AN ADAPTIVE AND EFFICIENT GRID SCHEDULER WITH DYNAMIC LOAD BALANCING

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

SHAH RUCHIRBHAI RAJENDRA

**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)**

JUNE, 2006

Candidate's Declaration

I hereby declare that work, which is being presented in this dissertation, entitled "**AN ADAPTIVE AND EFFICIENT GRID SCHEDULER WITH DYNAMIC LOAD BALANCING**", in partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** in **COMPUTER SCIENCE AND ENGINEERING**, submitted in the department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee is an authentic record of my own work carried out from July 2005 to June 2006, under the guidance and supervision of **Dr. Manoj Misra**, Associate Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee and **Dr. Bharadwaj Veeravalli**, Associate Professor, Department of Electrical and Computer Engineering, National University of Singapore, Singapore.

I have not submitted the matter embodied in this dissertation for the award of the any other degree.

R. R. Shah
Shah Ruchirbhai Rajendra

Certificate

This is to certify that the above statement made by candidate is correct to the best of our knowledge and belief.

Manoj Misra
Dr. Manoj Misra
Associate Professor
E & CE Department
IIT Roorkee, India.

[illegible]
Dr. Bharadwaj Veeravalli
Associate Professor
E & C Department
NUS, Singapore.

Acknowledgements

It is my privilege to express my profound sense of gratitude and thanks to my supervisors, **Dr. Manoj Misra** (IITR, Roorkee) and **Dr. Bharadwaj Veeravalli** (NUS, Singapore), for their valuable inspiration, guidance and suggestions throughout my dissertation work. I wish to thanks **Prof. Ravi Bhushan**, coordinator for exchange program for IITR and NUS and **Dr. Ankush Mittal** for their encouragement and faith on me.

I wish to thank technical staff of the Computer Networks and Distributed System (CNDS) lab, NUS for maintaining excellent working facilities during my stay at NUS. I am also thankful to staff member of software lab (E&CE Department, IITR) for their cooperation. I wish to express thanks to **Prof D. K. Mehra**, Head, Department of E&CE, for providing necessary facilities to complete my dissertation.

I can never thank enough my parents and my sister for being a constant source of love, encouragement and strength. They deserve special thanks for just being who they are and taking me to this stage in my life. Finally, I thank God for being kind to me and driving me through this journey.

Shah Ruchirbhai Rajendra
(Enrollment No: 044618)

Abstract

Grid computing holds the great promise to effectively share geographically distributed heterogeneous resources to solve large-scale complex scientific problems. Scheduling large scale computationally intensive applications in the Grid environment is challenging issue because target resources are heterogeneous and their load and availability may very with time. Further, as resources are geographically distributed in large-scale Grid environments and communication latency is significantly large due to Wide Area Network (WAN) through which resources are connected, job migration cost becomes an imperative factor for load balancing decision. Thus, performance of the Grid system depends greatly on the effective task scheduling and load balancing algorithm.

We address this problem by proposing load balancing algorithms, which are MELISA (Modified ELISA), R-MELISA (Receiver-initiated MELISA) and LBA (Load Balancing on Arrival). The algorithms differ in the way load balancing is carried out and is shown to be efficient in minimizing the response time on large and small scale Grid environments. MELISA and R-MELISA, applicable to large scale systems, is a modified version of ELISA[1] in which we consider job migration cost, resource heterogeneity and network heterogeneity when taking load balancing decision. LBA algorithm, applicable for small scale Grid systems, performs load balancing by estimating expected finish time of a job on buddy processors. One of the unique characteristics of our algorithms is system parameter estimation. Our algorithms estimate system parameters such as job arrival rate, CPU processing rate, load at processor and balance the load by migrating jobs to buddy processors taking into account all affecting factors for load balancing decision.

We quantify the performance of our algorithms using several influencing parameters such as, job size, data transfer rate, status exchange period, migration limit, and we discuss the implications of the performance and choice of our approaches. These load balancing algorithms are simulated in C++ language using Dev C++ software tool.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Overview

The Grid [2,3] is emerging as a wide-scale, distributed computing infrastructure that promises to support resource sharing and coordinated problem solving in dynamic, multi institutional Virtual Organizations [2]. The idea is similar to the former meta-computing [4] where the focus was limited to compute resources while Grid computing takes a broader approach. On one hand, Grid computing provides the user with access to locally unavailable resource types. On the other hand, there is the expectation that a large number of resources are available. A computational Grid is the cooperation of distributed computer systems where user jobs can be executed either on local or on remote computer systems. With the Grid becoming a viable high performance alternative to the traditional super-computing environment, various aspects of effective Grid resource utilization are gaining significance. With its multitude of resources, a proper scheduling and efficient load balancing across the Grid is required for improving the performance of the system.

In a Grid system, jobs are assumed to arrive independently at the processors. Due to an uneven and random arrival of jobs at different processors, and/or due to the difference in their processing capacities, quite often there is a build-up of jobs at some processors, while other processors remain *under-loaded.* To fully utilize the computational power offered by the Grid, jobs from overloaded processors have to be rescheduled to under-loaded processors to improve the overall processor utilization of the system. The problem of load redistribution in distributed system is recognized as load balancing. It is expected that this will result in a reduction of the *average job response time* (*ART*). Also the utilization of the Grid computers and the job throughput is likely to improve due to the load balancing effect between the participating systems.

Load balancing has been subject of intensive study for cluster computing and distributed computing. However, there are a wide variety of issues that need to be considered for heterogeneous Grid environment. For example, machine capacity in terms of processor speed differs because of resource heterogeneity. Also their usable capacities vary from moment to moment according to load imposed upon them. Network topology among resources is also not fixed due to dynamic nature of the Grid. Further, as resources are distributed in multiple domains in the Internet, underlying network connecting them is heterogeneous. The heterogeneity results in different capabilities for job processing and data access. As resources are geographically distributed and communication latency between them is very large due to Wide Area Network (WAN) through which they are connected, job migration cost becomes very important factor in load balancing decision. The focus of our work is to present load balancing algorithms adapted to heterogeneous Grid computing environment which considers all necessary factors such as load at processor, processor heterogeneity, network heterogeneity and job migration cost for load balancing. Our proposed algorithms are based on load estimation approach as carried out in the design of ELISA [1].

1.2 Motivation

In [5], it was pointed out that serious performance degradation will occur for slower networks such as for Grid environment if data migration cost is not considered when scheduling jobs. As resources are geographically distributed in Grid computing environment, communication latency between them is very large. So job migration cost becomes an imperative factor for load balancing. Further, to minimize total execution time of a job, load should be assigned to each processor proportional to its performance, taking into account processor heterogeneity in terms of its speed. Due to these reasons, in a heterogeneous Grid environment, performance of the system is largely affected by resource heterogeneity, considerable communication delay, dynamic changing environment of the Grid etc. In this work, we have proposed dynamic, de-centralized and distributed load balancing algorithms which are applicable to heterogeneous Grid environments and consider all necessary affecting factors for load balancing. One of the important characteristics of our algorithms is to estimate system parameters such as job arrival rate, processing rate and to perform proactive job migration. These load balancing algorithms are shown to be applicable to balance the load depending upon the size of the underlying Grid infrastructure. Thus, for smaller size Grids, one of our algorithms, LBA (Load Balancing on Arrival) is shown to be effective whereas for large-scale Grid systems, MELISA (Modified ELISA) and R-MELISA (Receiver-initiated MELISA) are shown to have better control in balancing the loads. Our algorithms are highly adaptive in nature in the sense that number of job migration performed for execution of N jobs triggers by available network bandwidth, processor heterogeneity and current load at processor.

1.3 Statement of Problem

The aim of our dissertation work is to design dynamic, de-centralized and distributed load balancing algorithms which improve performance of the system by minimizing the Average Response Time (*ART*) of the jobs for heterogeneous Grid environments. These load balancing algorithms should be highly adaptive in nature and should take into account all necessary factors which affect load balancing decision. These factors include resource heterogeneity in terms of processor speed, network heterogeneity in terms of available network bandwidth, job migration cost and arbitrary network topology due to dynamic behavior of the Grid.

1.4 Organization of The Report

This dissertation report is organized as follows: Chapter 1 introduces objectives of the dissertation. In Chapter 2, we give overview of Grid computing and Grid scheduling system. Chapter 3 discusses necessity of load balancing in heterogeneous Grid environment and discusses required steps to perform dynamic load balancing. In Chapter 4, we present Grid system model considered in this work and also introduces performance metrics which are used to compare results of our algorithms with reference algorithms. Chapter 5 gives design details of our proposed load balancing algorithms. It also provides details for reference algorithms. Performance of our algorithms and discussion of results are presented in Chapter 6. Finally, Chapter 7 presents conclusion and future work.

Chapter 2 Background

2.1 What is Grid Computing?

Heraclites, an ancient Greek orator, is credited to saying that the only thing constant is change. Notably, throughout history a small set of these changes have lead to some large scale evolutions. For example, the invention of the steam engine in 1712 and later the telegraph in 1836 are considered to be key innovations that drove the industrial revolutions. More recently in 1948, the invention of the transistor opened the information age we live in today, replacing inefficient vacuum tube technologies and forever changing the way electronics impact our lives. Similarly, it is predicted that *Grid Computing* will have the same, if not greater, effect on this world. It will revolutionize every aspect of Information Technology and its vast networked technologies like bioinformatics, finance, physics, chemistry, and business, to just name a few. So, what exactly is *Grid Computing*?

The term Grid comes from an analogy to a power Grid. When you plug an appliance into a receptacle, you expect that you will be supplied with electricity of correct voltage, whereas you need not care where the power comes from and how it is generated. In mid-1990s, inspired by the pervasiveness, reliability, and ease of use of electricity, Foster et al [3] began exploring the design and development of an analogous computational power Grid for wide-area parallel and distributed computing. According to definition given by Foster [2,3], *Grid* is defined as an infrastructure that allows for flexible, secure, coordinated resource sharing among dynamic collections of individuals, resources and organizations. Grid computing strives to aggregate diverse, heterogeneous, geographically distributed and multiple-domain-spanning resources to provide a platform for transparent, secure, coordinated, and high-performance resource-sharing and problem solving. The resources that Grid computing is attempting to integrate are various. They include supercomputers, workstations, databases, storages, networks, software, special instruments, advanced display devices, and even people. Knowing the distributed nature of *Grid Computing* and its focus on sharing resources among organizations, notion of a Virtual Organization (VO) [2] is introduced. A set of individuals and/or institutions defined by such sharing rules form what we call a Virtual Organization (VO) (Refer Figure 2.1).

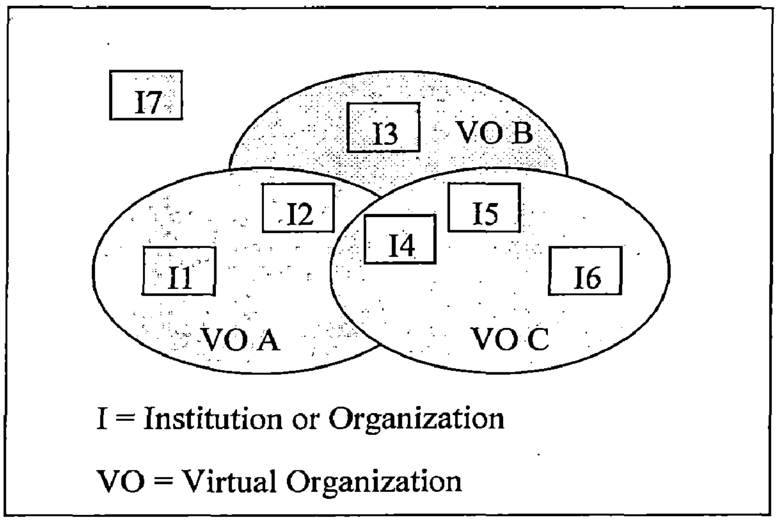


Figure 2.1 Virtual Organization

2.2 Grid Components : A High-Level Perspective

Now let us see at a high level the primary components of a Grid environment. Depending on the Grid design and its expected use, some of these components may or may not be required, and in some cases they may be combined to form a hybrid component. (Refer Figure 2.2)

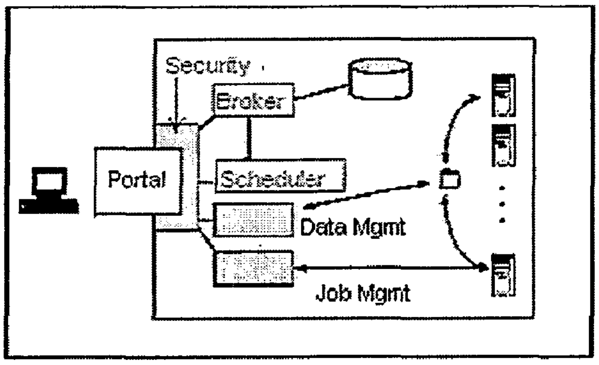


Figure 2.2 Grid Components

➢ **Portal - User interface**

Just as a consumer sees the power Grid as a receptacle in the wall, likewise a Grid user should not see all of the complexities of the computing Grid. Though the user interface could come in many forms and be application specific, for the purposes of discussion let us think of it as a portal. Most users today understand the concept of a Web portal, where their browser provides a single interface to access a wide variety of information sources. A Grid portal provides the interface for a user to launch applications that will utilize the resources and services provided by the Grid. From this perspective the user sees the Grid as a virtual computing resource just as the consumer of power sees the receptacle as an interface to a virtual generator.

- **Security**

  A major requirement for Grid computing is security. At the base of any Grid environment, there must be mechanisms to provide security including authentication, authorization, data encryption, and so on. For example, Grid Security Infrastructure (GSI) component of the Globus Toolkit [6, 7] provides robust security mechanisms. It also provides a single sign-on mechanism, so once a user is authenticated, a proxy certificate is created and used when performing actions within the Grid. The portal would then be responsible for signing into the Grid, either using the user's credentials, or using a generic set of credentials for all authorized users of the portal.

- **Broker**

  Once authenticated, the user will be launching an application. Based on the application, and possibly on other parameters provided by the user, the next step is to identify the available and appropriate resources to utilize within the Grid. This task could be carried out by a broker function. This service provides information about the available resources within the Grid and their status.

- **Scheduler**

  Once the resources have been identified, the next logical step is to schedule the individual jobs to run on them. If a set of standalone jobs are to be executed with no interdependencies, then a specialized scheduler may not be required. However, if it is desired to reserve a specific resource or to ensure that different jobs within the application run concurrently (for instance, if they require inter-process communication), then a job scheduler should be used to coordinate the execution of the jobs. It should also be noted that there could be different levels of schedulers within a Grid environment. For instance, a cluster could be represented as a single resource. The cluster may have its own scheduler to help manage the nodes it contains. A higher level scheduler (sometimes called a meta scheduler) might be used to schedule work to be done on a cluster, while the cluster's scheduler would handle the actual scheduling of work on the cluster's individual nodes.

- **Data management**

  If any data (including application modules) must be moved or made accessible to the nodes where an application's jobs will execute, then there needs to be a secure and reliable method for moving files and data to various nodes within the Grid.

- **Job and resource management**

  This is a core set of services that help perform actual work in a Grid environment. The Grid Resource Allocation Manager (GRAM) of Globus [6, 7] provides the services to actually launch a job on a particular resource, check on its status, and retrieve its results when it is complete.

2.3 Grid System Taxonomy

Grid computing can be used in a variety of ways to address various kinds of application requirements. According to the distinct targeted application realms, Grid systems can be classified into two categories. But there are actually no hard boundaries between these Grid categories. Real Grids may be a combination of one or more of these types. The two categories of the Grid systems are described below:

- **Computational Grid**

  A computational Grid is a system that aims at achieving higher aggregate computational power than any single constituent machine. According to how the computing power is utilized, computational Grids can be further subdivided into *distributed supercomputing* and *high throughput* categories. A distributed supercomputing Grid exploits the parallel execution of applications over multiple machines simultaneously to reduce the execution time. A high throughput Grid aims to increase the completion rate of a stream of jobs through utilizing available idle computing cycles as many as possible.

- **Data Grid**

  A data Grid is responsible for housing and providing access to data across multiple organizations. Users are not concerned with where this data is located as long as they have access to the data. For example, there can be two universities doing life science research, each with unique data. A data Grid would allow them to share their data, manage the data, and manage security issues. European DataGrid Project [8] is one example of Data Grid.

On the other hand, Grids can be built in all sizes, ranging from just a few machines in a department to groups of machines organized as hierarchy spanning the world. Grids can be classified into three categories according to the topology of the Grid. The relationship between the three Grid topologies is illustrated in Figure 2.3.

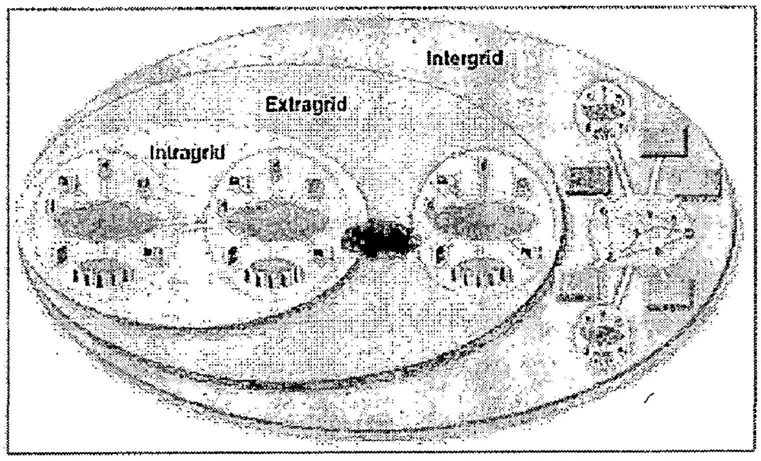


Figure 2.3 IntraGrid, ExtraGrid and InterGrid

- **IntraGrid**

  A typical IntraGrid topology exist between a single organization. The single organization could be made up of a number of computers that share a common security domain, which are connected by a private high-speed local network. The primary characteristics of an IntraGrid are a single administrative domain and bandwidth guarantee on the private network. Within an IntraGrid, it is easier to design the scheduling system, since an IntraGrid provides a static set of computing resources and communication capability between machines.

- **ExtraGrid**

  An ExtraGrid couple two or more IntraGrids. The ExtraGrid typically involves more than one administrative domains and the level of management complexity increases. The primary characteristics of an ExtraGrid are multiple domains and remote/WAN connectivity. Within an ExtraGrid, the resources become more dynamic. A business would benefit from an ExtraGrid if there was a business initiative integrating with external trusted business partners.

- **InterGrid**

  An InterGrid has an analogy with the Internet. It is most complicated form of Grid topology. The primary characteristics of an InterGrid are dispersed security, multiple domains and WAN connectivity. A business may deem an InterGrid necessary if there is a need for a collaborative computing community, or simplified end to end processes with the organizations that will use the InterGrid.

2.4 Advantages of Grid Computing

Grid computing is a promising paradigm with the following potential advantages:

- **Exploiting underutilized resources**

  Studies [3] have shown that most low-end machines (PCs and workstations) are often remain idle with utilization as low as 20%. And even for servers, only 50% of their capacity is utilized. Grid computing provides a platform to exploit these underutilized resources and thus has the possibility of increasing the efficiency of resource usage. A simple case is that we can run a local job on a remote machine elsewhere in the Grid if the local machine is busy.

- **Resource balancing**

  After joining a Grid, users will have a dramatically larger pool of resources available for their applications. When the local system is busy with heavy load, part of the workload can be scheduled to other resources in the Grid. Thus the function of resource balancing is achieved. This feature proves to be invaluable for handling occasional peak loads on a single system.

- **Distributed supercomputing capability**

  The parallel execution of parallel applications is one of the most attractive features of computational Grids. A wide spectrum of applications is parallel in nature and these applications are intended to be computation-intensive. In Grid systems, there are a large number of computational resources available for one parallel application, such that different jobs within the application can be executed simultaneously on a suite of Grid resources.

- **Virtual organizations for collaboration**

  Another important contribution of Grid computing is to enable the collaboration among wider-area members. Grid computing provides the infrastructure to integrate heterogeneous systems to form a virtual organization. Under the virtual organization, sharing is not limited to computational resources, but also includes various resources, such as storages, software, databases, special equipments, and so on. Furthermore, the sharing is more direct through using uniform interfaces. Although sharing in a virtual organization is quite direct, security and local policy are guaranteed. Local resources are protected securely against those who are not authorized to access.

- **Reliability**

  High-end conventional computing systems use expensive hardware to increase reliability. In the future, Grid computing provides a complementary approach to achieving high-reliability nevertheless with little additional investment. The resources in a Grid can be relatively inexpensive, autonomous and geographically dispersed. Thus, even if some of the resources within a Grid encounter a severe disaster, the other parts of the Grid are unlikely to be affected and remain working well.

With multitude of resources available for the Grid, a proper scheduling and efficient load balancing across the Grid is required for improving the performance of the system. In the following section, we will describe how to schedule a job in the Grid system and the challenges which are faced by a Grid scheduler in general.

2.5 Grid Scheduling System

A scheduler is the interface between the consumers and the underlying resources and acts as the mediate resource manager, as illustrated in Figure 2.4. Scheduling is a core function of resource management systems.

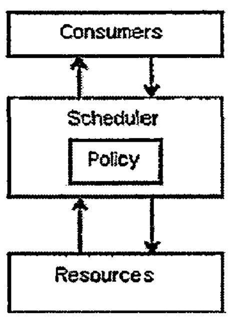


Figure 2.4 Scheduler Function

In a distributed environment, on one hand, there is a suite of computational resources interconnected by networks; on the other hand, there is a group of users who will submit applications for execution on the suite of resources. The scheduling system of such a distributed computing environment is responsible for managing the suite of resources and dealing with the set of applications. In face of a set of applications waiting for execution, the scheduling system should be able to allocate appropriate resources to applications, attempting to achieve some performance goals.

2.5.1 Definition of Grid Scheduling System

In traditional parallel computing environments, the scheduling system is made much simpler due to the uniform characteristics of both the target applications and the underlying resources. However, a computational Grid has more diverse resources as well as more diverse applications. According to GGF (Globus Grid Forum)'s [9] Grid scheduling dictionary, the Grid scheduler is responsible for:

(1) Discovering available resources for an application

(2) Selecting the appropriate system(s), and

(3) Submitting the application.

In brief, **Grid scheduling system** is a software framework with which the scheduler collects resoùrce state information, selects appropriate resources, predicts the potential performance for each candidate schedule, and determines the best schedule for the applications to be executed on a Grid system subject to some performance goals.

In principle, scheduling in Grids means two things: ordering and mapping. When there are more than one applications waiting for execution, *ordering* is performed in order to determine by which order the pending applications are arranged. *Ordering* is necessary if applications with priority or deadline are involved. *Mapping* is the process of selecting a set of appropriate resources and allocating the set of resources to the applications. For each mapping, the performance potential is estimated in order to decide the best schedule.

In general, a scheduling system of Grid computing environments aims at delivering better performance. Desirable performance goals of Grid scheduling includes: minimize average response time, maximizing system throughput, maximizing resource utilization, minimizing the execution time and fulfilling economical constraints.

