Conference on Computer Design and Applications (ICCDA)*, June 2010, vol. 1, pp. 100–104.

[26] F.M. Aymerich, G. Fenu, and S. Surcis, “An approach to a cloud computing network,” in *Proc. Conf. on the Applications of Digital Information and Web Technologies (CADIWT)*, Aug. 2008, pp. 113–118.

[27] P. Mathur and N. Nishchal, “Cloud computing: New challenge to the entire computer industry,” in *Proc. Conf. on Parallel, Distributed, and Grid Computing (PDGC)*, Oct. 2010, pp. 223–228.

[28] S. Yadav and W. Zeng, "CLOUD: a computing infrastructure on demand," in *Proc. International Conference on Computer Engineering and Technology (ICCET)*, Apr. 2010, vol. 1, pp. V1–423–V1–426.

[29] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, "The Characteristics of Cloud Computing," in *Proc. International Conference on Parallel Processing Workshops (ICPPW)*, Sept. 2010, pp. 275–279.

[30] T. Dillon, W. Chen, and E. Chang, "Cloud Computing: Issues and Challenges," in *Proc. IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Apr. 2010, pp. 27–33.

[31] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *Security & Privacy, IEEE*, vol. 9, no. 2, pp. 50–57, Apr. 2011.

[32] S. Ramgovind, M.M Eloff, and E. Smith, "The management of security in Cloud computing," in *Proc. Information Security for South Asia (ISSA)*, Aug. 2010, pp. 1–7.

[33] I. Foster, Z. Yong, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-Degree compared," in *Proc. Grid Computing Environments Workshop (GCE)*, Nov. 2008, pp. 1–10.

[34] S. Zhang, X. Chen, Z. Zhang, and X. Huo, "The comparison between cloud computing and grid computing," in *Proc. International Conference on Computer Application and System Modeling (ICCASM)*, Oct. 2010, vol. 11, pp. V11–72–V11–75.

[35] B. Guptill, "Utility Computing: Business reality and advantages," Report prepared by Saugatuck Technology Inc., March 2005.

[36] C. Waldspurger and W. Weihl, "Stride scheduling: Deterministic Proportional-Share resource management," Tech. Rep., MIT Laboratory for Computer Science, 1995.

[37] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, Massachusetts Institute of Technology, 1995.

[38] U. Maheshwari, "Charge-Based Proportional Scheduling," Tech. Rep., MIT Laboratory for Computer Science, 1995.

[39] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, vol. 31, pp. 44—55, 1988.

[40] I. Stoica and H. Abdel-wahab, "A New Approach to Implement Proportional Share Resource Allocation," Tech. Rep., Old Dominion University Department of Computer Science, 1995.

John H. Hagen

Digitally signed by John H. Hagen
DN: cn=John H. Hagen, o=West Virginia University Libraries, ou=Acquisitions Department, email=John.Hagen@mail.wvu.edu, c=US
Date: 2011.12.05 15:07:22 -05'00'