2.5.2 Challenges for Grid Scheduling

Although a Grid also falls into the category of distributed parallel computing environments, it has a lot of unique characteristics which make the scheduling in Grid environments highly difficult. An adequate Grid scheduling system should overcome these challenges to leverage the promising potential of Grid systems, providing high-performance services. The grand challenges imposed by the Grid systems are following:

- **Resource heterogeneity**

  A computational Grid mainly has two categories of resources: networks and computational resources. Heterogeneity exists in both of the two categories of resources. First, networks used to interconnect these computational resources may differ significantly in terms of their bandwidth and communication protocols. A wide-area Grid may have to utilize the best-effort services provided by the Internet. Second, computational resources are usually heterogeneous in that these resources may have different hardware, such as instruction set, computer architectures, number of processor, physical memory size, CPU speed, and so on, and also different software, such as different operating systems, file systems, cluster management software, and so on. The heterogeneity results in differing capability of processing jobs. Resources with different capacity could not be considered uniformly. An adequate scheduling system should address the heterogeneity and further leverage different computing power of diverse resources.

- **Dynamic behavior**

  In traditional parallel computing environments, such as a cluster, the pool of resources is assumed to be fixed or stable. In a Grid environment, dynamics exists in both the networks and computational resources. First, a network shared by many parties cannot provide guaranteed bandwidth. This is particularly true when wide-area networks such as the Internet are involved. Second, both the availability and capability of computational resources will exhibit dynamic behavior. On one hand new resources may join the Grid, and on the other hand, some resources may become unavailable due to problems such as network failure. The capability of resources may vary overtime due to the contention among many parties who share the resources. An adequate scheduler should adapt to such dynamic behavior. After a new resource joins the Grid, the scheduler should be able to detect it automatically and leverage the new resource in the later scheduling decision making. When a computational resource becomes unavailable resulting from an unexceptional failure, mechanisms, such as checkpointing or rescheduling, should be taken to guarantee the reliability of the Grid systems.

- **Site autonomy**

  Typically a Grid may comprise multiple administrative domains. Each domain shares a common security and management policy. Each domain usually authorizes a group of users to use the resources in the domain. Thus applications from non-authorized users should not be eligible to run on the resources in some specific domains. Furthermore, a site is an autonomous computational entity. A shared site will result in many problems. It usually has its own scheduling policy, which complicates the prediction of a job on the site. A single overall *Performance goal* is not feasible for a Grid system since each site has its own performance goal and scheduling decision is made independently of other sites according to its own performance goal.

  Local priority is another important issue. Each site within the Grid has its own scheduling policy. Certain classes of jobs have higher priority only on certain specific resources. For example, it can be expected that local jobs will be assigned higher priorities such that local jobs will be better served on the local resources. Most traditional schedulers are designed with the assumption of having complete control of the underlying resources. Under this assumption, the scheduler has adequate information of resources and therefore effective schedule is much easier to obtain. But in Grid environments, the Grid scheduler has only limited control over the resources. Site autonomy greatly complicates the design of effective Grid scheduling.

- **Resource non-dedication**

  Because of non-dedication of resources, resource usage *contention* is a major issue. Competition may exist in both computational resources and interconnection networks. Due to the non-dedication of resources, a resource may join multiple Grids simultaneously. The workloads from both local users and other Grids share the resource concurrently. The underlying interconnection network is shared as well. One consequence of contention is that behavior and performance can vary over time. For example, in wide area networks using the Internet Protocol suite, network characteristics such as latency and bandwidth may be varying over time. Under such an environment, designing an accurate performance model is extremely difficult. Contention is addressed by assessing the fraction of available resources dynamically, and using this information to predict the fraction available at the time of application to be scheduled. With quality-of-service (QoS) guarantees and resource reservation provided by the underlying resource management system, predicting resource performance is made easier. A scheduler can regard the fraction of those resources that are protected by QoS guarantees as "dedicated" (contention-free) at the guaranteed level. Schedulers must be able to consider the effects of contention and predict the available resource capabilities.

These challenges pose significant obstacles on the problem of designing an efficient and effective scheduling system for Grid environments. Some problems resulting from the above are not solved successfully yet and still are open research issues. In our system model, we have considered resource heterogeneity in terms of processor speed, network heterogeneity in terms of available network bandwidth, dynamic behavior of the Grid system and considerable network delay to reflect real time scenario of the Grid.

Chapter 3 Load Balancing in Grid

3.1 Need for Load Balancing

A Grid system is considered as a collection of autonomous computer (nodes) located at possibly different sites and connected by a communication network. Through the communication network, resources of the system can be shared by users at different locations. Performance enhancement is one of the most important issues in distributed systems. Obviously but expensive ways of achieving this goal are to increase the capacities of the nodes and to add more nodes to the system. Adding more nodes or increasing the capacity of some of the nodes may be required in cases in which all of the nodes in the system are overloaded; however, in many situations poor performance is due to the uneven load distribution throughout the system. As shown in Figure 3.1, due to an uneven and random arrival of jobs at different processors, and/or due to the difference in their processing capacities, quite often there is a build-up of jobs at some processors, while other processors remain *under-loaded.* The performance of the system can often be improved to an acceptable level simply by redistributing the load from highly loaded nodes to lightly loaded nodes. Therefore, load distribution is a cost-effective way for performance enhancement. The problem of load redistribution in distributed systems is recognized as *load balancing.*

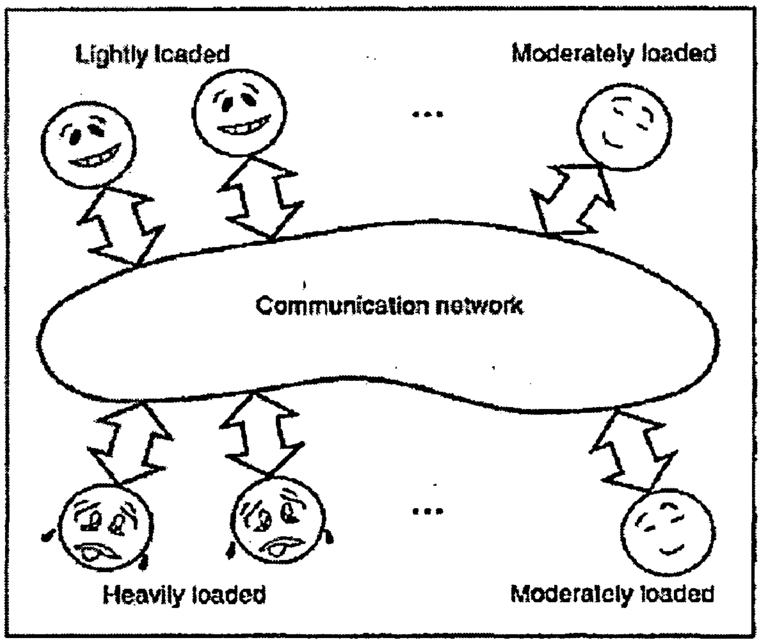


Figure 3.1 A distributed system with no load balancing

3.2 Classification of Load Balancing Algorithms

Any load balancing algorithm for the Grid can be classified into following categories:

- **Classification based on location policy**

  The location policy determines a suitably under-loaded processor. In other words, it locates complementary nodes to/from which a node can send/receive workload to improve overall system performance. Location based policies can be broadly classified as *sender-initiated*, *receiver-initiated* or *symmetrically-initiated* [10, 11, 12, 13]. Sender-initiated policies are those where heavily loaded nodes search for lightly-loaded nodes while receiver-initiated policies are those where lightly loaded nodes search for suitable senders. Symmetrically-initiated policies combine the advantages of these two by requiring both senders and receivers to look for appropriate partners.

- **Classification based on information policy**

  While balancing the load, certain type of information such as number of jobs waiting in queue, job arrival rate, CPU processing rate, etc. at each processor as well as at neighboring processors may be exchanged among the processors for improving the overall performance. Based on the information that can be used, load balancing algorithms are classified as *static*, *dynamic* or *adaptive* [13, 14, 15, 16]. In a static algorithm, the scheduling is carried out according to a predetermined policy. The state of the system at the time of the scheduling is not taken into consideration. On the other hand, a dynamic algorithm adapts its decision to the state of the system. Adaptive algorithms are a special type of dynamic algorithms where the parameters of the algorithm and/or the scheduling policy itself is changed based on the global state of the system.

- **Classification based on degree of centralization**

  According to another classification, based on the degree of centralization, load balancing algorithms could be classified as centralized or decentralized [12, 16]. In a centralized system, the load scheduling is done only by a single processor. Such algorithms are bound to be less reliable than decentralized algorithms, where load scheduling is done by many, if not all, processors in the system. However, decentralized algorithms have the problem of communication overheads incurred by frequent information exchange between processors.

➤ **Classification based on degree of cooperation**

If a distributed scheduling algorithm is adopted, the next issue that should be considered is whether the nodes involved in job scheduling are working cooperatively or independently (non-cooperatively). In the non-cooperative case, individual schedulers act alone as autonomous entities and arrive at decisions regarding their own optimum objects independent of the effects of the decision on the rest of system. Good examples of such schedulers in the Grid are application-level schedulers which are tightly coupled with particular applications and optimize their private individual objectives. In the cooperative case, each Grid scheduler has the responsibility to carry out its own portion of the scheduling task, but all schedulers are working toward a common system-wide goal. Each Grid scheduler's local policy is concerned with making decisions in concert with the other Grid schedulers in order to achieve some global goal, instead of making decisions which will only affect local performance or the performance of a particular job.

3.3 Steps for Performing Dynamic Load Balancing

A practical approach to dynamic load balancing is to divide the problem into the following five phases [14]:

1) **Load Evaluation:** Some estimate of a computer's load must be provided to first determine that a load imbalance exists. Estimates of the workloads associated with individual tasks must also be maintained to determine which tasks should be transferred to best balance the computation.

2) **Profitability Determination:** Once the loads of the computers have been calculated, the presence of a load imbalance can be detected. If the cost of the imbalance exceeds the cost of load balancing, then load balancing should be initiated.

3) **Work Transfer Vector Calculation:** Based on the measurements taken in the first phase, the ideal work transfers necessary to balance the computation are calculated.

4) **Task Selection:** Tasks are selected for transfer or *exchange* to best fulfill the vectors provided by the previous step. Task selection is typically constrained by communication locality and task size considerations.

5) Task Migration: Once selected, tasks are transferred from one computer to another. State and communication integrity must be maintained to ensure algorithmic correctness.

3.4 Related works in Load Balancing

Computational grids have the potential for solving large-scale scientific problems using heterogeneous and geographically distributed resources. However, heterogeneity of resources, high communication delay and dynamic nature are the major technical hurdles that must be overcome before this potential can be realized. One problem that is critical to effective utilization of computational grids is the efficient scheduling of jobs. Numerous researchers have proposed scheduling algorithms for parallel and distributed systems [1, 13, 15, 16, 17, 18] as well as for Grid computing environment [19, 20, 21, 22, 23, 24]. For a dynamic load balancing algorithm, it is unacceptable to frequently exchange state information because of the high communication overheads. In order to reduce the communication overheads, Anand et al. [1] proposed an estimated load information scheduling algorithm and Michael [25] analyzed the usefulness of the extent to which old information can be used to estimate the state of the system. Many job scheduling algorithms [19, 20, 23] have been proposed to deal with the heterogeneity and dynamic nature of distributed systems so as to optimize some figure of merit, for instance, minimize average job response time or better resource utilization. Martin [26] studied the effects of communication latency, overhead and bandwidth in cluster architecture to observe the impact on application performance. Arora et al. [23] proposed a decentralized load balancing algorithm for Grid environment. Although this work attempts to include communication latency between two nodes during triggering process on their model, it did not consider the actual cost for job transfer. Our approach takes job migration cost into account for load balancing decision. In Refs. [19, 20], sender processor collects status information about neighboring processors by communicating with them at every load balancing instant. This can lead to frequent message transfer. For large-scale Grid environment where communication latency is very large, status exchange at each load balancing instant can lead to large communication overhead. In our approach, the problem of frequent exchange of information is alleviated by estimating load, based on system state information received at sufficiently large interval of time.

We have proposed algorithms for heterogeneous Grid environment which are based on estimation approach as carried out in the design of ELISA [1]. In ELISA, load balancing is carried out based on queue lengths. Whenever there is difference in queue length, jobs will be migrated to lightly loaded processor ignoring job migration cost. This cost becomes important factor when communication latency is very large such as for Grid environment and/or jobs size is large. Further, for heterogeneous Grid environment, load balancing decision should consider all affecting factors which are current load at processor, processor heterogeneity, network heterogeneity and migration cost of a job. Our proposed algorithms effectively balance the load by considering all affecting factors.

Chapter 4 Grid System Model

4.1 Introduction

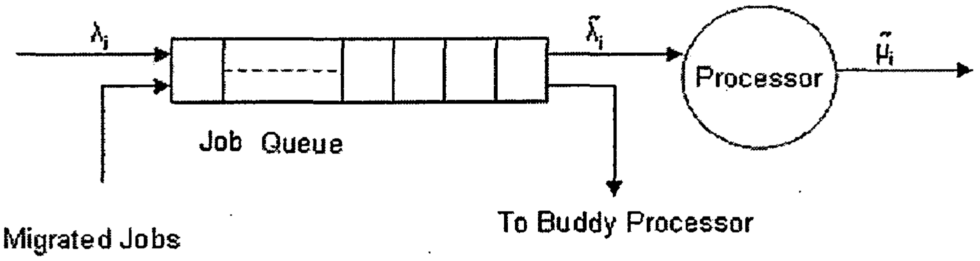


Figure 4.1 System Model

Our Grid system model consists of a set of *M* heterogeneous resources, labeled as $P_1, P_2, ..., P_M$, connected by a communication network. The resources may be of different hardware architecture and processing speed can be different for different resources. There is no possibility of dropping of a job due to unavailability of buffer space as we assume that each resource has an infinite capacity buffer. For any resource P_i, jobs are assumed to arrive randomly at the processors, the inter-arrival time being exponentially distributed with average $1/\lambda_i$. The jobs are assumed to require service time that are exponentially distributed with mean $1/\mu_i$. All jobs are assumed to be mutually independent and can be executed on any node. Thus, each node is modeled as a M|M|1 Markov chain, with the number of jobs queued up for processing at each node representing the state of the system. Job size is assumed to have normal distribution with a given mean and variance. This job size includes both program and data size. As Grid is dynamic in nature, there is no fixed network topology. In our model, we consider arbitrary network topology to capture this constraint. Also data transfer rate is not same for each link connecting two resources. Nodes which are directly connected to a node constitute its *buddy set*. We also assume that each node has knowledge about its buddy nodes (in terms of processor speed and communication latency between them) and load balancing is carried out within buddy sets only. It may be noted that two neighboring buddy sets may have few nodes common to each set. Job arrival rates and service rates are such that for some node (say P_i), $\lambda_i > \mu_i$ (that is, P_i is unstable), but whole system always remains stable, that is

$$\sum_{i=1}^{M} \lambda_i < \sum_{i=1}^{M} \mu_i \qquad (4.1)$$

4.2 List of Notations and Terminology

We first describe notations and terminology that are used throughout the report below.

M	Number of heterogeneous processors $(P_1, P_2, P_3, \ldots, P_M)$
N	Number of jobs to be processed
λ_i	Actual arrival rate for P_i (Poisson distribution)
$1/\mu_i$	Actual mean service time for P_i (Exponential distribution)
CST	Current System Time
T_s	Status exchange period
T_e	Load estimation period
η	Migration Limit
S_i	Normalized measure of speed for P_i
EFT_i^j	Estimated Finish Time of job j on P_i
ERT_i^j	Estimated Run Time of job j on P_i
t_c^j	Communication time for job j
$A_i(T)$	Actual number of job arrivals for P_i in time t
$D_i(T)$	Actual number of job departures for P_i in time t
$EA_i(T)$	Expected number of job arrivals for P_i in time t
$ED_i(T)$	Expected number of job departures for P_i in time t
α	Arrival rate estimation factor
β	Service rate estimation factor
$\tilde{\lambda}_i(T)$	Estimated arrival rate for P_i at time T
$\tilde{\mu}_i(T)$	Estimated service rate for P_i at time T
$\tilde{L}_i(T)$	Estimated load on P_i at time T
$\tilde{L}_{k,i}(T)$	Estimated load on buddy processor P_k calculated by P_i at time T
$Q_i(T)$	Number of jobs waiting in queue for P_i at time T

Table 4.1 List of notations and terminology

4.3 Performance Metrics

In this work, we have considered four performance metrics of relevance at different levels. At the job level, we consider the *Average Response Time* (*ART*) of the jobs processed in the system as the performance metric. If N jobs are processed by the system, then ART can be calculated as follows:

$$ART = \frac{1}{N}\sum_{i=1}^{N}(Finish_i - Arrival_i) \tag{4.2}$$

where $Arrival_i$ is the time at which the ith job arrives and $Finish_i$ is the time at which it leaves the system. The delay due to job transfer, waiting time in queue and processing time, together constitute the response time.

At the system level, we consider *Total Execution Time* as performance metric to measure algorithm's efficiency. It indicates time at which all N jobs get executed.

At the processor level, we consider *Resource Utilization* as performance metric. It is ratio between processor's busy time to sum of processor's busy and idle time.

$$U_i = \frac{Busy_i}{Busy_i + Idle_i} \tag{4.3}$$

where $Busy_i$ indicates amount of time P_i remains busy and $Idle_i$ indicates amount of time P_i remains idle during total execution time of N jobs.

We also consider *Total Job Migrations* as performance metric which indicates number of job migrations performed for execution of N jobs. In the event of high migration cost, it is not always advisable to perform job migration. This metric indicates adaptive-ness of an algorithm in the event of varying job migration cost.

Chapter 5 Design of Load Balancing Algorithms

5.1 Introduction

In any distributed systems, even simple load sharing policies yields significant improvements in performance over the no sharing case. But in a computational Grid, as resources are geographically distributed and located at different sites, job transfer time from one site to another site is very significant factor for load balancing. Communication latency is also very large for Wide Area Network (WAN) through which Grid resources are normally connected. Due to these reasons, one can not ignore job transfer cost when taking job migration decision. Further, when resources are heterogeneous, we need to assign jobs to processors according to its performance. Our proposed algorithms consider this fact. Our algorithms are based on ELISA [1] and does parameter estimation and information exchange at regular intervals. We shall first describe the process of parameter estimation and the way in which load balancing is carried out in our algorithms, in general. This will also be useful in understanding the terminology associated with the notations used.

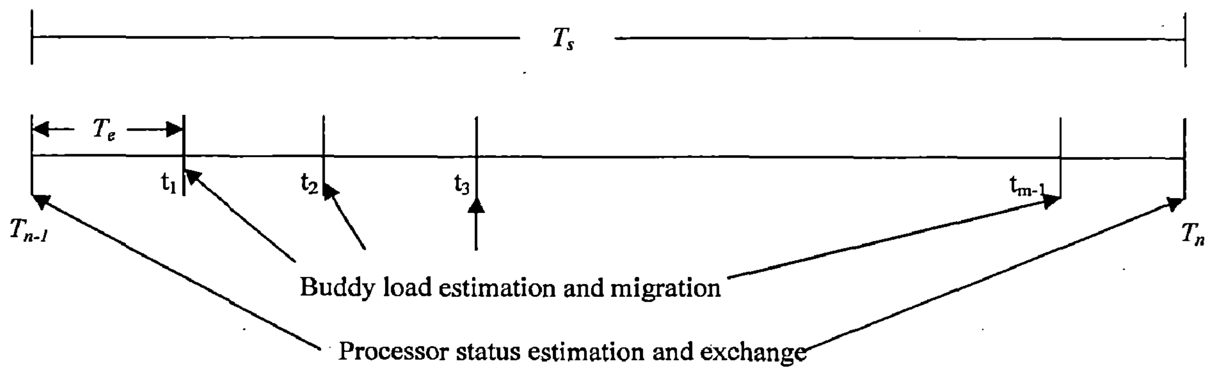

T_s
T_e
t_1
t_2
t_3
t_{m-1}
T_{n-1}
T_n
Buddy load estimation and migration
Processor status estimation and exchange

Figure 5.1 Estimation and status exchange intervals

As shown in Figure 5.1, at each periodic interval of time T_s, called the status exchange interval, each P_i in the system calculates its status parameters which are estimated arrival rate, service rate and load on processor. Each P_i in the system exchanges its status information with the processors in its buddy set. The instant at which this information exchange takes place is called a status exchange instant. In Figure 5.1, T_{n-1} and T_n represent the status exchange instant. Each P_i calculates its status information at status exchange instant T_{n-1} using following relationships:

$$\widetilde{\lambda}_i(T_{n-1}) = \alpha * \widetilde{\lambda}_i(T_{n-2}) + (1-\alpha) * (A_i(T_s)/T_s) \tag{5.1}$$

$$\widetilde{\mu}_i(T_{n-1}) = \beta * \widetilde{\mu}_i(T_{n-2}) + (1-\beta) * (D_i(T_s)/T_s) \tag{5.2}$$

$$\widetilde{L}_i(T_{n-1}) = Q_i(T_{n-1}) / \widetilde{\mu}_i(T_{n-1}) \tag{5.3}$$

Thus, in the above relationship, by tuning the parameter α ($0 <= \alpha <= 1$), one can vary the estimate. A value of 0.5 for α would mean that an equal weight has been considered for the current period and the previous estimate of the arrival rate. Similarly, we tune the parameter β ($0 <= \beta <= 1$) for service rate estimation.

Each status exchange period is further divided into equal subintervals called estimation interval T_e. These points are known as estimation instants. In Figure 5.1, $t_1, t_2, \ldots, t_{m-1}$ represent estimation instants. As each processor balances the load within its buddy set, every processor estimates the load in the processors belonging to its buddy set at each estimation instants. Each P_i calculates estimated load on its buddy processor P_k using following equations:

$$\widetilde{L}_{k,i}(T_{n-1} + t_i) = (EA_k(T_e) - ED_k(T_e)) / \widetilde{\mu}_k(T_{n-1}) + \widetilde{L}_{k,i}(T_{n-1} + t_{i-1}) \tag{5.4}$$

where $i = 1, 2, 3, \ldots, m-1$ and

$EA_k(T_e) = a$ such that

$$\sum_{x=0}^{a} \frac{e^{(-\widetilde{\lambda}_k(T_{n-1})*T_e)} * (\widetilde{\lambda}_k(T_{n-1}) * T_e)^x}{x!} \approx 1 \tag{5.5}$$

$ED_k(T_e) = d$ [1] such that

$$\sum_{x=0}^{d} \frac{e^{(-\widetilde{\mu}_k(T_{n-1})*T_e)} * (\widetilde{\mu}_k(T_{n-1}) * T_e)^x}{x!} \approx 1 \tag{5.6}$$

Depending on the accuracy required, computations of $EA_k(T_e)$ and $ED_k(T_e)$ can be terminated after computing a sufficiently large number of terms in equations (5.5) and (5.6).

The status exchange instants and the estimation instants together constitute the set of transfer instants $(T_{n-1}, t_1, t_2, \ldots, t_{m-1}, T_n)$ in Figure 5.1. At the transfer instants, rescheduling of jobs is carried out. Thus, the decision to transfer jobs and the actual transfer of jobs is done at the transfer instants. By making the interval between status exchange instants large, and by

[1] Note that number of job departures can not be greater than number of job arrivals. That is,

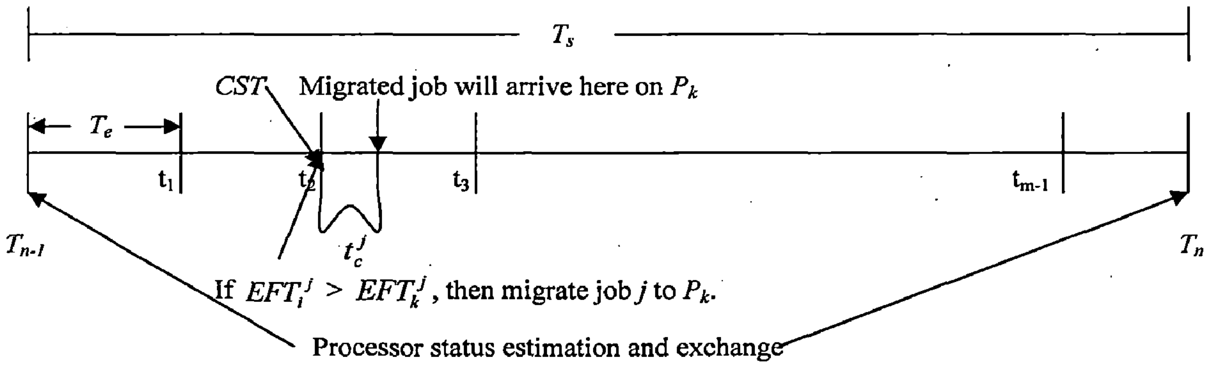
$$ED_k(T_e) \le (EA_k(T_e) + \widetilde{L}_{k,i}(T_{n-1} + t_{i-1}) * \widetilde{\mu}_k(T_{n-1}))$$

restricting the exchange of information to the buddy set, the communication overheads are kept at a minimum.

5.2 Modified ELISA (MELISA)

Although ELISA primarily works on estimates, it is mainly proposed for cluster based super-computing systems wherein communication cost is not very large as resources are connected through high bandwidth network. However, for Grid based super-computing systems, the transfer delays are significantly high contributing to a large communication cost. Thus direct applicability of ELISA will yield inferior performance that is unacceptable for the Grid based systems. Later, in our simulation study we highlight this fact. Further, when resources are heterogeneous, we need to assign jobs to processors according to its performance. Due to these reasons, one needs to take into account all affecting factors for load balancing to achieve better performance. Hence, we revisit the design of ELISA and introduce the job transfer rate and resource heterogeneity explicitly in the formulation that is more akin for Grids.

In ELISA, at every status exchange time period T_s, each P_i communicates its status (queue length, estimate of arrival rate) to all its buddy processors. At each estimation instant T_e, every processor calculates queue length on buddy processors using estimated arrival rate and exact service rate of buddy processor. P_i will take decision of job migration if its queue length is greater than an average queue length in its buddy set.



T_s
CST
Migrated job will arrive here on P_k
T_e
t_1
t_2
t_3
t_{m-1}
T_{n-1}
t_c^j
T_n
If $EFT_i^j > EFT_k^j$, then migrate job j to P_k.
Processor status estimation and exchange

Figure 5.2 Job migration decision in MELISA

In the design of MELISA, as shown in Figure 5.2, each P_i estimates its arrival rate, service rate and the load using equations (5.1), (5.2), and (5.3) at each status exchange instant. At each estimation instant, P_i calculates load on its all buddy processors using equations (5.4), (5.5), and (5.6). Based on this calculated buddy load, each processor calculates average load in its buddy set. P_i will take decision of job migration if its load is greater than an average load in its buddy set and will try to distribute its load such that load on all buddy processors get finished at almost same time taking into account node's heterogeneity in terms of processor speed. This average buddy load can be calculated using following relationships.

```
Main Algorithm
At the status exchange instant, for each processor:
1. Estimate the arrival rate, service rate and load on processor using
   equations (5.1), (5.2) and (5.3).
2. Communicate the status defined by a 3-tuple as: <estimated arrival
   rate, estimated service rate, estimated load> to all processors in
   the buddy set.
3. Call TRANSFER.
At the estimation instant, for each processor
1. Estimate the load for each processor in the buddy set using equation
   (5.4), (5.5) and (5.6).
2. Call TRANSFER.
```

```
Procedure TRANSFER by $P_i$, $i=1,2,...,M$.
1. Estimate an average normalized buddy load using equation (5.7).
2. If load of a processor is greater than average load (as computed in
     1), then
   a) Construct active set as follows: if a processor in the buddy set has
      load less than the average normalized buddy load, include
      processor in active set.
   b) Determine how much load can be transferred to buddy processors
      in active set such that load on all processors gets finished at
      almost same time.
   c) Attempt to migrate the load in excess over average buddy load to
      all buddy processors in active set by calculating $EFT_k^j$ on
      destination processor and migrating job only if $EFT_k^j < EFT_i^j$.
```

Figure 5.3 MELISA Load balancing algorithm

Let S_i denote the weight of a processor P_i which is a normalized measure of its speed. So a value of 2 for S_i means P_i will take half amount of time to execute job than time taken by reference processor[2] having value of 1 for S_i. Here, each P_i will calculate normalized buddy average load $[(NBL_{avg})_i]$ using value of $\tilde{L}_{k,i}(T)$ and S_i by following equation:

[2] This could be an abstract processor within the system.

$$(NBL_{avg})_i = \frac{\sum_{k \in buddyset_i} S_k * \widetilde{L}_{k,i}(T)}{\sum_{k \in buddyset_i} S_k} \tag{5.7}$$

$(NBL_{avg})_i$ indicates normalized average buddy load for reference processor. P_i is considered as a sender processor, if $(NBL_{avg})_i < S_i * \widetilde{L}_i(T)$. Now P_i will try to transfer its extra load to all receiver processors P_k such that they receive extra load based on their current load ($\widetilde{L}_{k,i}(T)$) and processor weight (S_k). After determining how much load P_i can transfer to P_k, as shown in Figure 5.2, P_i will calculate expected finish time of job j on buddy processor (P_k) by estimating load on P_k at time $CST + t_c^j$ (where t_c^j is migration time for job j from P_i to P_k). Job will be migrated to P_k ***only if***

$$EFT_k^j \; < \; EFT_i^j \tag{5.8}$$

where

$$EFT_i^j = Q_i(CST) / \widetilde{\mu}_i(T_{n-1}) + ERT_i^j \tag{5.9}$$

$$EFT_k^j = \max\left((\widetilde{L}_{k,i}(CST) + (EA_k(t_c^j) - ED_k(t_c^j)) / \widetilde{\mu}_k(T_{n-1})),\, t_c^j \right) + ERT_k^j \tag{5.10}$$

In equation (5.10), first term which is maximum of two values - approximate wait time of job j on P_k and job transfer time - indicates expected starting time of job j on P_k. We assume that these activities can be performed simultaneously. So job will be migrated only if its expected finish time on destination processor is less than expected finish time on source processor. The complete working of MELISA is shown in Figure 5.3.

5.3 Receiver-initiated MELISA (R-MELISA)

It is expected that receiver driven strategy gives better performance than sender initiated policy since receiver can best determine how much load it can accept from other processors. We also observed similar type of behavior when comparing sender-initiated policy and receiver-initiated policy for ELISA. So we have designed receiver-initiated MELISA (R-MELISA) to compare results of both sender-initiated and receiver-initiated policies. The basic difference in this approach and in MELISA is that in R-MELISA, it is receiver processor who will inform to sender processor how much load it can accept from sender processor. In MELISA, it is sender processor who determines how much load it should migrate to other buddy processor to balance the load in buddy set.

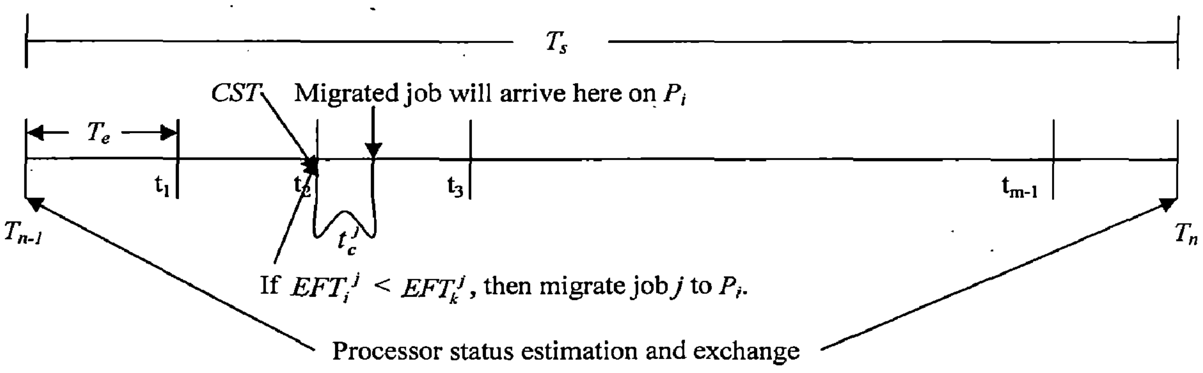

T_s
CST
Migrated job will arrive here on P_i
T_e
t_1
t_2
t_3
t_{m-1}
T_{n-1}
T_n
t_c^j
If EFT_i^j < EFT_k^j, then migrate job j to P_i.
Processor status estimation and exchange

Figure 5.4 Job migration decision in R-MELISA

Here, similar to MELISA algorithm, each processor P_i calculates its status parameters which are estimated arrival rate, service rate and load at every status exchange period T_s using equations (5.1), (5.2), and (5.3). This information gets exchanged to every buddy processor in buddy set. At each estimation instant, P_i calculates load on its all buddy processors using equations (5.4), (5.5), and (5.6). Based on this calculated buddy load, each processor calculates average buddy load in its buddy set. P_i will take decision of accepting a job if its load is less than an average load in its buddy set. This normalized buddy average load $[(NBL_{avg})_i]$ is calculated using value of $\tilde{L}_{k,i}(T)$ and S_i by equation (5.7). P_i is considered as a receiver processor, if $(NBL_{avg})_i > S_i * \tilde{L}_i(T)$. Now P_i will determine how much load it can accept from all sender processor P_k (having $(NBL_{avg})_i < S_k * \tilde{L}_k(T)$) using buddy information which is current buddy load ($\tilde{L}_{k,i}(T)$) and processor weight (S_k). After determining how much load P_i can accept from P_k, it will inform to P_k the amount of load it can transfer to P_i. As shown in Figure 5.4, P_k will calculate expected finish time of job j on buddy processor P_i by estimating load on P_i at time $CST + t_c^j$. Job will be migrated to P_i, ***only if,***

$$EFT_i^j < EFT_k^j \tag{5.11}$$

where

$$EFT_k^j = Q_k(CST)/\tilde{\mu}_k(T_{n-1}) + ERT_k^j \tag{5.12}$$

$$EFT_i^j = \max\left(\left(\tilde{L}_{i,k}(CST) + (EA_i(t_c^j) - ED_i(t_c^j))/\tilde{\mu}_i(T_{n-1})\right), t_c^j\right) + ERT_i^j \tag{5.13}$$

So job j will be migrated to receiver processor P_i, only if its expected finish time on destination processor (P_i) is less than expected finish time on source processor (P_k). Figure 5.5 shows complete working of R-MELISA.

```
Main Algorithm
At the status exchange instant, for each processor:
1. Estimate the arrival rate, service rate and load on processor using
   equations (5.1), (5.2) and (5.3).
2. Communicate the status defined by a 3-tuple as: <estimated arrival rate,
   estimated service rate, estimated load> to all processors in the buddy
   set.
3. Call TRANSFER.
At the estimation instant, for each processor
1. Estimate the load for each processor in the buddy set using equations
   (5.4), (5.5) and (5.6).
2. Call TRANSFER.
```

```
Procedure TRANSFER by $P_i$, $i=1,2,\dots,M$.
1. Estimate an average normalized buddy load using equation (5.7).
2. If load of a processor is less than average load (as computed in 1), then
    a) Determine buddy processors whose load is greater than average
       normalized buddy load.
    b) Determine how much load can be accepted from sender buddy
       processors such that load on all processors gets finished at almost
       same time.
    c) Inform to all sender processors amount of load they can transfer to $P_i$.
    d) Sender processor $P_k$ will attempt to migrate the load in excess over
       average buddy load to all receiver processors by calculating $EFT_i^j$
       on destination processor and migrating job only if $EFT_k^j > EFT_i^j$.
```

Figure 5.5 R-MELISA Load Balancing Algorithm

5.4 Load Balancing on Arrival (LBA)

The applicability of MELISA (and R-MELISA) is more appropriate to cases wherein Grid infrastructure is large and communication delays are significant such as for InterGrid. Moreover, the load balancing is done only at transfer instants (which can be estimation instant or status exchange instant) for ELISA and MELISA, which is acceptable as communication delays are severe. However, this becomes an obvious inherent disadvantage when either of these algorithms is applied to small scale Grids such as for IntraGrids. That is, with these approaches, jobs need to wait till next transfer instant for migration and due to the random arrival rate and service rate at each processor, it is possible that load does not get distributed evenly across all processors. In this case, there can be large waiting times at highly loaded processors whereas lightly loaded processors continue to remain idle. Jobs from highly loaded processors will be migrated to lightly loaded processor after finite amount of time depending on value of T_e, T_s and distance between highly loaded processor and lightly loaded processor. This can lead to performance degradation even for moderate value of T_e and T_s. Our simulation results also support this observation. Consequently, there seems to be a need for designing an alternate load balancing algorithm to take into account of such situations.

We design and propose a new algorithm, referred to as *Load Balancing on Arrival* (LBA), which balances load by transferring job on its arrival epoch rather than waiting for next transfer instant. This is clearly a faster reaction to respond to higher arrival rates on smaller Grids. In LBA algorithm, instead of estimating the expected finish time of a job at every estimation time period T_e, it will be calculated on each arrival of a job to a processor. Here estimating finish time of a job is an aperiodic event and job migration will now happen aperiodically. So when load is not distributed evenly across all processors, job will be migrated to lightly loaded processors much faster in LBA approach than in (M)ELISA.

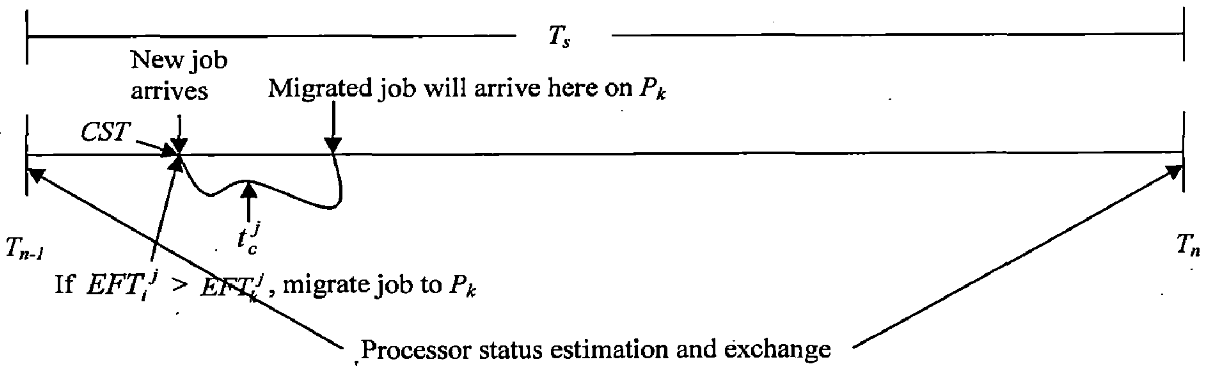
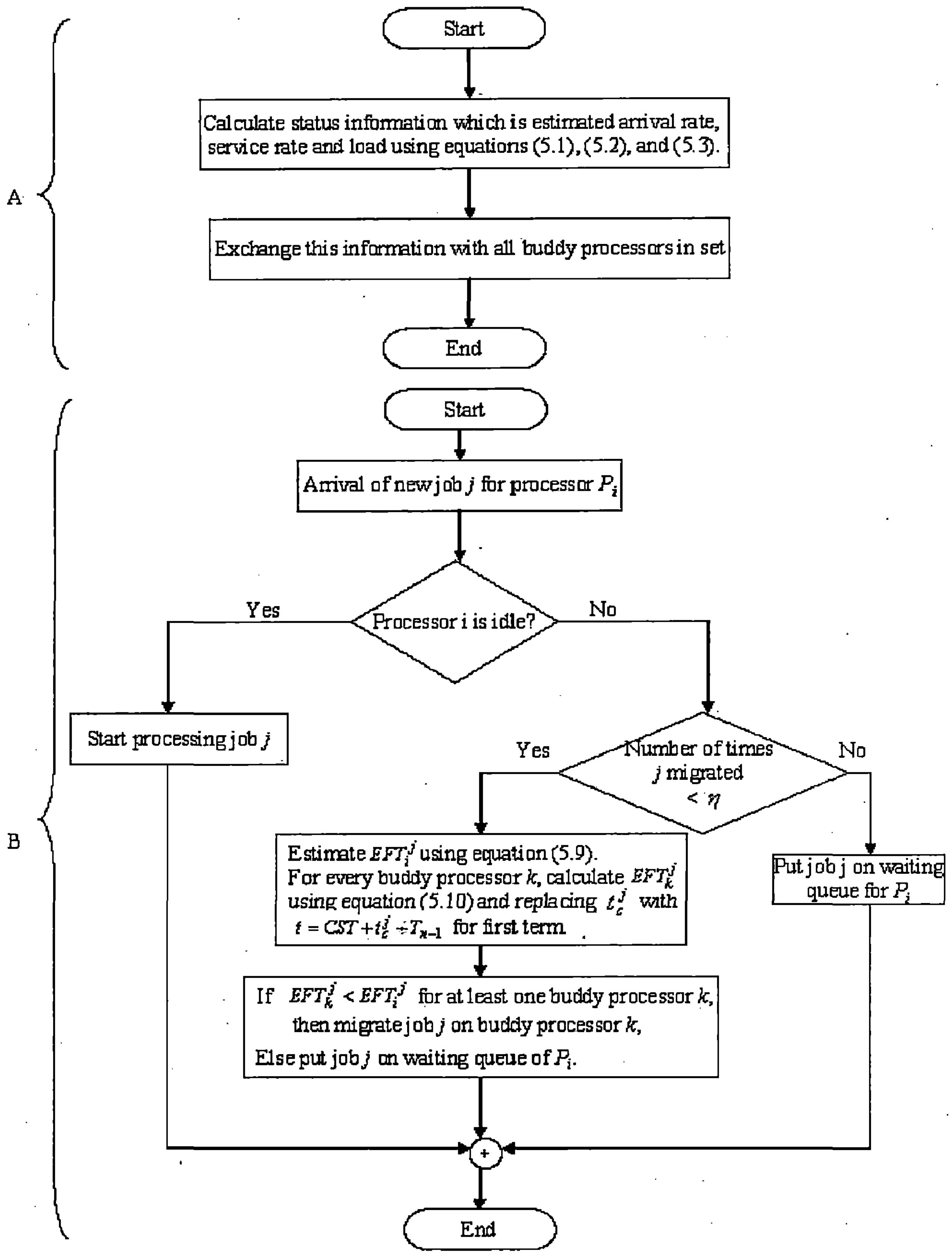

T_s
New job arrives
Migrated job will arrive here on P_k
CST
t_c^j
T_{n-1}
T_n
If $EFT_i^j > EFT_k^j$, migrate job to P_k
Processor status estimation and exchange

Figure 5.6 Job migration decision in LBA

In this approach (refer to Figure 5.6), similar to MELISA algorithm, each processor P_i calculates its status parameters which are estimated arrival rate, service rate and load at every status exchange period T_s using equations (5.1), (5.2), and (5.3). This information gets exchanged to every buddy processor in buddy set. On every job arrival, processor P_i will calculate the expected finish time of job j on buddy processor P_k by estimating load on P_k at time $CST + t_c^j$ (where t_c^j is communication time for job j from P_i to P_k) using equation (5.10). For this estimation, P_i will calculate expected number of arrival and departure on buddy processor P_k for time period $t = CST + t_c^j - T_{n-1}$[3]. If any buddy processor P_k can start execution of this job before processor P_i, then that job will be migrated to P_k. Flowchart for LBA is shown in Figure 5.7.

[3] Here, T_{n-1} is last status exchange instant.


Start
Calculate status information which is estimated arrival rate, service rate and load using equations (5.1), (5.2), and (5.3).
Exchange this information with all buddy processors in set
End
A
Start
Arrival of new job j for processor P_i
Processor i is idle?
Yes
No
Start processing job j
Number of times j migrated $< \eta$
Yes
No
Estimate EFT_i^j using equation (5.9).
For every buddy processor k, calculate EFT_k^j using equation (5.10) and replacing t_c^j with $t = CST + t_c^j + T_{x-1}$ for first term
Put job j on waiting queue for P_i
If $EFT_k^j < EFT_i^j$ for at least one buddy processor k, then migrate job j on buddy processor k,
Else put job j on waiting queue of P_i.
+
End
B

A: Processing by each P_i on every status exchange instant (T_x)

B: Processing by P_i on arrival of job j

Figure 5.7   Flowchart for LBA

5.5 Reference Algorithms

We have used three algorithms, which are relevant to our context, as reference algorithms to compare results of our algorithms.

5.5.1 Perfect Information Algorithm (PIA)

In Perfect Information Algorithm (PIA), each processor has perfect information about the state (in terms of load, arrival rate and service rate) of every other processor in its buddy set. When a job arrives, processor computes job's finish time on all buddy processors using exact information about current load of buddy processor, its arrival rate and service rate. Source processor selects buddy processor with the minimum finish time and immediately migrate job on that buddy processor if it can start job earlier than this processor. Although maintaining up-to-date information about all buddy processors require plenty of message transmission, this algorithm basically provides lower bound for our LBA algorithm.

5.5.2 Estimated Load Information Scheduling Algorithm (ELISA)

In ELISA [1], each processor estimates queue length of its buddy processor at estimation instant using information exchanged at status exchange instant. Information exchanged at status exchange instant includes queue length and estimated arrival rate. This algorithm assumes to have perfect information about service rate of each processor. Refer to Appendix A for a brief explanation of ELISA.

5.5.3 Load Balancing based on Load and processor Speed (LBLS)

Owing to resource heterogeneity, queue length is not always best criteria for determining load imbalance. Instead, product of average processing time of a job and queue length provides better load index for balancing load. In this approach, load balancing is done based on load (in terms of expected time to execute all jobs waiting in queue) rather than based on queue length. Here, P_i will take decision of job migration if its load is greater than an average load in its buddy set and will try to distribute its load such that load on all buddy processors get finished at almost same time on all buddy processors taking into account node's heterogeneity in terms of processor speed.

Chapter 6 Performance Evaluation and Discussion

6.1 Simulation Model

Now, we present the results of our simulation study and compare the performance of our proposed algorithms with reference algorithms discussed in Section 5.5. The amount of information that is made available for use at the instant of decision making for transfer of jobs is expected to have a significant effect on the relative performance of the algorithms. Table 6.1 summarizes the information that the algorithms use for scheduling of jobs.

Algorithm	Arrival rate	Service rate	System state
ELISA	Estimated Information	Perfect Information	Estimated Information
LBLS	Estimated Information	Estimated Information	Estimated Information
PIA	Perfect Information	Perfect Information	Perfect Information
MELISA	Estimated Information	Estimated Information	Estimated Information
R-MELISA	Estimated Information	Estimated Information	Estimated Information
LBA	Estimated Information	Estimated Information	Estimated Information

Table 6.1 Information used by algorithms

In our simulation model, we have considered *16* heterogeneous processors connected by communication channels. Here heterogeneity exist in terms of processor speed, instruction set and hardware architecture. We assume that any job can be run on any processor, but amount of time taken to execute a job is different on different processors and depends on processor speed. These processors are connected through an arbitrary network topology to reflect dynamic behavior of the Grid. This network topology is generated by a graph generator tool as shown in Figure 6.1. Weight on each link indicates data transfer rate in Mbps (Mega bits per second). Various parameter values used for simulation are shown in Table 6.2. These parameter values are used for all cases unless otherwise stated explicitly.

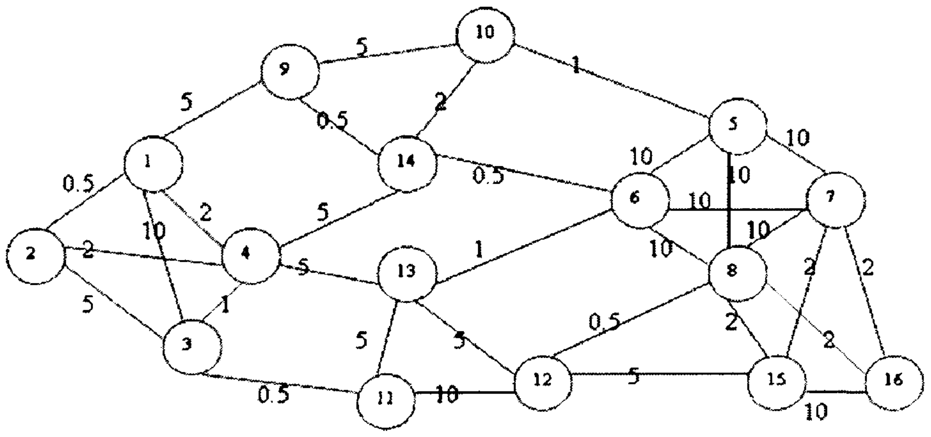


Figure 6.1 Network topology

Parameter	Value
Mean inter-arrival time	Exponentially distributed in *[1,4]*
Mean service time	Exponentially distributed in *[1,4]*
Threshold level for load distribution	*1*
N	*10000*
T_s	*20*
T_e	*4*
α	*0.5*
β	*0.5*
η	*4*
Job Size	Normal distribution with μ=*50MB* and σ=*10MB*

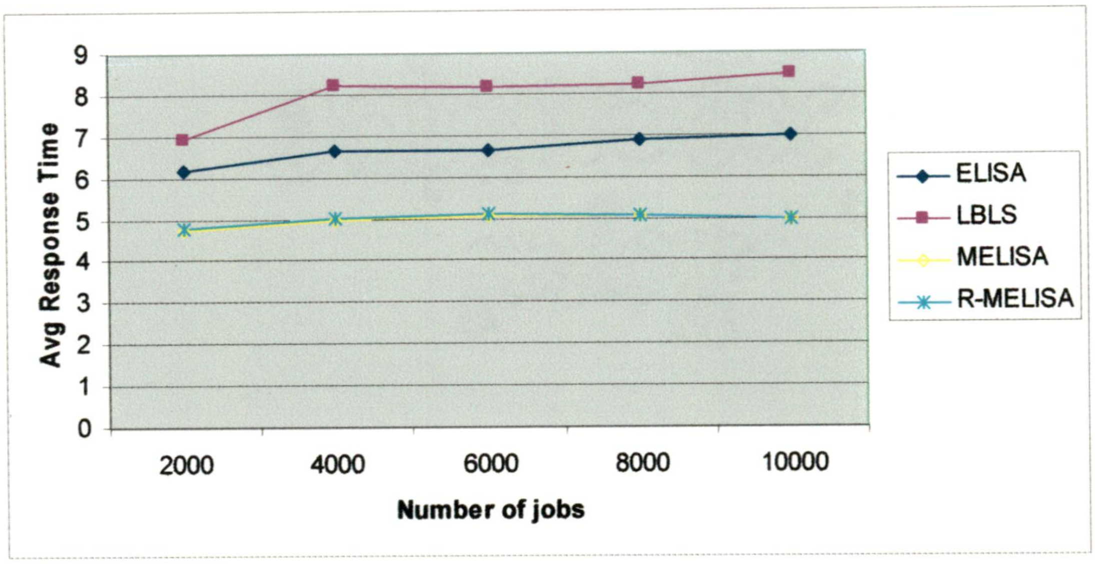
Table 6.2 Parameter values

6.2 Performance of MELISA and R-MELISA

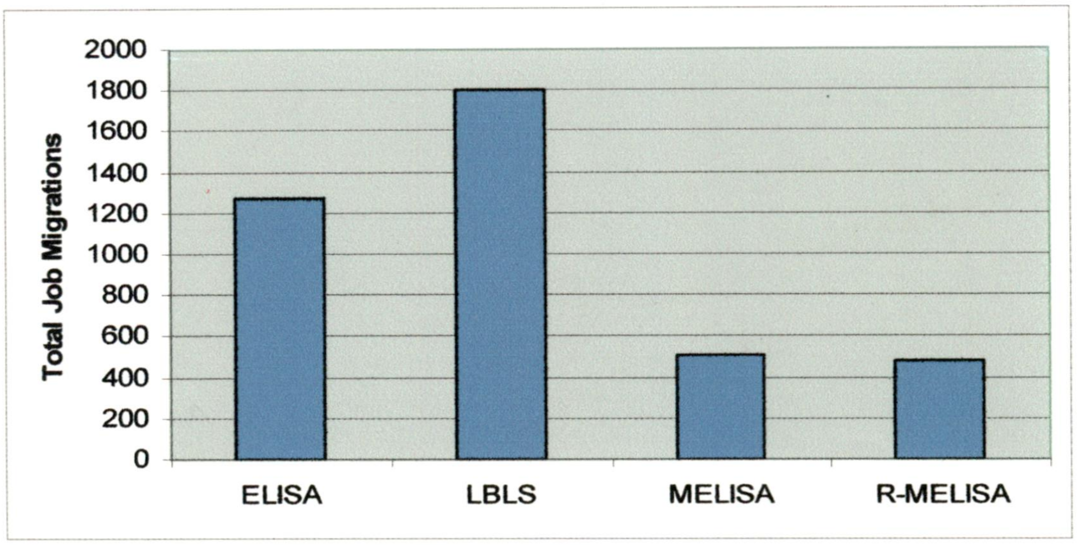
In this section, we will evaluate performance of our load balancing algorithms MELISA and R-MELISA with ELISA and LBLS. As R-MELISA differs from MELISA only in who initiates load balancing, it would be better to compare result of R-MELISA with MELISA.

6.2.1 Heterogeneous Case

For heterogeneous case, we considered different data transfer rate for each link as shown in Figure 6.1. We also considered resource heterogeneity by setting value of S_i to *2* for randomly half of processors. Our algorithms MELISA and R-MELISA give better performance for *ART* as can be seen from Figure 6.2(a). As from Figure 6.2(b), total number of migration performed by our algorithms is less than other reference algorithms as it is not always advisable to perform migration in event of low data transfer rate and resource heterogeneity. Comparing results of MELISA and R-MELISA, we can conclude that both algorithms give almost same performance in terms of *ART*. Also number of job migrations performed by each algorithm is also same. So there is not much performance difference for sender-initiated policy and receiver-initiated policy in MELISA case.



(a) ART comparison for heterogeneous case



(b) Total Job Migrations comparison for heterogeneous case

Figure 6.2 Performance measure of MELISA and R-MELISA for heterogeneous case

6.2.2 Homogeneous Case

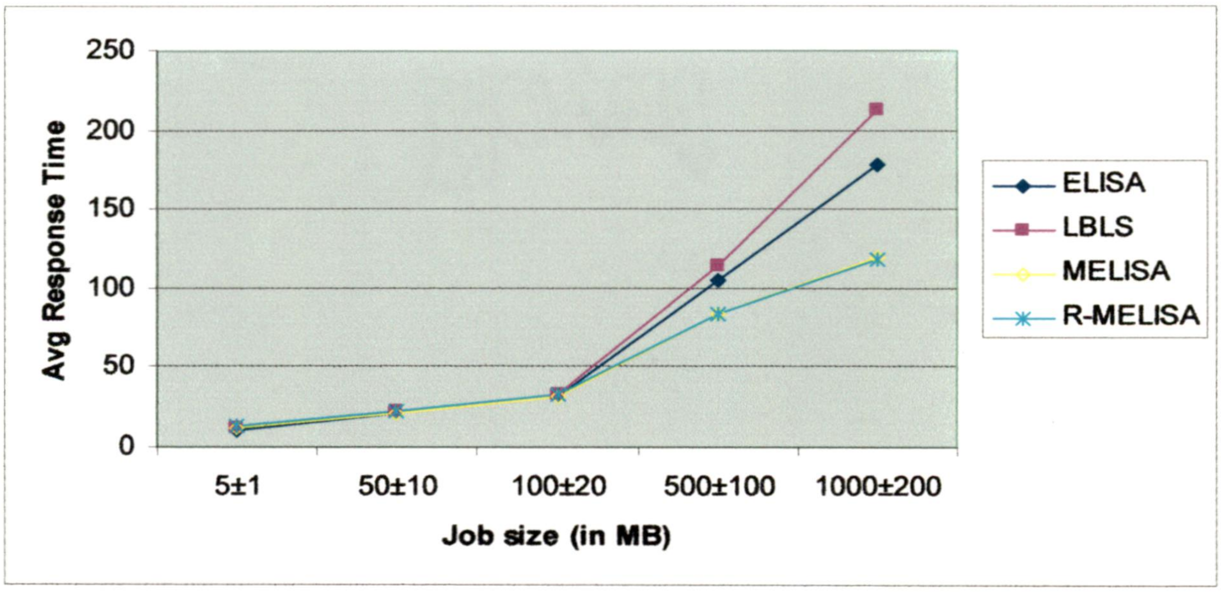
This is a special case to our heterogeneous environment. In this case, we have considered all nodes are homogeneous, that means S_i is set to *1* for all processors. Also, network bandwidth is fixed across link. The main difference between ELISA and MELISA (R-MELISA) algorithm is that MELISA algorithm takes into account the job migration cost when balancing the load across buddy processors. Job migration cost is influenced mainly by two parameters: Job size and data transfer rate. So by varying job size and/or data transfer rate, we can evaluate performance of MELISA and R-MELISA algorithms over ELISA. The following two subsections compare results of MELISA and R-MELISA with ELISA and LBLS for varying job sizes and data transfer rates.

6.2.2.1 Effect of job size

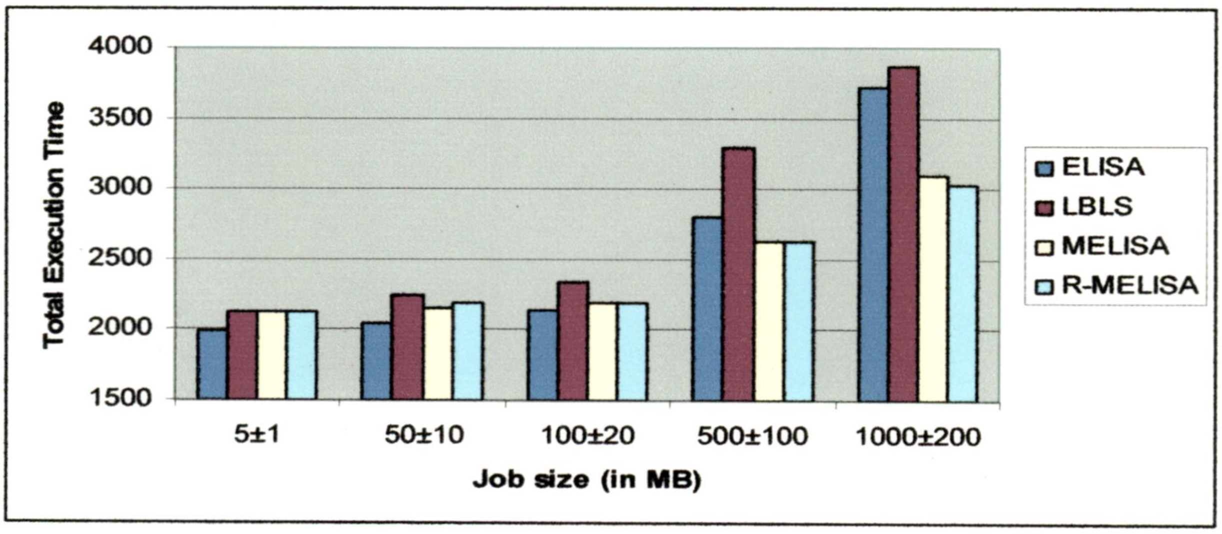
In this set of experiments, we vary job size to measure effect of job size on *ART* and execution time. The job sizes in our experiments range from *5MB±1MB* to *1000MB±200MB* and we set data transfer rate to *10Mbps*. From Figure 6.3(a), we observe that as job size increases, MELISA and R-MELISA give better performance than ELISA and LBLS and similar behavior is exhibited for execution time from Figure 6.3(b). However, when job sizes are very small, job transfer time becomes negligible and the performance remains almost same for all algorithms. But when job size is very large, clearly, MELISA and R-MELISA outperform ELISA and LBLS, as job migration cost becomes an important factor for load balancing decision. Further, the number of migrations carried out by these algorithms is shown in Figure 6.3(c). From this figure, it may be observed that the number of job migrations performed by MELISA and R-MELISA is significantly less when compared with ELISA and LBLS in the case of large job size. This is an important property of MELISA (R-MELISA) that makes it applicable for large-scale Grid systems.

6.2.2.2 Effect of data transfer rate

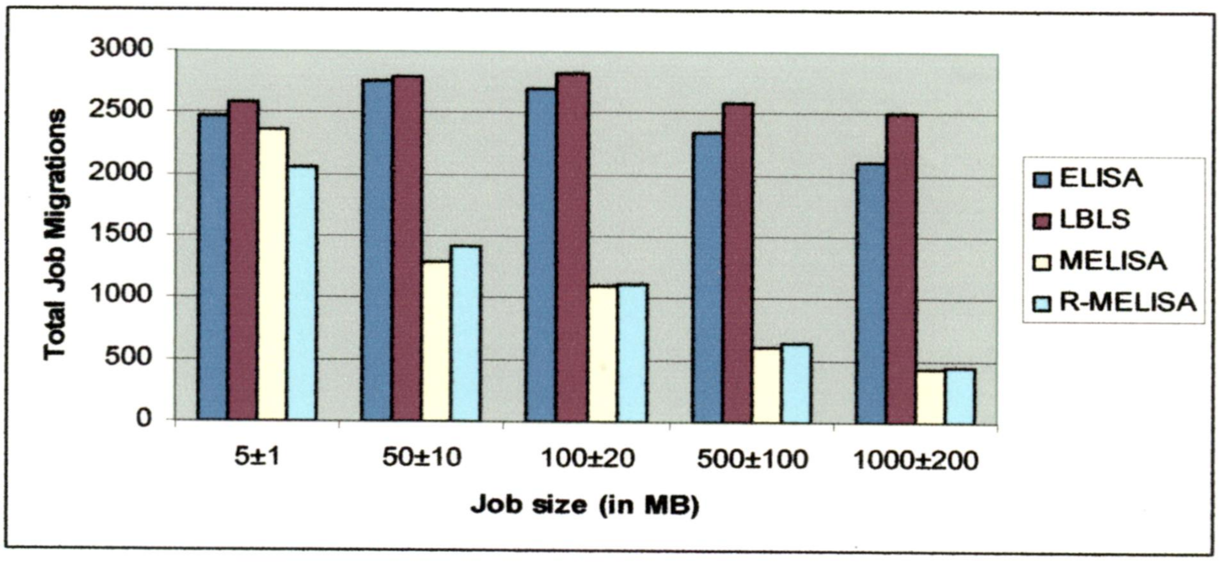
Here we consider the performance of our two algorithms under the influence of data transfer rates. We vary the data transfer rates in the range between *0.5Mbps* to *100Mbps*. For this analysis, we have set job size as *50MB±10MB*. Results are shown in Figures 6.4(a) and 6.4(b). From the figures, we observe that MELISA and R-MELISA give better performance when data transfer rate is very low. For higher data transfer rate, job migration cost is negligible and performance is same for all algorithms. However, when data transfer rate is very low, job migration cost is high and we should migrate job only if it is beneficial. MELISA and R-MELISA take this migration cost into account and that is why total number of migrations for lower data transfer rate is less than for higher data transfer rate, whereas, for ELISA and LBLS, number of migrations remains almost same. (Refer Figure 6.4(c)).



(a) Effect of job size on ART

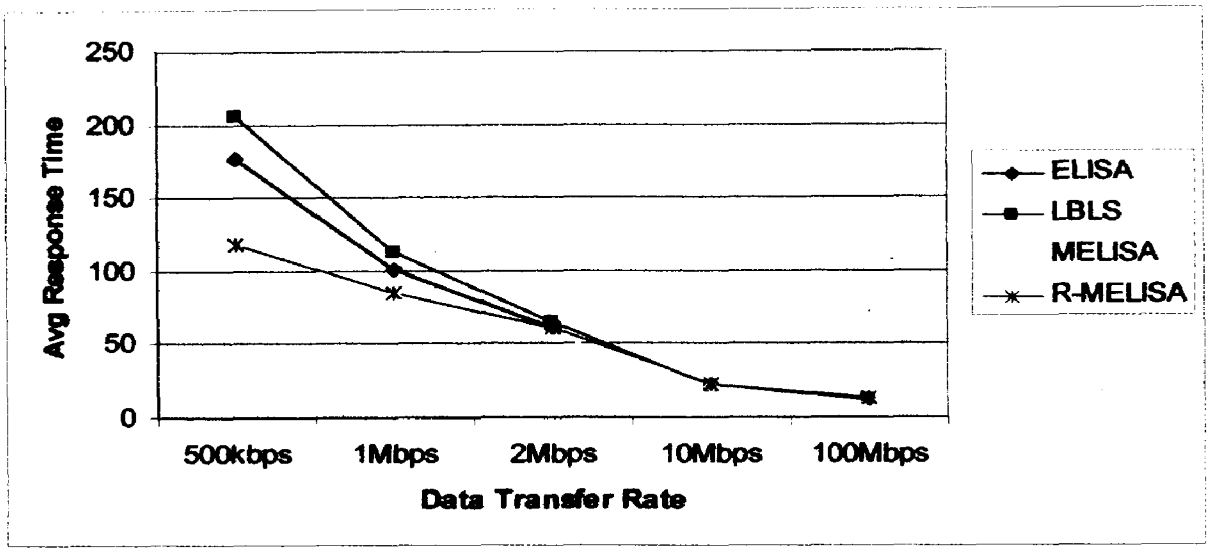


(b) Effect of job size on Total Execution Time

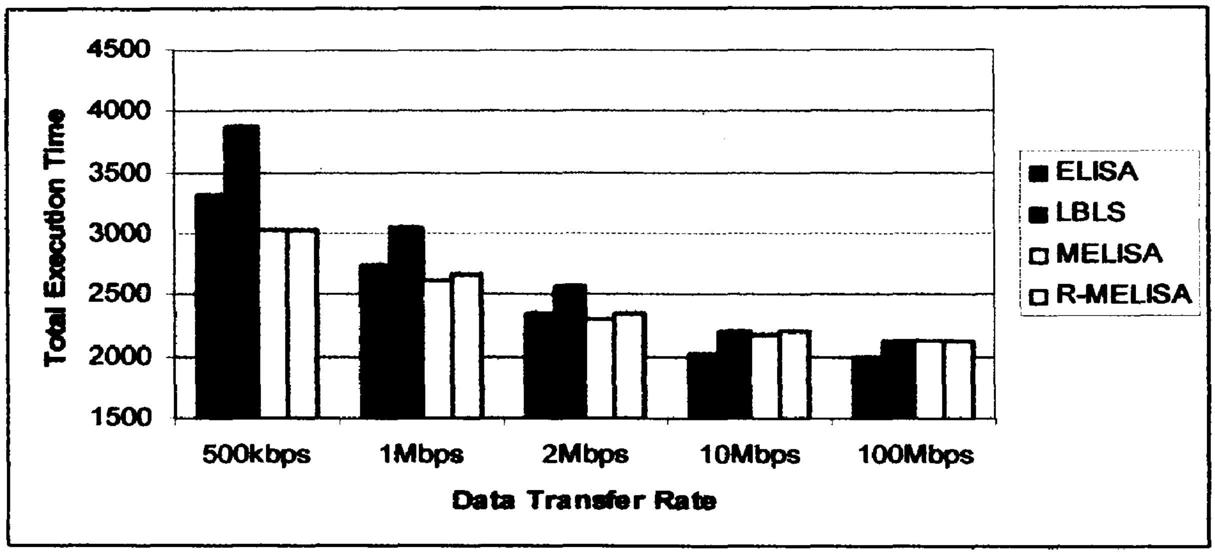


(c) Effect of job size on Total Job Migrations

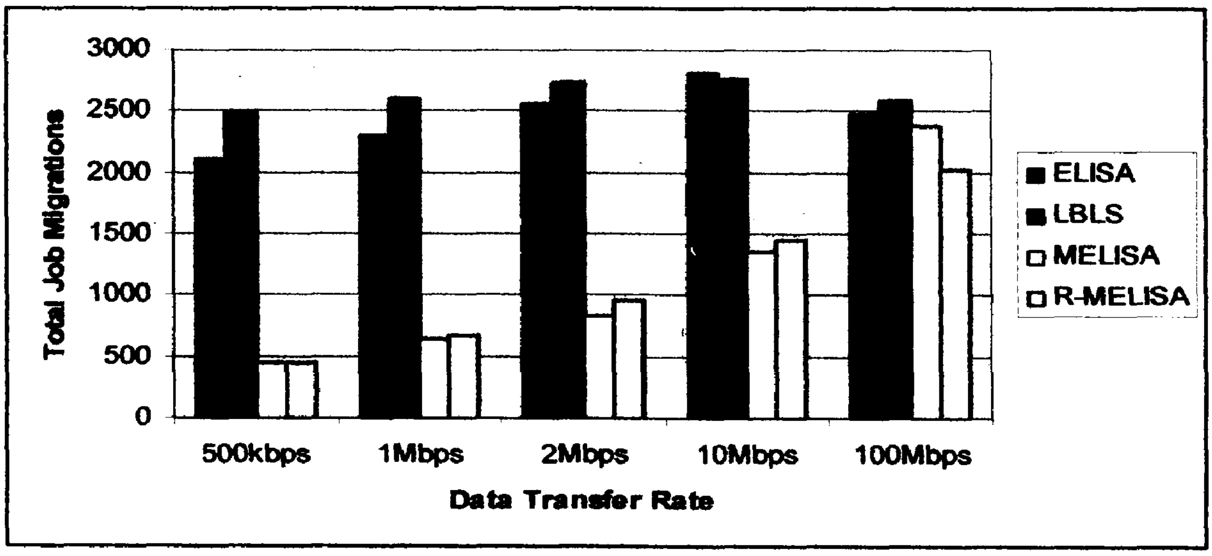
Figure 6.3 Performance measure of MELISA and R-MELISA for varying job size



(a) Effect of data transfer rate on ART



(b) Effect of data transfer rate on Total Execution Time



(c) Effect of data transfer rate on Total Job Migrations

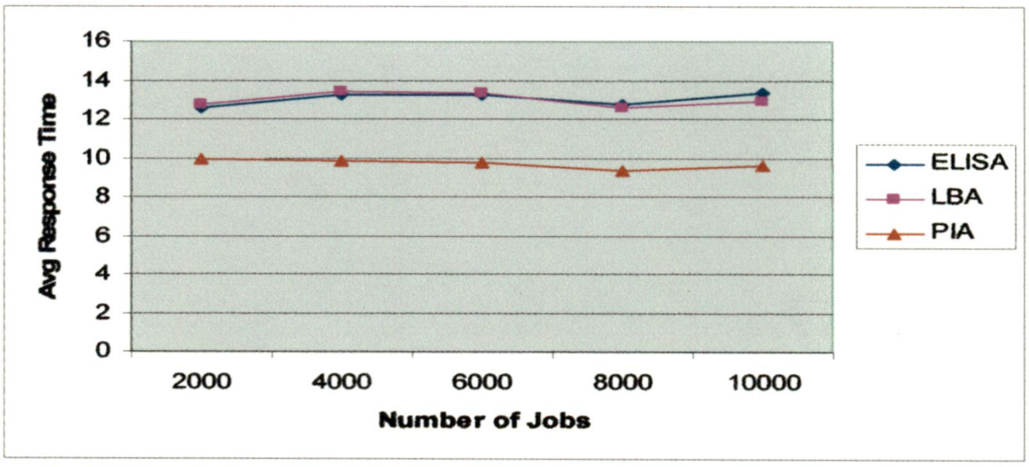
Figure 6.4 Performance measure of MELISA and R-MELISA for varying data transfer rate

6.3 Performance of LBA

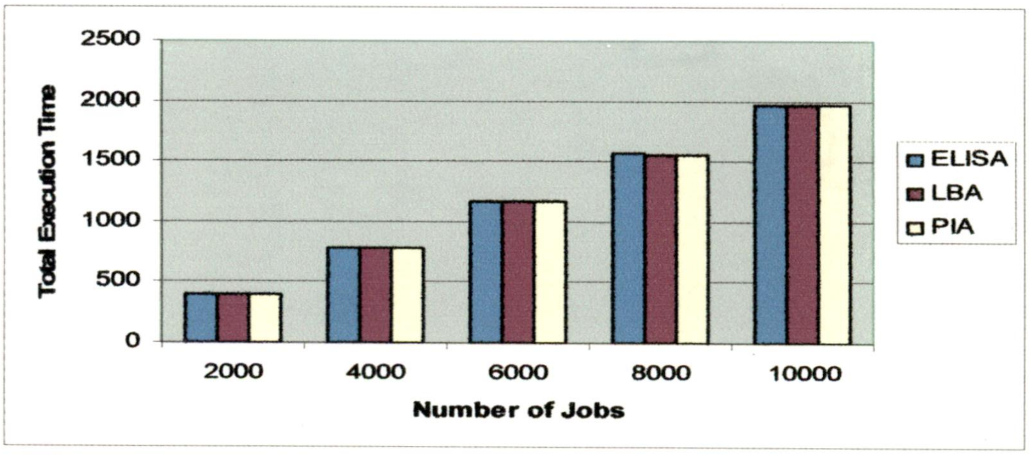
In this section, we will evaluate the performance of our proposed algorithm Load Balancing on Arrival (LBA) with ELISA and PIA. We have considered different cases to measure the performance of LBA. Following subsections describe various cases and performance analysis.

6.3.1 Random arrival and service rates

In this set of experiments, we have quantified the performance of our LBA algorithm for real-life situations wherein arrival rates and service rates are completely random. As seen from Figure 6.5(a), there is not much difference in *ART* for LBA and ELISA, that is, both the algorithms exhibit an increasing tendency as we increase the arrival and service rates. Both algorithms take almost same amount of time for execution of *N* jobs as we can observe it from Figure 6.5(b). As expected, performance of PIA is better than LBA and ELISA as it uses perfect information at the time of load balancing.



(a) ART comparison for random arrival and service rates



(b) Total Execution Time comparison for random arrival and service rates

Figure 6.5 Performance measure of LBA for random arrival and service rates

6.3.2 Effect of status exchange period

ELISA algorithm is highly sensitive to the magnitude of status exchange period T_s. That is, if we set the value of status exchange period to be high, then its performance degrades. For LBA, increasing value of T_s also increases *ART*, but its performance is much better than ELISA. As seen from Figure 6.6, by increasing the value of T_s, there is very high increase in ART for ELISA than for LBA. For PIA, there is no change in *ART* as it uses perfect information about system state at each job migration decision. So for LBA algorithm, by setting the value of status exchange period to be large, we can decrease the number of status exchange messages and communication overheads can be kept at a low value.

6.3.3 Effect of uneven load distribution

One of the major advantages of LBA approach is that it attempts to balance load on each processor "*as soon as possible*". Whenever a job arrives at a processor, that processor will determine whether any of its buddy set members can execute the job earlier than itself. If it finds such a member, then the job will be migrated to that processor. In this way, the load will be balanced as soon as possible. However in ELISA, a job has to wait for next transfer instant before migrating to a lightly loaded processor.

For this set of experiments, we set values of arrival rates and service rates of processors in such a way that load distribution is uneven across all processors. Here (refer Figure 6.1), processors 1, 2, 3 and 4 are highly loaded whereas processors 7,8,15 and 16 are lightly loaded. Other processors are moderately loaded. Results are shown in Figures 6.7(a) and 6.7(b). From the graphs, we observe that LBA gives much better performance for *ART*. Figure 6.7(b) shows the results for minimum utilization of a processor, average utilization of system, and maximum utilization of a processor, after executing all *N* jobs. These values indicate how load has been balanced across processors. From Figure 6.7(b), it may be observed that for ELISA, there is wide variation in terms of processor utilization. In LBA, variation in processor utilization is less than for ELISA. For PIA, there is very little or no variation in processor utilization.

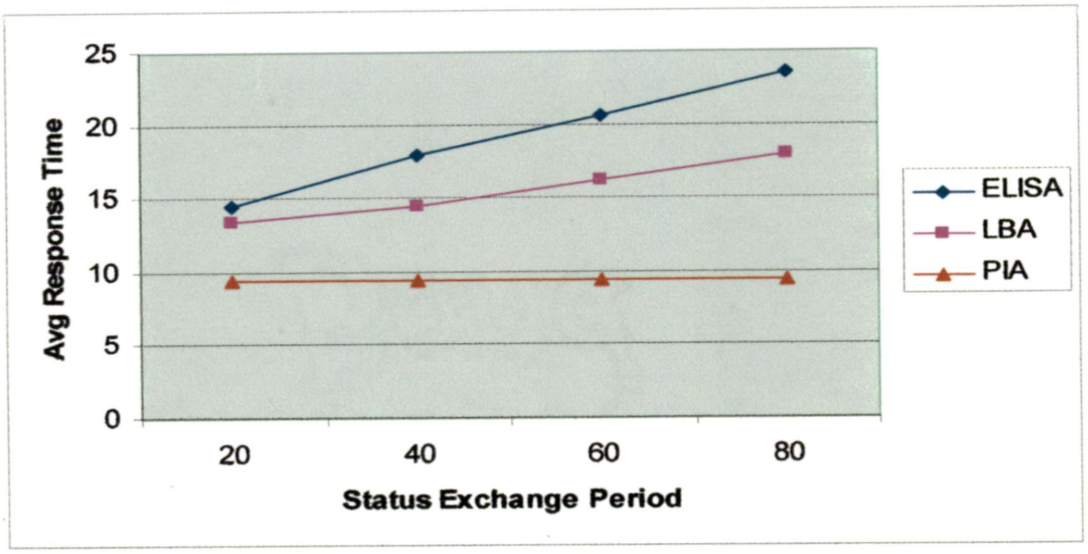
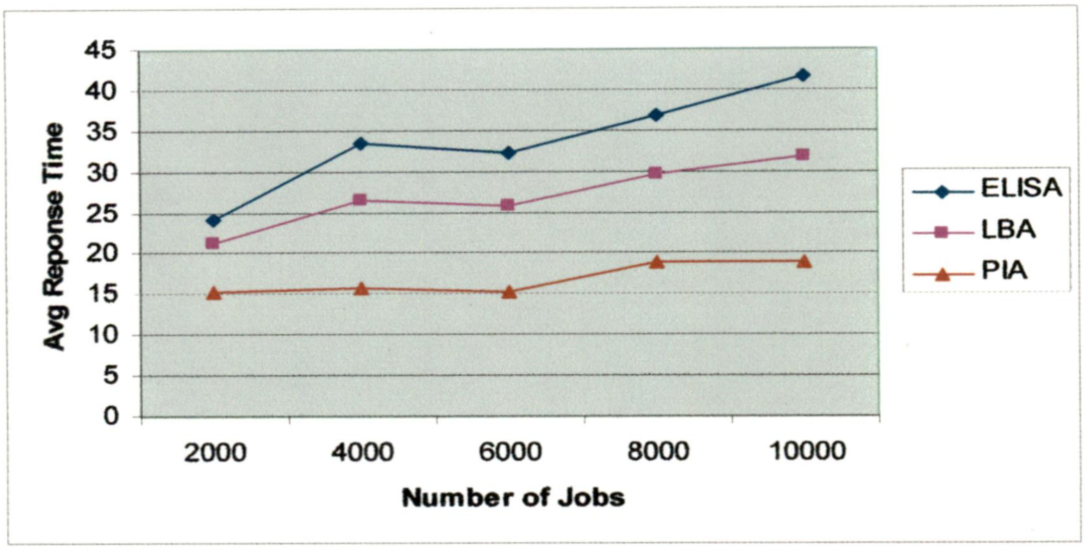
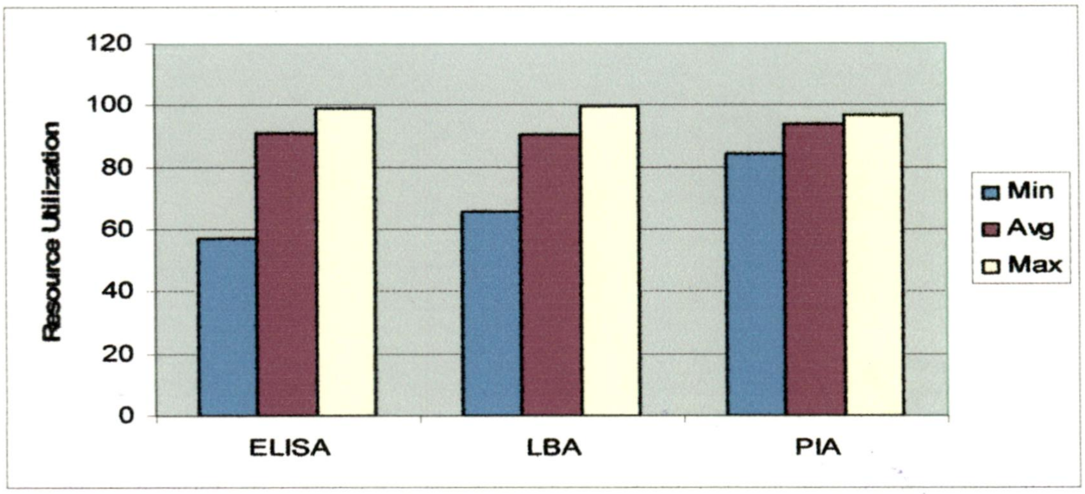


Figure 6.6 Effect of Status exchange period (T_s) on ART



(a) ART comparison for uneven load distribution



(b) Resource Utilization comparison for uneven load distribution

Figure 6.7 Performance measure of LBA for uneven load distribution

6.3.4 Effect of migration limit

One of the important parameters for LBA is *migration limit* (denoted as η), that is, how many hops we should allow a job to migrate, before execution. Obviously, this decision depends on network topology considered. Setting value of η to maximum path length of the graph, we can obtain almost same result when η is very large[4]. By restricting the value of η to a finite value, we can reduce job migration cost by reducing the total number of job migrations.

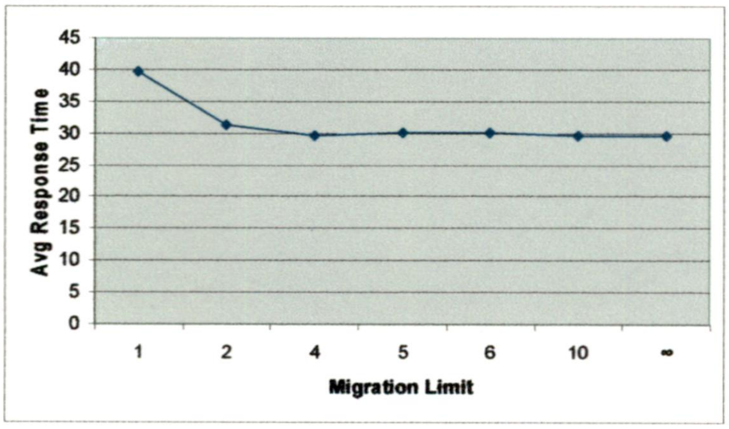
We have used a network topology shown in Figure 6.1 to capture this effect. As from Figure 6.8(a), when $\eta > 4$, there is hardly any change in ART. Also when $\eta = 4$, performance is better for execution time (refer Figure 6.8(b)). Thus we observe that by setting value of η around maximum path length gives a better acceptable performance for LBA.

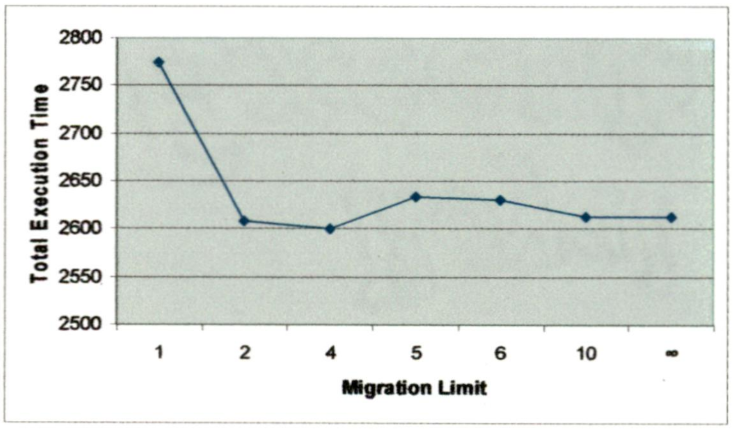
6.3.5 Effect of job size

As from LBA algorithm, job migration cost is also one of the factors for load balancing across its buddy processors. Indeed, we can expect that it should give better performance when we increase the job size. Figures 6.9(a) and 6.9(b) show the results for various job sizes ranging from *5MB±1MB* to *1000MB±200MB*. As it can be observed from Figures 6.9(a) and 6.9(b), for larger job size, performance of LBA is better than ELISA. This is due to the fact that as the job size increases the migration cost is expected to increase which prevents migration in LBA.

Remarks: As seen above, increase in job size would lead to increase in the job migration cost. However, this is handled differently by MELISA and LBA as follows. In MELISA (and R-MELISA), for each transfer instant, P_i will calculate expected finish time (EFT_i^j) of every job j on buddy processor k. If $EFT_k^j < EFT_i^j$, then job will be migrated to buddy processor P_k. In LBA, each P_i will calculate EFT_k^j for job j on buddy processor P_k only on its arrival. If no buddy processor P_k can execute job earlier than P_i, then job will be placed in waiting queue of P_i. In LBA, this job j will be never migrated to any buddy processor and will be executed on P_i, whereas in MELISA, there is a possibility that this job can be migrated if, after some time, there is at least one buddy processor who can execute job earlier than this processor.

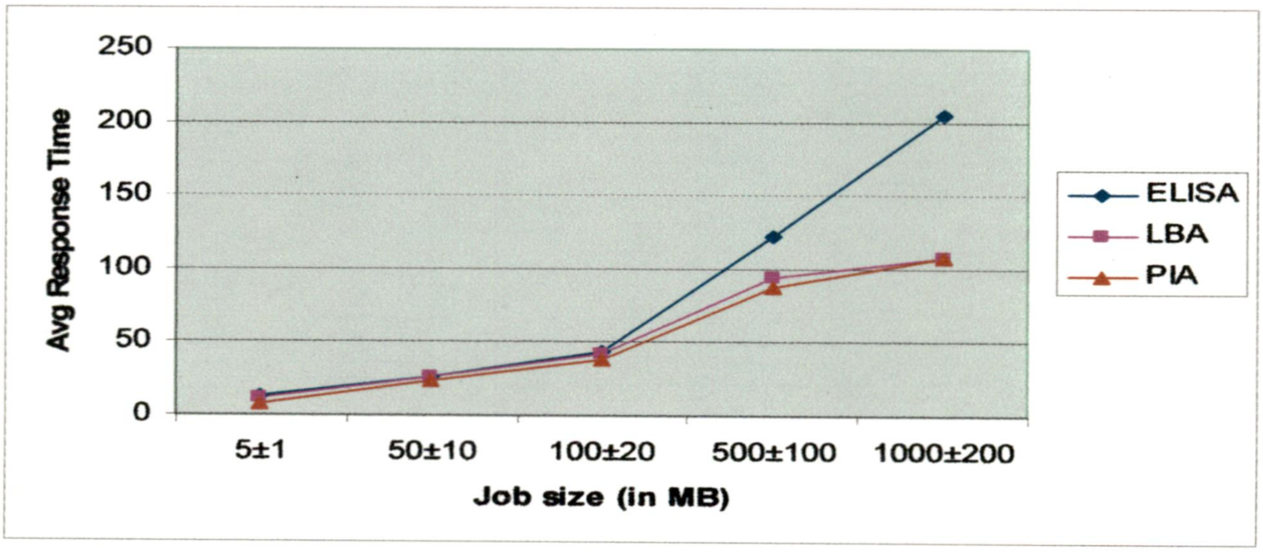
[4] Theoretically setting to infinity.



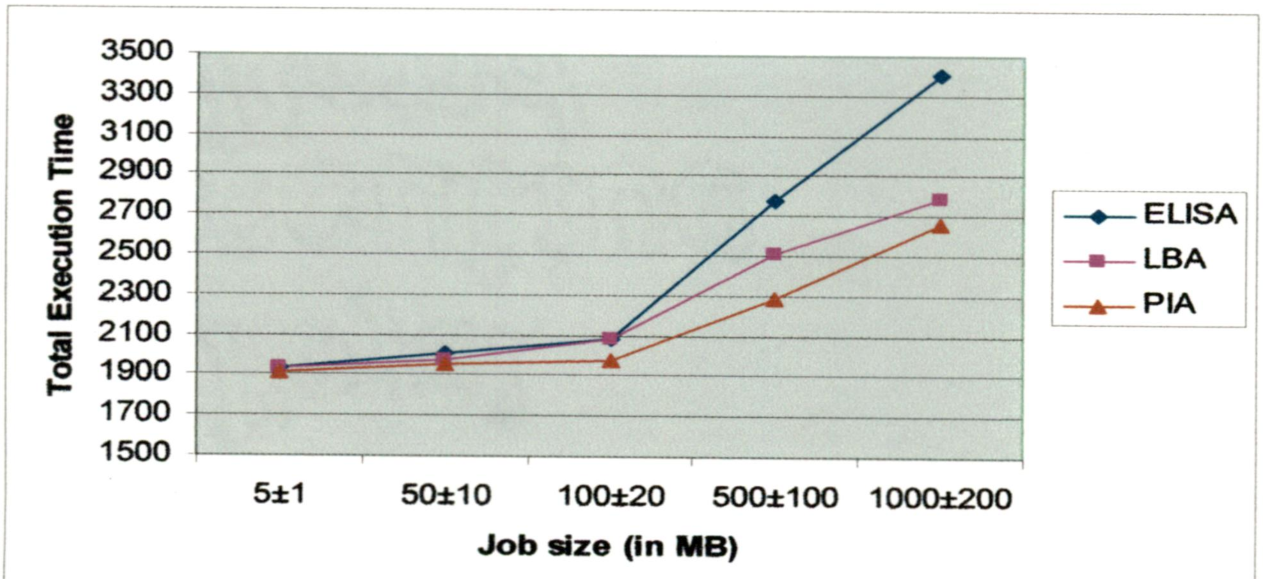


(a) Effect of η on ART (b) Effect of η on Total Execution Time

Figure 6.8 Effect of migration limit (η) for LBA



(a) Effect of job size on ART



(b) Effect of job size on Total Execution Time

Figure 6.9 Performance measure of LBA for varying job size

Chapter 7 Conclusion and Future Work

7.1 Conclusion

In this work, we presented decentralized, scalable, adaptive, and distributed algorithms for load balancing across resources for data intensive computations on Grid environments. The objective is to minimize *ART* and total execution time for jobs that arrive to a Grid system for processing. Several constraints such as communication delays due to underlying network, processing delays at the processors, resource heterogeneity, and an arbitrary topology for the Grid system, are explicitly considered in the problem formulation. Our algorithms are adaptive in the sense that they estimate different types of strongly influencing system parameters such as job arrival rate, processing rate, load on processor and use this information for estimating finish time of job on buddy processor. Through this study, we demonstrate the usefulness and effectiveness of load estimation approach to devise adaptive and dynamic load balancing strategies for data-hungry computational Grid structures.

Our algorithms also consider overheads of job migration due to large communication latency between Grid resources. MELISA and R-MELISA take decision of job migration based on its expected finish time on buddy processor which also includes transfer time of the job. MELISA and R-MELISA give better performance than ELISA for large-scale Grid environments such as InterGrid that is when available bandwidth between two processors is very low and/or job size is large. So MELISA is better suited for large-scale Grid environments. We also observed that there is not much performance difference in MELISA and R-MELISA. Both algorithms give almost same performance for all cases. LBA algorithm performs load balancing on each job arrival by estimating expected finish time on neighboring processor instead of waiting for next transfer instant. Results show that LBA algorithm gives much better performance when load is not evenly balanced across all resources. For small scale Grids such as IntraGrid, LBA is the best solution for load balancing across processors.

Through this study, we also observed the influence of λ (the arrival rates) at the nodes. We noted that by increasing the value of λ_i for P_i, ART of the system also increases. This can be taken care by tuning the estimation time epoch T_e. Thus, by decreasing value of T_e (in case of

(M)ELISA), we can improve the performance of the system as the load is now balanced more frequently. Further, there is a small non-zero probability that a load can shuttle between processors. In our strategies, we prevented this through a control parameter, the migration limit (denoted by η). That is, if a job is migrated for η times, then it will be not transferred again.

7.2 Future Work

This work can be extended in following ways:

- Although we have considered resource heterogeneity in terms of processor speeds, it can be extended to heterogeneity in other dimensions such as, operating system, available storage space, available programming environment etc. Due to this, all jobs will not be eligible to run on all resources and performance of the system will be affected by scheduling and load balancing algorithm.
- Our load balancing algorithms work only for independent jobs. That is, there should not be any dependency among jobs. This can be extended for jobs with dependency. This can be implemented using methods like Task Dependency Graph, Directed Acyclic Graph (DAG), etc.
- In Grid computing systems, scheduler does not have complete control over resources as resources are shared by users and scheduler need to implement user's policy. User policy generally specifies when scheduler can use its resource, how much memory can be used by job, how many jobs are allowed to run at a time etc. Our load balancing algorithms can be enhanced to implement such user policy.
- Fault tolerance is also one of the important characteristics of any distributed systems. Although main aim of this research work is to provide load balancing algorithm, this work can be extended to provide fault tolerance in the Grid systems by using any fault tolerant algorithms available for distributed systems such as heart-beat algorithm.

References

[1] L.Anand, D.Ghose, and V.Mani, "ELISA: An Estimated Load Information Scheduling Algorithm for distributed computing systems", International Journal on Computers and Mathematics with Applications, Vol.37, Issue 8, pp. 57-85, April 1999.

[2] I.Foster, C.Kesselman, and S.Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations", International Journal of High Performance Computing Applications, Vol. 15, Issue 3, pp. 200-222, 2001.

[3] I.Foster and C.Kesselman, "The Grid : Blueprint for a future computing infrastructure", Morgan Kaufmann Publishers, USA, 1999.

[4] L.Smarr and C.E.Catlett, "Metacomputing", Communications of the ACM, Vol. 35, Issue 6, pp. 44-52, June 1992.

[5] H.Shan, L.Oliker, R.Biswas, and W.Smith, "Scheduling in heterogeneous grid environments: The effects of data migration", in Proceedings of ADCOM2004: International Conference on Advanced Computing and Communication, Ahmedabad, Gujarat, India, December, 2004.

[6] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit", International Journal of Supercomputer Applications, Vol. 11, Issue 2, pp. 115-128, 1997.

[7] Globus Project website, http://www.globus.org

[8] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data Management in an International Data Grid Project", Proceedings of the first IEEE/ACM International Workshop on Grid Computing, India, 2000.

[9] GGF's working group on Grid scheduling dictionary, http://www.fzjuelich.de/zam/RD/coop/ggf/sd-wg.html.

[10] Y.Feng, D.Li, H.Wu, and Y.Zhang, "A dynamic load balancing algorithm based on distributed database system", Proceedings 4th International Conference on High Performance Computing in the Asia-Pacific Region,China, pp. 949-952, May 2000.

[11] M.Willebeek-LeMair and A.Reeves, "Strategies for dynamic load balancing on highly parallel computers", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, Issue 4, pp. 979-993, September 1993.

[12] N.Shivaratri, P.Krueger, and M.Singhal, "Load distributing for locally distributed systems", IEEE Computer, Vol. 25, Issue 12, pp. 33-44, December 1992.

[13] H.Lin and C.Raghavendra, "A dynamic load-balancing policy with a central job dispatcher (LBC)", IEEE Transactions on Software Engineering, Vol. 18, Issue 2, pp. 148-158, February 1992.

[14] J.Watts and S.Taylor, "A practical approach to dynamic load balancing", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, pp. 235-248, March 1998.

[15] G.Manimaran and C.Siva Ram Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, Issue 3, pp. 312-319, March 1998.

[16] M.J.Zaki and W.Li.S.Parthasarathy, "Customized dynamic load balancing for a network of workstations", Journal of Parallel and Distributed Computing, Vol. 43, Issue 2, pp. 156-162, June 1997.

[17] J.Krallmann, U.Schwiegelshohn, and R.Yahyapour, "On the design and evaluation of job scheduling algorithms", In 5th Workshop on Job Scheduling Strategies for Parallel Processing, Vol. LNCS 1659, pp. 17-42, 1999.

[18] D.G.Feitelson, L.Rudolph, U.Schwiegelshohn, K.C.Sevcik, and P.Wong, "Theory and practice in parallel job scheduling", In 3rd Workshop on Job Scheduling Strategies for parallel processing, Vol. LNCS 1291, pp. 1-34, 1997.

[19] Y.Murata, H.Takizawa, T.Inaba, and H.Kobayashi, "A distributed and cooperative load balancing mechanism for large-scale P2P systems", International Symposium on Applications and Internet Workshops (SAINT 06), pp. 126-129, Jan 2006.

[20] L.Oliker, R.Biswas, H.Shan, and W.Smith, "Job scheduling in heterogeneous grid environment", Lawrence Berkeley National Laboratory, LBNL-54906, 2004.

[21] Z.Zeng and B.Veeravalli, "Design and analysis of a non-preemptive decentralized load balancing algorithm for multi-class jobs in distributed networks", Computer Communications, Vol. 27, pp. 679-693, 2004.

[22] V.Subramani, R.Kettimuthu, S.Srinivasan, and P.Sadayappan, "Distributed job scheduling on computational grid using multiple simultaneous requests", Proceedings of 11th IEEE Symposium on High Performance Distributed Computing (HPDC 2002), July 2002.

[23] M.Arora, S.K.Das, and R.Biswas, "A De-centralized scheduling and load balancing algorithm for heterogeneous grid environments", Proceedings of the International Conference on Parallel Processing Workshops (ICPPW,2002), pp. 499-505, 2002.

[24] H.A.James and K.A.Hawick, "Scheduling independent tasks on metacomputing systems", Proceedings ISCA 12th International Conference on Parallel and Distributed Computing Systems (PDCS-99), Fort Lauderdale, USA, March 1999.

[25] M.Mitzenmacher, "How useful is old information?", IEEE Transactions on Parallel and Distributed Systems, Vol. 11, Issue 1, pp. 6-20, 2000.

[26] R.Martin, A.Vahdat, D.Culler, and T.Anderson, "Effects of communication latency, overhead and bandwidth in a cluster architecture", Proceedings 24th Annual International Symposium on Computer Architecture, pp. 85-97, June 1997.

List of submitted papers

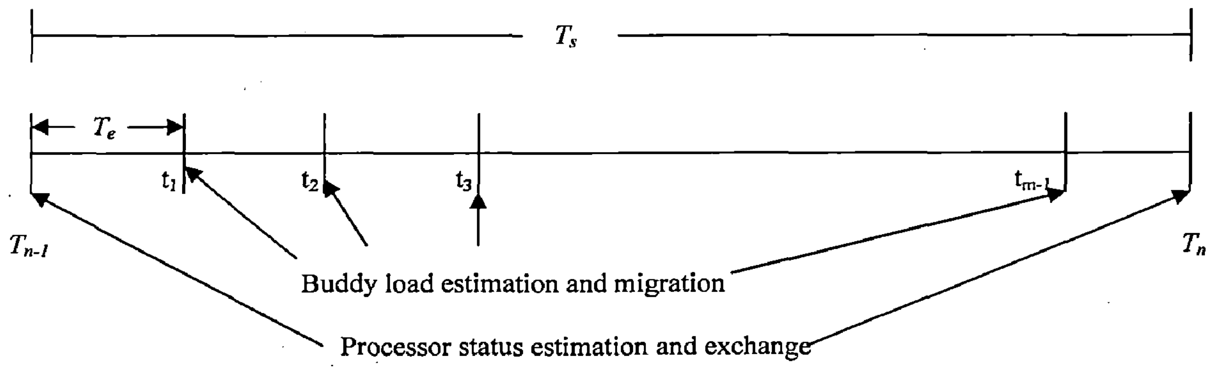
(1) "On the Design of Adaptive and De-centralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments", submitted in IEEE Transactions on Parallel and Distributed Systems (TPDS).

(2) "Estimation Based Load Balancing Algorithm for Data-Intensive Heterogeneous Grid Environments", submitted in 13th IEEE International Conference on High Performance Computing (HiPC 2006).

(3) "A Receiver-initiated Load Balancing Algorithm with Parameter Estimation for Heterogeneous Grid Environments", submitted in 2nd International Conference on Semantics, Knowledge and Grid (SKG-2006).

Appendix A : ELISA (Estimated Load Information Scheduling Algorithm)

In ELISA, as shown in Figure A.1, the load is estimated at every estimation period T_e using status information (queue length, estimate of arrival rate) exchanged at status exchange period T_s. The working of ELISA can be summarized as shown in Figure A.2:



T_s
T_e
t_1
t_2
t_3
t_{m-1}
T_{n-1}
T_n
Buddy load estimation and migration
Processor status estimation and exchange

Figure A.1 Estimation and status exchange intervals in ELISA

```
Main Algorithm

At the status exchange epoch, for each processor:
1. Estimate arrival rate by averaging the number of arrivals over the
   previous n status exchange intervals.
2. Communicate status (queue length and estimate of arrival rate) to all
   processors in the buddy set;
3. Call Transfer.

At the estimation epoch, for each processor:
1. Estimate the queue length for each processor in the buddy set.
2. Call Transfer.
```

```
Procedure Transfer (Computation of transfer possibilities and
transfer of jobs.)

1. Find average queue length of the processor in the buddy set.
2. If the queue length of a processor is greater than the average queue
   length (as computed in 1), then:
   a) Construct the active set as follows: if a processor in the buddy set
      has a queue length less than the average queue length, include the
      processor in the active set.
   b) Compute the probability of transferring from the processor
      (source) to each processor (destination) in the active set such that
      the source processor load in excess of average queue length is
      distributed among processors of active set.
3. Transfer the jobs as per the probabilities computer in 2(b).
```

Figure A.2 Working of ELISA

```
**********************************************************************************
   File : Definition.h
   Description : This file contains general declaration of a program.
**********************************************************************************
/* Following two variables describe resource status*/
#define BUSY 1
#define IDLE 0

/*Following two variables are weighting factor for estimating arrival rate and service rate*/
#define ALPHA 0.5
#define BETA  0.5

/*TIME_PERIOD will indicates when information will be exchanged among buddy set*/
#define TIME_PERIOD 20
#define EST_PERIOD 5

/* NUMBER_OF_PROCESSORS indicates how many processors are there in system*/
#define NUMBER_OF_PROCESSORS 16

/*number of jobs to be processed*/
#define MAX_JOBS 10000

/*Following two variables are for determining program size having normal distribution*/
#define PROGRAM_SIZE_MEAN 4.0*1024*1024*8
#define PROGRAM_SIZE_VARIANCE 2.0*1024*1024*8

/*Following variable is still not used, but can be used to indiacate threshold value*/
#define THRESHOLD 0
#define THRESHOLD1 100

/*This indicates number of migration allowed for a job*/
#define MIGRATION_LIMIT 4

/*RANDOM_SEED is used to produce different random variable*/
#define RANDOM_SEED 81

/*ACCURACY is used for determining probability*/
#define ACCURACY 0.98

/*commCost indicated transfer rate between two processor in system*/
double commCost1 = 10.0*1024.0*1024;

#define SPEED_FACTOR 0.0
/*This function will generate random number between 0 and 1*/
double getUniformRandomVariable()
{
   return (((double)rand() + 1)/ ((double)(RAND_MAX)+(double)(1)));
}

/*This function will generate exponential random varialbe given meanTime as beta*/
double getExponentialRandomVariable(double meanTime)
```

```
{
    return -meanTime * log(getUniformRandomVariable());
}

/*This function will generate normal random variable given mean and variance*/
double getNormalRandomVariable(double mean,double variance)
{
    return mean + variance*getUniformRandomVariable();
}

/*This fuction will return factorial of given number k*/
int factorial(int k)
{
  int prod = 1;
  for(int i = 2; i <=k; i++)
       prod *= i;
  return prod;
}

/*This function will return m^n, here n must be integer*/
double power(double m, int n)
{
    double prod = m;
    for(int i = 2; i <= n; i++)
         prod *= m;
    return prod;
}


*********************************************************************************
  File : Task.h
  Description : This file contains class definition for task. For each job that need to executed in
  system, object of class Task has to be created.
*********************************************************************************
/* Class definition for Task*/
class Task
{
    int jobID;                  //Indicates job ID
    double actualArrivalTime;   //Indicates actual arrival time
    double arrivalTime;         //Indicates arrival time can be migration time
    double waitingTime;         //Indicates waiting time for a job in system
    double executionTime;       //Indicates execution time for this job
    double taskSize;            //Indicates job size for this job
    int assignedNodeID;         //Indicates to which processor this job is assigned
    double expectedExecutionTime;//Indicates expected execution time of a job
    double expectedFinishTime;          //Indicates when job is likely to finish
    int noOfTimesMigrated;      //Indicated how many times job is migrated

public:
     Task *next;            //Used to maintain linked list of Task objects

     /*Following constructor initialize different variables of Task object*/
     Task(int id, double aTime, double eTime, double tSize, int nID, double expTime)
     {
          jobID = id;
    actualArrivalTime = aTime;
    arrivalTime = aTime;
    waitingTime = 0;
    executionTime = eTime;
    expectedExecutionTime = expTime;
    taskSize = tSize;
    assignedNodeID = nID;
    expectedFinishTime = -1;
    noOfTimesMigrated = 0;
    next = NULL;                //NULL indicates zero
}

/*This is a copy constuctor which performs same task as done by previous constructor.
 This constructor initialize variables though object j*/
Task(Task &j)
{
     jobID = j.getJobID();
     actualArrivalTime = j.getActualArrivalTime();
     arrivalTime = j.getArrivalTime();
     waitingTime = j.getWaitingTime();
     executionTime = j.getExecutionTime();
     expectedExecutionTime = j.getExpectedExecutionTime();
     taskSize = j.getTaskSize();
     assignedNodeID = j.getAssignedNodeID();
     expectedFinishTime = j.getExpectedFinishTime();
     noOfTimesMigrated = j.getNoOfTimesMigrated();
     next = j.getNextTask();
}

/*This function will return job id of this object*/
int getJobID()
{
  return jobID;
}

/*This function will return processor ID to which this job is assigned*/
int getAssignedNodeID()
{
   return assignedNodeID;
}

/*This function will return arrival time of this job. This time can be migration time also*/
double getArrivalTime()
{
    return arrivalTime;
}

/*This function will set arrival time of job*/
void setArrivalTime(double time)
{
   arrivalTime = time;
}

/*This function will return actual arrival time of job in the system*/
double getActualArrivalTime()
```

```
{
    return actualArrivalTime;
}

/*This function will set processor ID for this job. This function will be used when job will
migrated to some other processor. We need to call this funtion to set new processor id*/
void setAssignedNodeId(int id)
{
  assignedNodeID = id;
}

/*This function will set waiting time for a job in the system*/
void setWaitingTime(double time)
{
   waitingTime = time;
}

/*This function will return waiting time of a job*/
double getWaitingTime()
{
    return waitingTime;
}

/*This function will set execution time for job*/
void setExecutionTime(double time)
{
   executionTime = time;
}

/*This function will return execution time of this job*/
double getExecutionTime()
{
    return executionTime;
}

void setExpectedExecutionTime(double val)
{
   expectedExecutionTime = val;
}

double getExpectedExecutionTime()
{
    return expectedExecutionTime;
}

/*This function will set size of this job*/
void setTaskSize(double tSize)
{
   taskSize = tSize;
}

/*This function will return size of this job*/
double getTaskSize()
{
   return taskSize;
```

```
    }

    /*This function will set expected finish time for this job for  assignedNodeId processor*/
    void setExpectedFinishTime(double time)
    {
       expectedFinishTime = time;
    }

    /*This function will return expected finish time for this job for processor assignedNodeId*/
    double getExpectedFinishTime()
    {
        return expectedFinishTime;
    }

    /*This function will increment variable noOfTimesMigrated. This need to be called every time
      job is    migrated.*/
    void incrementNoOfTimesMigrated()
    {
       noOfTimesMigrated = noOfTimesMigrated + 1;
    }

    /*This function will return how many times this job is migrated*/
    int getNoOfTimesMigrated()
    {
       return noOfTimesMigrated;
    }

    /*Following function is used for linked list This will return next task in queue.*/
    Task * getNextTask()
    {
       return next;
    }
};

*************************************************************************************
  File : Node.h
  Description : This file contains class definition for Processor. We need to create as many objects of
  this class as there are processors in the system.
*************************************************************************************
/*Class node definition*/
class Node
{
   int ID;                         // processor ID
   int status;                     // indicates status of processor
   int buddySetCount;              // indicates how many processors are there in buddy set
   double speedFactor;             // indicates relative speed of a processor
   int *buddySet;                  // this array will contain id of buddy processor
   double *commSpeed;              // this array will contain comm speed between two buddy processors
   double *estBuddyArrivalTime; // this array will contain estimated arrival time of buddy processor
   double *estBuddyServiceTime;   // this array will contain estimated service time of buddy
  processor
   double *estBuddyLoad;           // this array will contain estimated load of buddy processor
   double nextArrival;             // indicates when next job arrival event will take place
   double nextDeparture;           // indicates when next job departure event will take place
```

```
    double nextMigration;       // indicates when next job migration will take place
    double lastEventTime;       // indicates time for last event
    double totalIdleTime;       // indicates total idle time for this processor
    double lastIdleTimePeriod;  // indicates how much time processor remain idle in last period
                                // this varaible is used for determining service rate of processor
    int jobsWaitingInQueue;     // indicates how many jobs are waiting in queue
    int noOfArrivalInPeriod;    // indicates how many jobs arrive in time period
                                // this is used to determine arrival rate
    int noOfDepartureInPeriod;  // indicates how many departure takes place in time period
    double estMeanArrivalTime;  // indicates estimated mean arrival time of job
    double estMeanServiceTime;  // indicates estimated mean service time of job
    double estCurrentLoad;      // indicates estimated load of processor
    Task *head,*tail,*headMigration; // variables used for implementing waiting queue using linked
       list

public:
    /*Constructor used to initialize variables*/
    Node(int id = 0)
    {
        ID = id;
        status = IDLE;              //initially processor is idle
        totalIdleTime = 0;
        lastIdleTimePeriod = 0;
        jobsWaitingInQueue = 0;
        estMeanArrivalTime = 0.0;
        estMeanServiceTime = 0.0;
        nextArrival = INFINITY;         //INFINITY = we do not when next arrival will take place
        nextDeparture = INFINITY;
        nextMigration = INFINITY;
        lastEventTime = 0;
        noOfArrivalInPeriod = 0;
        noOfDepartureInPeriod = 0;
        head = NULL;
        tail = NULL;
        headMigration = NULL;
    }

    /*This function will set buddySetCount and buddy ids for this processor*/
    void setBuddySet(int id, int count, int *set, double *speed)
    {
      ID = id;
      buddySetCount = count;
      buddySet = new int[buddySetCount];
      commSpeed = new double[buddySetCount];
      estBuddyLoad = new double[buddySetCount];
      estBuddyArrivalTime = new double[buddySetCount];
      estBuddyServiceTime = new double[buddySetCount];
      /*Following loop will set buddy id for this processor*/
      for(int i = 0; i < buddySetCount; i++)
      {
         buddySet[i] = set[i];
         commSpeed[i] = speed[i];
         estBuddyLoad[i] = 0.0;
         estBuddyArrivalTime[i] = 0.0;
         estBuddyServiceTime[i] = 0.0;
    }
}

void setSpeedFactor(double value)
{
    speedFactor = value;
}

double getSpeedFactor()
{
     return speedFactor;
}

/*Fuction will return number of buddy processors*/
int getBuddySetCount()
{
   return buddySetCount;
}

/*Function will return buddy id for given index count*/
int getBuddyId(int count)
{
   return buddySet[count];
}

/*Function will set node id for this processor*/
void setNodeID(int id)
{
    ID = id;
}

/*This function will return id for this processor*/
int getNodeID()
{
   return ID;
}

/*This will set status of this processor,either BUSY or IDLE*/
void setNodeStatus(int s)
{
    status = s;
}

/*This function will return status of this processor*/
int getNodeStatus()
{
   return status;
}

/*This function will increment node idle time by amount value*/
void incrementNodeIdleTime(double amount)
{
    totalIdleTime += amount;
}
```

```
/*This function will return node idle time*/
double getNodeIdleTime()
{
  return totalIdleTime;
}

/*This function will return idle time for last period*/
double getLastIdleTimePeriod()
{
     return lastIdleTimePeriod;
}

/*This function will increment idle time by time value for last time period*/
void incrementLastIdleTimePeriod(double time)
{
   lastIdleTimePeriod += time;
}

/*This function will reset variable lastIdleTimePeriod.
  This is necessary once we have calculated mean service time for given period*/
void resetLastIdleTimePeriod()
{
   lastIdleTimePeriod = 0;
}

/*This function will set next arrival event for this processor*/
void setNextArrivalEvent(double time)
{
   nextArrival = time;
}

/*This function will return next arrival event time*/
double getNextArrivalEvent()
{
     return nextArrival;
}
/*This function will set next departure event time for this processor*/
void setNextDepartureEvent(double time)
{
   nextDeparture = time;
}

/*This function will return next departure event time*/
double getNextDepartureEvent()
{
     return nextDeparture;
}

/*This function will set next migration event time*/
void setNextMigrationEvent(double time)
{
   nextMigration = time;
}

/*This function will return next migration event time*/
```

```
double getNextMigrationEvent()
{
   return nextMigration;
}

/*This will set last event time*/
void setLastEventTime(double time)
{
   lastEventTime = time;
}

/*This will return last event time.*/
double getLastEventTime()
{
     return lastEventTime;
}

/*Increment or decrement number of jobs waiting in queue.*/
void setJobsWaitingInQueue(int amount)
{
   jobsWaitingInQueue += amount;
}

/*This will return number of jobs waiting in queue*/
int getJobsWaitingInQueue()
{
  return jobsWaitingInQueue;
}

/*This function will return estimated mean arrival time of jobs for this processor*/
double getEstMeanArrivalTime()
{
     return estMeanArrivalTime;
}

/*This function will set estimated mean arrival time of jobs for this processor*/
void setEstMeanArrivalTime(double time)
{
   estMeanArrivalTime = time;
}

 /*This function will return estimated mean service time of jobs for this processor*/
double getEstMeanServiceTime()
{
     return estMeanServiceTime;
}

/*This function will set estimated mean service time of jobs for this processor*/
void setEstMeanServiceTime(double time)
{
   estMeanServiceTime = time;
}

/*This function will return estimated current load of this processor*/
double getEstCurrentLoad()
```

```
{
    return estCurrentLoad;
}

/* This function will set estimated current load to amount value*/
void setEstCurrentLoad(double amount)
{
   estCurrentLoad = amount;
}
/*This will increment variable noOfArrivalInPeriod when new job arrives for this processor*/
void incrementNoOfArrivalInPeriod()
{
   noOfArrivalInPeriod = noOfArrivalInPeriod + 1;
}

/*This will reset variable noOfArrivalInPeriod.
  Call this function once mean arrival time has been calculated for given time period*/
void resetNoOfArrivalInPeriod()
{
   noOfArrivalInPeriod = 0;
}
/*This will increment variable noOfDepartureInPeriod when job departures from this processor*/
void incrementNoOfDepartureInPeriod()
{
   noOfDepartureInPeriod = noOfDepartureInPeriod + 1;
}

/*This will reset variable noOfDepartureInPeriod.
  Call this function once mean service time has been calculated for given time period*/
void resetNoOfDepartureInPeriod()
{
   noOfDepartureInPeriod = 0;
}

/*This function will calculate estimated mean arrival time for given period*/
void calculateMeanArrivalTime()
{
   double meanArrivalTime = TIME_PERIOD;
   /*Check whether there is at least one arrival in last interval*/
   if(noOfArrivalInPeriod != 0)
      meanArrivalTime = TIME_PERIOD / (double)noOfArrivalInPeriod;
   estMeanArrivalTime = ALPHA * estMeanArrivalTime + (1-ALPHA) * meanArrivalTime;
   /*Reset variable noOfArrivalInPeriod*/
   resetNoOfArrivalInPeriod();
}

/*This function will calculate estimated mean serivce time in last interval*/
void calculateMeanServiceTime()
{
   /*meanServiceTime indicates total busy time for this processor*/
   double meanServiceTime = TIME_PERIOD - getLastIdleTimePeriod();
   /*If no of departure > 0,then divide meanServiceTime by total departure in last interval*/
   if(noOfDepartureInPeriod != 0)
      meanServiceTime = meanServiceTime / (double) noOfDepartureInPeriod;
   estMeanServiceTime = BETA * estMeanServiceTime + (1-BETA) * meanServiceTime;
```

```
    /*Reset no of departure and last idle time period*/
    resetNoOfDepartureInPeriod();
    resetLastIdleTimePeriod();
}

/*This function will calculate estimated finish time of job based on estimated service time for P_i*/
void calculateExpectedFinishTimeofTasks(double currentSystemTime)
{
    /*If no job to process, then return*/
    if(head == NULL)
        return;
    Task *p = head;
    /*Set expected finish time for job which is running on processor*/
    double time = currentSystemTime - getLastEventTime();
    time = estMeanServiceTime - time;
    p->setExpectedFinishTime(time);
    /*Following loop will set expected finish time of jobs waiting in queue*/
    while(p->next != NULL)
    {
        p = p->next;
        time += estMeanServiceTime;
        p->setExpectedFinishTime(time);
    }
}

/*This function will set exact finish time of jobs waiting in queue. This function will use
execution time of job directly. This is used when we used Perfect Information Algorithm.*/
void calculateExactFinishTimeofTasks(double currentSystemTime)
{
    if(head == NULL)
        return;
    Task *p = head;
    double time = currentSystemTime - getLastEventTime();
    time = p->getExecutionTime() - time;
    p->setExpectedFinishTime(time);
    while(p->next != NULL)
    {
        p = p->next;
        time += p->getExecutionTime();
        p->setExpectedFinishTime(time);
    }
}

/*This function will set variable estCurrentLoad to current queue length*/
void calculateEstCurrentLoad(double currentSystemTime)
{
   estCurrentLoad = getJobsWaitingInQueue();
}

void calculateEstCurrentLoadForNextPeriod(double currentSystemTime,double time)
{
  double factor = time / estMeanArrivalTime;
  double ePower = exp((-1) * factor);
  double prob = ePower;
  int arrival = 1;
  while(prob < ACCURACY)
  {
      prob += ((ePower * power(factor,arrival))/factorial(arrival));
      arrival++;
  }
  factor = time / estMeanServiceTime;
  ePower = exp((-1) * factor);
  prob = ePower;
  int departure = 1;
  while(prob < ACCURACY )
  {
      prob += ((ePower * power(factor,departure)) / factorial(departure));
      departure++;
  }
  /* Add differece of arrival and departure to previous load value*/
  estCurrentLoad = getJobsWaitingInQueue() + (arrival - departure);
  estCurrentLoad *= estMeanServiceTime;
  if(estCurrentLoad < 0)
      estCurrentLoad = 0;
}

/*This functiion will calculate estimated load on buddy processor. This load will be calculated for
time value*/
void calculateEstBuddyLoad(double time)
{
   /*Loop for all buddy processors*/
   for(int count = 0; count < buddySetCount; count++)
   {
    /*Find out number of arrival and departured in time*/
    double factor = time / estBuddyArrivalTime[count];
    double ePower = exp((-1) * factor);
    double prob = ePower;
    int arrival = 1;
     while(prob < ACCURACY)
     {
        prob += ((ePower * power(factor,arrival))/factorial(arrival));
        arrival++;
     }
    factor = time / estBuddyServiceTime[count];
    ePower = exp((-1) * factor);
    prob = ePower;
    int departure = 1;
    while(prob < ACCURACY )
    {
        prob += ((ePower * power(factor,departure)) / factorial(departure));
        departure++;
    }
    /* Add differece of arrival and departure to previous load value*/
    estBuddyLoad[count] += (arrival - departure);
    if(estBuddyLoad[count] < 0)
              estBuddyLoad[count] = 0;

   }
}
```

```
/*This function will calculate load on buddy processor in terms of execution time required*/
void calculateEstBuddyLoad1(double time)
{
   /*Loop for all buddy processors*/
   for(int count = 0; count < buddySetCount; count++)
   {
    double factor = time / estBuddyArrivalTime[count];
    double ePower = exp((-1) * factor);
    double prob = ePower;
    int arrival = 1;
    while(prob < ACCURACY)
    {
        prob += ((ePower * power(factor,arrival))/factorial(arrival));
        arrival++;
    }
    factor = time / estBuddyServiceTime[count];
    ePower = exp((-1) * factor);
    prob = ePower;
    int departure = 1;
    while(prob < ACCURACY )
    {
        prob += ((ePower * power(factor,departure)) / factorial(departure));
        departure++;
    }
    /*Find out total time required by new arrivals*/
    estBuddyLoad[count] += ((arrival - departure) * estBuddyServiceTime[count]);
    if(estBuddyLoad[count] < 0)
            estBuddyLoad[count] = 0;

   }
}

/*This function will return estimated buddy load of buddy processor.
Here count indicates index value for estBuddyLoad array*/
double getEstBuddyLoad(int count)
{
   return estBuddyLoad[count];
}

/*This function will return estimated arrival time of buddy processor.
Here count indicates index value for estBuddyArrivalTime array*/
double getEstBuddyArrivalTime(int count)
{
   return estBuddyArrivalTime[count];
}

/*This function will return estimated service time of buddy processor.
Here count indicates index value for estBuddyServiceTime array*/
double getEstBuddyServiceTime(int count)
{
   return estBuddyServiceTime[count];
}

/*This function will set estimated buddy load of buddy processor to load value.
```

```
Here count indicated index value for estBuddyLoad array*/
void setEstBuddyLoad(int count, double load)
{
   estBuddyLoad[count] = load;
}

/*This function will set estimated buddy arrival time of buddy processor to time value.
Here count indicated index value for estBuddyArrivalTime array*/
void setEstBuddyArrivalTime(int count, double time)
{
   estBuddyArrivalTime[count] = time;
}

/*This function will set estimated buddy service time of buddy processor to time value.
Here count indicated index value for estBuddyServiceTime array*/
void setEstBuddyServiceTime(int count, double time)
{
   estBuddyServiceTime[count] = time;
}

/*This function will increment buddy processor's load by 1. Here id indicates ID of buddy
processor. This function should be called when load is in terms of queue length*/
void incrementEstBuddyLoad(int id)
{
   int count = 0;
   for(count = 0; count < buddySetCount; count++)
   {
        if(id == buddySet[count])
        {
            estBuddyLoad[count]++;
            break;
        }
   }
   if(count == buddySetCount)
         cout << "No match\n";
}

/*This function will increment buddy processor's load by amount value.
This function should be called when load is in terms of time*/
void incrementEstBuddyLoad(int id,double amount)
{
   int count = 0;
   for(count = 0; count < buddySetCount; count++)
   {
        if(id == buddySet[count])
        {
            estBuddyLoad[count] += amount;
            break;
        }
   }
   if(count == buddySetCount)
         cout << "No match\n";
}

/*This function will calculate estimated current load for amount time*/
```

```
void calculateEstimateCurrentLoad(double time)
{
    double factor = time / getEstMeanArrivalTime();
    double ePower = exp((-1) * factor);
    double prob = ePower;
    int arrival = 1;
    while(prob < ACCURACY)
    {
        prob += ((ePower * power(factor,arrival))/factorial(arrival));
        arrival++;
    }
    factor = time / getEstMeanServiceTime();
    ePower = exp((-1) * factor);
    prob = ePower;
    int departure = 1;
    while(prob < ACCURACY )
    {
        prob += ((ePower * power(factor,departure)) / factorial(departure));
        departure++;
    }
    estCurrentLoad += (arrival - departure);
    if(estCurrentLoad < 0)
            estCurrentLoad = 0;
}

/*This function will calculate exact current load using job's exact execution time*/
void calculateExactCurrentLoad(double currentSystemTime)
{
   /*If head = NULL, then there is no job in queue*/
   if(head == NULL)
      estCurrentLoad = 0;
   else
   {
     /*Loop for all jobs waiting in queue*/
    Task *p = head;
    double time = currentSystemTime - getLastEventTime();
    time = p->getExecutionTime() - time;
    while(p->next != NULL)
    {
        p = p->next;
        time += p->getExecutionTime();
    }
    estCurrentLoad = time;
   }
}

/*This function will calculate exact current load at time currentSystemTime + transferTime.
This function can be used for Perfect Information algorithm.*/
void calculateExactCurrentLoad(double currentSystemTime, double transferTime)
{
   if(head == NULL)
      estCurrentLoad = 0;
   else
   {
   /*Find total execution time for jobs waiting in queue*/
```

```
      Task *p = head;
      double time = currentSystemTime - getLastEventTime();
      time = p->getExecutionTime() - time;
      while(p->next != NULL)
      {
           p = p->next;
           time += p->getExecutionTime();
      }
      estCurrentLoad = time;
     }
     if(headMigration != NULL)
     {
     /*Add execution time for those jobs which will migrated to this processor
      before transfer time */
      Task *p = headMigration;
      while(p!=NULL)
      {
       if(p->getArrivalTime() < (currentSystemTime + transferTime))
       {
          estCurrentLoad += p->getExecutionTime();
          p=p->next;
       }
       else
          break;
       }
     }
}

/*This function will return exact current load for this processor*/
double getExactCurrentLoad()
{
     return estCurrentLoad;
}

/*When new job arrives to the processor, this function will be called. This function will add new
job at tail end to maintain queue*/
void addNewTask(Task j)
{
    Task *p = new Task(j);
    p->next = NULL;
    /*If there is no job in queue*/
    if(head == NULL)
    {
         /*set both head and tail pointer here*/
         head = p;
         tail = p;
    }
    else
    {
      /*Set tail pointer to new job p as new job will be added at last*/
      tail->next = p;
      tail = p;
    }
}
```

```
/*This function will return first job in FIFO order*/
Task *getNextTaskInQueue()
{
   /*If head is NULL, that means there is no job to process so return NULL*/
  if(head == NULL)
       return NULL;
  /*Return first job in queue*/
/*   Task *p = head;
  Task *q=NULL;
  while(p!=NULL)
  {
  if(p->getJobID() <= MAX_JOBS)
  {
       if(q != NULL)
       {
          q->next = p->next;
          if(q->next == NULL)
                 tail = q;
          p->next = head;
          head = p;
       }
       break;
  }
  q = p;
  p = p->next;
  }*/
  return head;
}

Task *getLastTaskInQueue()
{
   if(head == NULL)
        return NULL;
   return tail;
}
/*When job gets executed, then call this function to remove it from queue*/
Task *deleteFirstTask()
{
   if(head == NULL)
        return NULL;
   Task *p = head;
   head = head->next;
   /*Now head will point to next task in queue*/
   if(head == NULL)
        tail = NULL;
   return p;
}

/*This function will check migration possibility based on finish time only*/
Task * checkForMigration(double load, Task *p)
{
   if(p == head)
      p = p->next;
   while(p != NULL)
   {
        if(p->getExpectedFinishTime() > (load + p->getExecutionTime()))
            return p;
        else
            p = p->next;
    }
    return NULL;
}
```

/*This function will check migration possibility based on number of arrival and departure in transfer time. If expected finish time is less, then migrate job*/

```
Task * checkForMigration(double bMeanArrivalTime, double bMeanServiceTime, double
    bLoad, Task *p)
{
    while(p != NULL)
    {
        double transferTime = p->getTaskSize() / commCost1;
        /*factor = h * t  */
        double factor = transferTime / bMeanArrivalTime;
        /*ePower = e^factor  */
        double ePower = exp((-1) * factor);
        double prob = ePower;
        /*arrival will indicate number of arrivals in transfer time*/
        int arrival = 1;
        while(prob < ACCURACY)
        {
           prob += ((ePower * power(factor,arrival))/factorial(arrival));
           arrival++;
        }
        factor = transferTime / bMeanServiceTime;
        ePower = exp((-1) * factor);
        prob = ePower;
        /*departure will indicate number of departure in transfer time*/
        int departure = 1;
        while(prob < ACCURACY )
        {
            prob += ((ePower * power(factor,departure)) / factorial(departure));
            departure++;
        }
        arrival += bLoad;
        double expectedFinishTime = (arrival - departure) * bMeanServiceTime +
            estMeanServiceTime;
        if(expectedFinishTime - estMeanServiceTime < transferTime)
           expectedFinishTime = transferTime + estMeanServiceTime;
        /*If expected finish time for this job is less than current time,
         then migrate this job*/
        if(expectedFinishTime < p->getExpectedFinishTime())
            return p;
        else
           p = p->next;
    }
    return NULL;
}
```

/*This function will check migration possibility based on number of arrival and departure in transfer time. If expected finish time is less, then migrate job*/

```
Task * checkForMigration(double bMeanArrivalTime, double bMeanServiceTime, double
                         bLoad, Task *p, double bSpeedFactor)
{
  while(p != NULL)
  {
     double expectedFinishTime = bLoad * bMeanServiceTime + (estMeanServiceTime *
                                 getSpeedFactor() / bSpeedFactor);
     /*If expected finish time for this job is less than current time,
      then migrate this job*/
     if(expectedFinishTime < p->getExpectedFinishTime())
        return p;
     else
        p = p->next;
  }
  return NULL;
}

double getCommSpeed(int buddyId)
{
   int count;
   for(count = 0; count < buddySetCount; count++)
   {
      if(buddySet[count] == buddyId)
         break;
   }
   return commSpeed[count];
}

/*This function will check migration possibility based on number of arrival and departure in
transfer time. If expected finish time is less, then migrate job*/
Task * checkForMigration(double bMeanArrivalTime, double bMeanServiceTime, double
                         bLoad, Task *p, double commSpeed, double bSpeedFactor)
{
  while(p != NULL)
  {
     double transferTime = p->getTaskSize() / commSpeed;
     /*factor = h * t  */
     double factor = transferTime / bMeanArrivalTime;
     /*ePower = e^factor  */
     double ePower = exp((-1) * factor);
     double prob = ePower;
     /*arrival will indicate number of arrivals in transfer time*/
     int arrival = 1;
     while(prob < ACCURACY)
     {
        prob += ((ePower * power(factor,arrival))/factorial(arrival));
        arrival++;
     }
     factor = transferTime / bMeanServiceTime;
     ePower = exp((-1) * factor);
     prob = ePower;
     /*departure will indicate number of departure in transfer time*/
     int departure = 1;
     while(prob < ACCURACY )
     {
            prob += ((ePower * power(factor,departure)) / factorial(departure));
            departure++;
        }
        arrival += bLoad;
        double exeTime = p->getExpectedExecutionTime() / bSpeedFactor;
        double expectedFinishTime = (arrival - departure) * bMeanServiceTime + exeTime;
        if(expectedFinishTime - exeTime < transferTime)
            expectedFinishTime = transferTime + estMeanServiceTime;
        /*If expected finish time for this job is less than current time,
         then migrate this job*/
        if(expectedFinishTime < p->getExpectedFinishTime())
             return p;
        else
            p = p->next;
    }
    return NULL;
}

/*This function will insert migration task in migration linked list according to its arrival time*/
void insertMigrationTask(Task *p)
{
    p->next = NULL;
    if(headMigration == NULL)
    {
        headMigration = p;
        setNextMigrationEvent(p->getArrivalTime());
    }
    else
    {
       Task *q = headMigration;
       Task *q1 = NULL;
       while(q != NULL)
       {
            if(q->getArrivalTime() < p->getArrivalTime())
            {
               q1 = q;
               q = q->next;
            }
            else
               break;
       }
       p->next = q;
       if(q1 == NULL)
       {
           headMigration = p;
           setNextMigrationEvent(p->getArrivalTime());
       }
       else
            q1->next = p;
    }
}

/*This function will return first migration task from it linked list*/
Task * removeFirstMigrationTask()
{
    if(headMigration == NULL)
        return NULL;
    else
    {
        Task *p = headMigration;
        headMigration = headMigration->next;
        return p;
    }
}

/*This fucntion will return first migration task*/
Task * getNextMigrationTask()
{
    return headMigration;
}

/*This function will remove given task p from linked list*/
Task * removeTask(Task *p, double currentSystemTime)
{
    setJobsWaitingInQueue(-1);
    if(head == p)
    {
            p->setExecutionTime(p->getExecutionTime() - (currentSystemTime -
                        getLastEventTime()));
            head = head->next;
            if(head == NULL)
            {
                tail = NULL;
                setNextArrivalEvent(INFINITY);
            }
            p->next = NULL;
            Task *j1 = getNextTaskInQueue();
            if(j1 == NULL)
            {
                setNodeStatus(IDLE);
                setNextDepartureEvent(INFINITY);
            }
            else
            {
              j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
              setNextDepartureEvent(currentSystemTime + j1->getExecutionTime());
            }
            setLastEventTime(currentSystemTime);
            return head;
    }
    else
    {
            Task *j = head;
            while(j->next != p)
            {
              j = j->next;
            }
            j->next = p->next;
            if(tail == p)
                tail = j;
                p->next = NULL;
                return j->next;
            }
        }

        /*This function will print id of buddy set processor*/
        void printBuddySet()
        {
            for(int count = 0; count < buddySetCount; count++)
                cout << buddySet[count] << "\t";
            cout << "\n";
        }

        /*This function will print job id of tasks which are waiting in queue*/
        void printTasks()
        {
            Task *p = head;
            while(p != NULL)
            {
                cout << p->getJobID() << "\t";
                p = p->next;
            }
            cout << "\n";
        }
};
```

```
***************************************************************************************
  File : ELISA.cpp
  Description : This file contains implementation of ELISA algorithm. ELISA algorithm will try to do
  load balancing on each estimated time interval Te, based on estimated queue length of buddy
  processor. On each status exchange period Ts, they will transfer information (queue length, arrival
  time) to its buddy processors.
***************************************************************************************
#include <iostream>
#include <fstream>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

/*definition.h contains declaration of all necessary parameters.*/
#include "definition.h"
/*task.h file contains implementation of class task*/
#include "task.h"
/*node.h file contains implementation of class node*/
#include "node.h"

int main()
{
    /* Declare necessary variables*/
    int t = 0;
    double currentSystemTime = 0.0;        // currentSystemTime will indicate simulation time
    int count, idCount = 0;
    double nextTimePeriod = EST_PERIOD; // nextTimePeriod will be set to Te value
double statusPeriod = TIME_PERIOD;
int jobFinished = 0;                  // jobFinised will indicate number of jobs finished
double totalResponseTime = 0;         // It will find total response time of jobFinished jobs
double totalWaitingTime = 0;          // It will find total waiting time of jobFinished jobs
int max = 0;
int totalEstimation = 0;

srand(RANDOM_SEED);                   // This will just set random seed value

// Create as many object of Node as processors in system
Node *node = new Node[NUMBER_OF_PROCESSORS];
double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
double meanServiceTime[NUMBER_OF_PROCESSORS] = {6,6,6,6,3,4,2,2,3,4,3,4,3,4,2,2};

/*graph.txt is input file for network topology. This file is generated by GraphGenerator tool.
File format : ProcessorID BuddySetCount BuddySetID BuddySetID ... linkspeed1 linkspeed2 ...*/
ifstream in1("graph.txt",ios::in);
ofstream out("output.txt",ios::out);

int tempID,tempCount,*tempBuddy;
double *tempCommCost;
double a = 0, s= 0;
/*Following loop will read from file graph.txt and will set topology accordingly.*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
        in1 >> tempID;
        in1 >> tempCount;
        tempBuddy = new int[tempCount];
        int temp;
        for(temp = 0; temp < tempCount; temp++)
              in1 >> tempBuddy[temp];
        tempCommCost = new double[tempCount];
        for(temp = 0; temp < tempCount; temp++)
        {
              in1 >> tempCommCost[temp];
              tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
        }
        node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
        double uVar = getUniformRandomVariable();
        node[count].setSpeedFactor(1.0 + uVar);
        node[count].setEstMeanServiceTime(meanServiceTime[count]);
        delete [] tempBuddy;
}
in1.close();
int migration = 0;
double tempTime = 0;
/* Following loop will generate first arrival event for each processor*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
   tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
   node[count].setNextArrivalEvent(count);
}

struct EventList
{
    char eventType;
    int nodeNo;
    double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;
while(jobFinished < MAX_JOBS)
{
  eventList.eventType = 'T';
  eventList.nodeNo = -1;
  eventList.eventTime = nextTimePeriod;

  /*Following loop will find out which event will happen next : arrival, departure,
    migration,estimation interval. And also for which processor will this happen*/
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
      if(node[count].getNextArrivalEvent() < eventList.eventTime)
      {
          eventList.eventType = 'A';
          eventList.nodeNo = count;
          eventList.eventTime = node[count].getNextArrivalEvent();
      }
      if(node[count].getNextDepartureEvent() < eventList.eventTime)
      {
          eventList.eventType = 'D';
          eventList.nodeNo = count;
          eventList.eventTime = node[count].getNextDepartureEvent();
      }
      if(node[count].getNextMigrationEvent() < eventList.eventTime)
      {
          eventList.eventType = 'M';
          eventList.nodeNo = count;
          eventList.eventTime = node[count].getNextMigrationEvent();
      }
  }
  int nodeNo = eventList.nodeNo;
  currentSystemTime = eventList.eventTime;
  if(eventList.eventType == 'A')
  {
      /*Next event to happen is arrival for processor nodeNo*/
      node[nodeNo].incrementNoOfArrivalInPeriod();
      /* New job has arrived, so create execution time and program size for this job*/
      double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
      double size =
       getNormalRandomVariable(PROGRAM_SIZE_MEAN,PROGRAM_SIZE_VARIANCE);
      Task p(idCount,currentSystemTime,time,size,nodeNo,meanServiceTime[nodeNo]);
      /*add this task in queue*/
      node[nodeNo].addNewTask(p);
      /*If processor is idle, then process job immediately*/
      if(node[nodeNo].getNodeStatus() == IDLE)
      {
          node[nodeNo].setNodeStatus(BUSY);
          Task *j = node[nodeNo].getNextTaskInQueue();
          node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                                                    node[nodeNo].getSpeedFactor()));
```

```
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
            node[nodeNo].getLastEventTime());
        if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
              TIME_PERIOD));
        else
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
              node[nodeNo].getLastEventTime());
    }
    else
    {
      /*Put job in waiting queue*/
        node[nodeNo].setJobsWaitingInQueue(1);
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
    /*Now set time for next arrival event for this processor if number of jobs in system is less
            than MAX_JOBS*/
    time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
    if(idCount < MAX_JOBS)
      node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
    else
      node[nodeNo].setNextArrivalEvent(INFINITY);
    idCount = idCount + 1;
}
else if(eventList.eventType == 'D')
{
    /*Next event is departure for processor nodeNo*/
    Task *j = node[nodeNo].deleteFirstTask();
    if(j->getJobID() > MAX_JOBS)
     t+=1;
    jobFinished = jobFinished + 1;
    /*Calculate response time and waiting time for this processor*/
    totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
    totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                            j->getExecutionTime());
    if(jobFinished%2000 == 0)
    {
       cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
       cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
       cout << "\nTotal Execution Time " << currentSystemTime;
       cout << "\nNumber of migration " << migration;
       cout << "\n\n";
       out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
            "\t" <<currentSystemTime << "\t" <<migration<<"\n";
    }

    delete j;
    node[nodeNo].incrementNoOfDepartureInPeriod();
    Task *j1 = node[nodeNo].getNextTaskInQueue();
    /*If there is no job to process, then set processor to IDLE state*/
    if(j1 == NULL)
    {
       node[nodeNo].setNodeStatus(IDLE);
       node[nodeNo].setNextDepartureEvent(INFINITY);
    }
    else
    {
      /*If there is at least one job, then process that jon*/
      j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
      node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                    node[nodeNo].getSpeedFactor()));
      // decrement queue length by one
      node[nodeNo].setJobsWaitingInQueue(-1);
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
}
else if(eventList.eventType == 'M')
{
    //Next event is job arrival through migration for processor nodeNo
     Task *p = node[nodeNo].removeFirstMigrationTask();
     node[nodeNo].incrementNoOfArrivalInPeriod();
     //add job in queue
     node[nodeNo].addNewTask(*p);

     //if processor is idle,then process this job directly
     if(node[nodeNo].getNodeStatus() == IDLE)
     {
        node[nodeNo].setNodeStatus(BUSY);
        Task *j = node[nodeNo].getNextTaskInQueue();
        node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                    node[nodeNo].getSpeedFactor()));
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                            node[nodeNo].getLastEventTime());
        if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
              TIME_PERIOD));
        else
             node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
             node[nodeNo].getLastEventTime());
     }
     else
     {
       //place job in waiting queue
        node[nodeNo].setJobsWaitingInQueue(1);
     }
     node[nodeNo].setLastEventTime(currentSystemTime);
     //set time for next migration event for this processor
     p = node[nodeNo].getNextMigrationTask();
     if(p == NULL)
        node[nodeNo].setNextMigrationEvent(INFINITY);
     else
        node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
}
else
{
   //it is either estimation time period or status exchange period
   if((int)(currentSystemTime)%(TIME_PERIOD) == 0)
   {
     statusPeriod += TIME_PERIOD;
```

```
 // this is status exchange period. Every processor will calculate its load, arrival rate and
// service rate
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
      if(node[count].getNodeStatus() == IDLE)
      {
          double time = currentSystemTime - node[count].getLastEventTime();
          if(time > TIME_PERIOD)
               node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
          else
               node[count].incrementLastIdleTimePeriod(time);
      }
      node[count].calculateMeanArrivalTime();
      node[count].calculateMeanServiceTime();
      node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
      node[count].calculateEstCurrentLoad(currentSystemTime);
  }
   // Following loop will pass this information to all its buddy set
   // so this is status exchange communication
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
        for(tempCount = 0; tempCount < node[count].getBuddySetCount(); tempCount++)
        {
                 int buddyId = node[count].getBuddyId(tempCount);
                 node[count].setEstBuddyArrivalTime(tempCount,
                        node[buddyId-1].getEstMeanArrivalTime());
                 node[count].setEstBuddyServiceTime(tempCount,
                        node[buddyId-1].getEstMeanServiceTime());
                 node[count].setEstBuddyLoad(tempCount,
                        node[buddyId-1].getEstCurrentLoad());
        }
  }
 }
else
{
  // this is estimation period, so every processor will find load on its buddy processor
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
        node[count].calculateEstBuddyLoad(EST_PERIOD);
        totalEstimation = totalEstimation + node[count].getBuddySetCount();
  }
}
/* Now starts Load balancing code*/
double amountOfLoadAcceptance[NUMBER_OF_PROCESSORS]
                                   [NUMBER_OF_PROCESSORS];
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      for(int count1 = 0; count1 < NUMBER_OF_PROCESSORS; count1++)
        amountOfLoadAcceptance[count][count1] = 0.0;
}
double avgLoad[NUMBER_OF_PROCESSORS];
// Following loop will do balancing for each processor if it load is
// above avg load of its buddy processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
```

```
            // first find out avg load in buddy set
            int buddyCount = node[count].getBuddySetCount();
            avgLoad[count] = node[count].getJobsWaitingInQueue();
            for(tempCount = 0; tempCount < buddyCount; tempCount++)
            {
                avgLoad[count] += node[count].getEstBuddyLoad(tempCount);
            }
            avgLoad[count] = avgLoad[count] / (node[count].getBuddySetCount() + 1);
            double myLoad = node[count].getJobsWaitingInQueue();

            // if my load is greated than avg load, then transfer load
            if(avgLoad[count] < myLoad)
            {
                double extraLoad = myLoad - avgLoad[count];
                double availableCapacity = 0;
                // find out how much i can tranfer to my buddy processor
                for(tempCount = 0; tempCount < buddyCount; tempCount++)
                {
                    double buddyLoad = node[count].getEstBuddyLoad(tempCount);
                    if(buddyLoad < avgLoad[count])
                    {
                        availableCapacity += (avgLoad[count] - buddyLoad);
                    }
                }
                // amountOfLoadAcceptance array will indicate how many jobs can be transferred
                //to buddy processor
                for(tempCount = 0; tempCount < buddyCount; tempCount++)
                {
                    int buddyId = node[count].getBuddyId(tempCount);
                    double buddyLoad = node[count].getEstBuddyLoad(tempCount);
                    if(buddyLoad < avgLoad[count])
                    {
                        double bCapacity = avgLoad[count] - buddyLoad;
                        amountOfLoadAcceptance[count][buddyId - 1] = bCapacity *
                                    extraLoad / availableCapacity;
                    }
                }
            }
    }
    //Now tranfer load to buddy processor
    for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
    {
        double myLoad = node[count].getJobsWaitingInQueue();
        if(myLoad > avgLoad[count])
        {
            double extraLoad = myLoad - avgLoad[count];
            Task *p = node[count].getNextTaskInQueue();
            p = p->next;
            for(tempCount = 0; tempCount < NUMBER_OF_PROCESSORS;
                    tempCount++)
            {
                // if i can tranfer to buddy having ID tempCount, then tranfer load
                if(amountOfLoadAcceptance[count][tempCount] > 0.5)
                {
                    while(p != NULL)
```

```
                                    {
                                        Task *q = p;
                                        //remove task from waiting queue
                                        p = node[count].removeTask(p,currentSystemTime);
                                        double commCost = node[count].getCommSpeed(tempCount + 1);
                                        double transferTime = q->getTaskSize() / commCost;
                                        q->incrementNoOfTimesMigrated();
                                        migration += 1;
                                        if(max < q->getNoOfTimesMigrated())
                                             max = q->getNoOfTimesMigrated();
                                        q->setArrivalTime(currentSystemTime + transferTime);
                                        // insert task in buddy list after tranfer time
                                        node[tempCount].insertMigrationTask(q);
                                        amountOfLoadAcceptance[count][tempCount]--;
                                        node[count].incrementEstBuddyLoad(tempCount + 1);
                                        extraLoad--;
                                        if(amountOfLoadAcceptance[count][tempCount] < 0.5 || extraLoad
                                                <= 0)
                                            break;
                                    }
                                }
                                if(p == NULL || extraLoad <= 0)
                                    break;
                          }
                    }
              }
              nextTimePeriod = nextTimePeriod + EST_PERIOD;
           }
        }


// All jobs processed, so print final values
cout << "\nJob finished " << jobFinished;
cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
cout << "\nTotal Execution Time " << currentSystemTime;
cout << "\n Number of migration " << migration;
cout <<"\n\nResource Utilization\tQueue Length\n";
double avgUtil = 0.0;
double max1 = 0, min = 1;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      if(node[count].getNodeStatus() == IDLE)
         node[count].incrementNodeIdleTime(currentSystemTime -
                 node[count].getLastEventTime());
      double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
      cout << util;
      avgUtil += util;
      if(util > max1) max1 = util;
      if(util < min)   min = util;
      cout << "\t" << node[count].getJobsWaitingInQueue() << "\n";
}
out << min << "\n" << max1 << "\n" << avgUtil / NUMBER_OF_PROCESSORS << "\n\n" <<
  totalEstimation<< "\n";
```

```
cout << min << "\t" << max1 << "\t" << avgUtil / NUMBER_OF_PROCESSORS << "\t" <<
  totalEstimation<< "\n";
cout << "\nAverage Utilization " << avgUtil / NUMBER_OF_PROCESSORS ;
out.close();
cout << "\nMax Migration " << max;
cout << "\nTotal Estimation " << totalEstimation;
cout << "\n t = " << t;
getch();
return 0;
}


*********************************************************************************
  File : MELISA.cpp
  Description : This file contains implementation of MELISA algorithm.
*********************************************************************************
#include <iostream>
#include <fstream>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

/*definition.h contains declaration of all necessary parameters. */
#include "definition.h"
/*task.h file contains implementation of class task*/
#include "task.h"
/*node.h file contains implementation of class node*/
#include "node.h"

int main()
{
  /* Declare necessary variables*/
  int t = 0;
  double currentSystemTime = 0.0;          // currentSystemTime will indicate simulation time
  int count, idCount = 0;
  double nextTimePeriod = EST_PERIOD; // nextTimePeriod will be set to Te value
  double statusPeriod = TIME_PERIOD;
  int jobFinished = 0;                     // jobFinised will indicate number of jobs finished
  double totalResponseTime = 0;            // totalResponseTime will find total response time of
 jobFinished jobs
  double totalWaitingTime = 0;    // totalWaitingTime will find total waiting time of jobFinished jobs
  int max = 0;
  int totalEstimation = 0;

  srand(RANDOM_SEED);            // This will just set random seed value

 // Create as many object of Node as processors in system
  Node *node = new Node[NUMBER_OF_PROCESSORS];
  double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
  double meanServiceTime[NUMBER_OF_PROCESSORS] = {6,6,6,6,3,4,2,2,3,4,3,4,3,4,2,2};

  ifstream in1("graph.txt",ios::in);
  ofstream out("output2.txt",ios::out);
```

```
int tempID,tempCount,*tempBuddy;
double *tempCommCost;
double a = 0, s= 0;
/*Following loop will read from file graph.txt and will set topology accordingly.*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
        in1 >> tempID;
        in1 >> tempCount;
        tempBuddy = new int[tempCount];
        int temp;
        for(temp = 0; temp < tempCount; temp++)
              in1 >> tempBuddy[temp];
        tempCommCost = new double[tempCount];
        for(temp = 0; temp < tempCount; temp++)
        {
              in1 >> tempCommCost[temp];
              tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
        }
        node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
        double uVar = getUniformRandomVariable();
        node[count].setSpeedFactor(1.0 + uVar);
        node[count].setEstMeanServiceTime(meanServiceTime[count]);
        delete [] tempBuddy;
}
in1.close();
int migration = 0;
double tempTime = 0;
/* Following loop will generate first arrival event for each processor*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
   tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
   node[count].setNextArrivalEvent(count);
}

struct EventList
{
      char eventType;
      int nodeNo;
      double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;
while(jobFinished < MAX_JOBS)
{
   eventList.eventType = 'T';
   eventList.nodeNo = -1;
   eventList.eventTime = nextTimePeriod;

   /*Following loop will find out which event will happen next : arrival, departure, migration,
    estimation interval. And also for which processor will this happen*/
   for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
   {
       if(node[count].getNextArrivalEvent() < eventList.eventTime)
       {
          eventList.eventType = 'A';
```

```
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextArrivalEvent();
        }
        if(node[count].getNextDepartureEvent() < eventList.eventTime)
        {
            eventList.eventType = 'D';
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextDepartureEvent();
        }
        if(node[count].getNextMigrationEvent() < eventList.eventTime)
        {
            eventList.eventType = 'M';
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextMigrationEvent();
        }
    }
    int nodeNo = eventList.nodeNo;
    currentSystemTime = eventList.eventTime;
    if(eventList.eventType == 'A')
    {
        /*Next event to happen is arrival for processor nodeNo*/
        node[nodeNo].incrementNoOfArrivalInPeriod();
        /* New job has arrived, so create execution time and program size for this job*/
        double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
        double size =  getNormalRandomVariable(PROGRAM_SIZE_MEAN,
                       PROGRAM_SIZE_VARIANCE);
        Task p(idCount,currentSystemTime,time,size,nodeNo,meanServiceTime[nodeNo]);
        /*add this task in queue*/
        node[nodeNo].addNewTask(p);
        /*If processor is idle, then process job immediately*/
        if(node[nodeNo].getNodeStatus() == IDLE)
        {
            node[nodeNo].setNodeStatus(BUSY);
            Task *j = node[nodeNo].getNextTaskInQueue();
            node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                       node[nodeNo].getSpeedFactor()));
            node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                       node[nodeNo].getLastEventTime());
            if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                TIME_PERIOD));
            else
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
        }
        else
        {
          /*Put job in waiting queue*/
            node[nodeNo].setJobsWaitingInQueue(1);
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        /*Now set time for next arrival event for this processor if number of jobs in system is less
         than MAX_JOBS*/
        time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
        if(idCount < MAX_JOBS)
        node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
      else
        node[nodeNo].setNextArrivalEvent(INFINITY);
      idCount = idCount + 1;
}
else if(eventList.eventType == 'D')
{
    /*Next event is departure for processor nodeNo*/
    Task *j = node[nodeNo].deleteFirstTask();
    if(j->getJobID() > MAX_JOBS)
     t+=1;
    jobFinished = jobFinished + 1;
    /*Calculate response time and waiting time for this processor*/
    totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
    totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                               j->getExecutionTime());
    if(jobFinished%2000 == 0)
    {
        cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
        cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
        cout << "\nTotal Execution Time " << currentSystemTime;
        cout << "\nNumber of migration " << migration;
        cout << "\n\n";
        out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
             "\t" <<currentSystemTime << "\t" <<migration<<"\n";
    }

    delete j;
    node[nodeNo].incrementNoOfDepartureInPeriod();
    Task *j1 = node[nodeNo].getNextTaskInQueue();
    /*If there is no job to process, then set processor to IDLE state*/
    if(j1 == NULL)
    {
        node[nodeNo].setNodeStatus(IDLE);
        node[nodeNo].setNextDepartureEvent(INFINITY);
    }
    else
    {
      /*If there is at least one job, then process that jon*/
      j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
      node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                    node[nodeNo].getSpeedFactor()));
      // decrement queue length by one
      node[nodeNo].setJobsWaitingInQueue(-1);
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
}
else if(eventList.eventType == 'M')
{
    //Next event is job arrival through migration for processor nodeNo
     Task *p = node[nodeNo].removeFirstMigrationTask();
     node[nodeNo].incrementNoOfArrivalInPeriod();
     //add job in queue
     node[nodeNo].addNewTask(*p);
     //if processor is idle, then process this job directly
```

```
        if(node[nodeNo].getNodeStatus() == IDLE)
        {
            node[nodeNo].setNodeStatus(BUSY);
            Task *j = node[nodeNo].getNextTaskInQueue();
            node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                        node[nodeNo].getSpeedFactor()));
            node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                        node[nodeNo].getLastEventTime());
            if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                TIME_PERIOD));
            else
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
        }
        else
        {
          //place job in waiting queue
            node[nodeNo].setJobsWaitingInQueue(1);
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        //set time for next migration event for this processor
        p = node[nodeNo].getNextMigrationTask();
        if(p == NULL)
            node[nodeNo].setNextMigrationEvent(INFINITY);
        else
            node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
    }
    else
    {
      //it is either estimation time period or status exchange period
      if((int)(currentSystemTime)%(TIME_PERIOD) == 0)
      {
        statusPeriod += TIME_PERIOD;
       // this is status exchange period
       // every processor will calculate its load, mean arrival time and service time
        for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
        {
            if(node[count].getNodeStatus() == IDLE)
            {
                double time = currentSystemTime - node[count].getLastEventTime();
                if(time > TIME_PERIOD)
                    node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
                else
                    node[count].incrementLastIdleTimePeriod(time);
            }
            node[count].calculateMeanArrivalTime();
            node[count].calculateMeanServiceTime();
            node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
            node[count].calculateEstCurrentLoad(currentSystemTime);
        }
        // Following loop will pass this information to all its buddy set
         // so this is status exchange communication
        for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
        {
            for(tempCount = 0; tempCount < node[count].getBuddySetCount(); tempCount++)
            {
                    int buddyId = node[count].getBuddyId(tempCount);
                    node[count].setEstBuddyArrivalTime(tempCount,
                            node[buddyId-1].getEstMeanArrivalTime());
                    node[count].setEstBuddyServiceTime(tempCount,
                            node[buddyId-1].getEstMeanServiceTime());
                    node[count].setEstBuddyLoad(tempCount,
                            node[buddyId-1].getEstCurrentLoad());
            }
    }
  }
else
{
  // this is estimation period, so every processor will find load on its buddy processor
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
        node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
        node[count].calculateEstBuddyLoad(EST_PERIOD);
        totalEstimation = totalEstimation + node[count].getBuddySetCount();
  }
}
/* Now starts Load balancing code*/
double amountOfLoadAcceptance[NUMBER_OF_PROCESSORS]
                [NUMBER_OF_PROCESSORS];
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      for(int count1 = 0; count1 < NUMBER_OF_PROCESSORS; count1++)
        amountOfLoadAcceptance[count][count1] = 0.0;
}
double avgLoad[NUMBER_OF_PROCESSORS];
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      // first find out avg load in buddy set
      int buddyCount = node[count].getBuddySetCount();
      avgLoad[count] = node[count].getJobsWaitingInQueue() *
                node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
      double totalWeight = node[count].getSpeedFactor();
      for(tempCount = 0; tempCount < buddyCount; tempCount++)
      {
          int buddyId = node[count].getBuddyId(tempCount);
          avgLoad[count] += (node[count].getEstBuddyLoad(tempCount) *
          node[count].getEstBuddyServiceTime(tempCount) *
                                node[buddyId-1].getSpeedFactor());
          totalWeight += node[buddyId - 1].getSpeedFactor();
      }
      avgLoad[count] = avgLoad[count] / totalWeight;
      double myLoad = node[count].getJobsWaitingInQueue() *
                node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
        double commLoad = avgLoad[count] * node[count].getSpeedFactor();
      // if my load is greated than avg load, then transfer load
      if(commLoad < myLoad)
      {
          double extraLoad = myLoad - commLoad;
          double availableCapacity = 0;
```

```
                // find out how much i can tranfer to my buddy processor
                for(tempCount = 0; tempCount < buddyCount; tempCount++)
                {
                        int buddyId = node[count].getBuddyId(tempCount);
                        double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                                node[count].getEstBuddyServiceTime(tempCount) * node[buddyId -
                                        1].getSpeedFactor();
                        if(buddyLoad < avgLoad[count] * node[buddyId - 1].getSpeedFactor())
                        {
                                availableCapacity += (avgLoad[count] *
                                        node[buddyId - 1].getSpeedFactor() - buddyLoad);
                        }
                }
                // amountOfLoadAcceptance array will indicate how many jobs can be transferred
                to buddy processor
                for(tempCount = 0; tempCount < buddyCount; tempCount++)
                {
                        int buddyId = node[count].getBuddyId(tempCount);
                        double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                                node[count].getEstBuddyServiceTime(tempCount) *
                                 node[buddyId – 1].getSpeedFactor();
                        if(buddyLoad < avgLoad[count] * node[buddyId - 1].getSpeedFactor())
                        {
                                double bCapacity = avgLoad[count] *
                                        node[buddyId - 1].getSpeedFactor() -    buddyLoad;
                                amountOfLoadAcceptance[count][buddyId - 1] = bCapacity *
                                        extraLoad / availableCapacity;
                        }
                }
        }
}
//Now transfer load to buddy processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
        double myLoad = node[count].getJobsWaitingInQueue() *
                node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
        if(myLoad > avgLoad[count] * node[count].getSpeedFactor())
        {
                double extraLoad = myLoad - avgLoad[count] * node[count].getSpeedFactor();
                for(tempCount = 0; tempCount < NUMBER_OF_PROCESSORS;
                        tempCount++)
                {
                    // if i can tranfer to buddy having ID tempCount, then tranfer load
                    Task *p = node[count].getNextTaskInQueue();
                            p = p->next;
                    if(amountOfLoadAcceptance[count][tempCount] > 0.5)
                    {
                        double bMeanArrivalTime =
                                node[tempCount].getEstMeanArrivalTime();
                        double bMeanServiceTime =
                                node[tempCount].getEstMeanServiceTime();
                        double bLoad = node[tempCount].getEstCurrentLoad();
                        double bSpeedFactor = node[tempCount].getSpeedFactor();
                        double commCost = node[count].getCommSpeed(tempCount + 1);
                         while(p != NULL)
                    {
                       p = node[count].checkForMigration(bMeanArrivalTime,
                              bMeanServiceTime, bLoad,p,commCost,bSpeedFactor);
                       if(p == NULL)
                         break;
                       Task *q = p;
                       //remove task from waiting queue
                       p = node[count].removeTask(p,currentSystemTime);
                       double transferTime = q->getTaskSize() / commCost;
                       q->incrementNoOfTimesMigrated();
                       migration += 1;
                       if(max < q->getNoOfTimesMigrated())
                            max = q->getNoOfTimesMigrated();
                       q->setArrivalTime(currentSystemTime + transferTime);
                       // insert task in buddy list after tranfer time
                       node[tempCount].insertMigrationTask(q);
                       amountOfLoadAcceptance[count][tempCount] -=
                     node[count].getEstMeanServiceTime()*node[count].getSpeedFactor();
                       node[count].incrementEstBuddyLoad(tempCount + 1,
                                   node[count].getEstMeanServiceTime()*
                                   node[count].getSpeedFactor());
                       extraLoad -= node[count].getEstMeanServiceTime() *
                              node[count].getSpeedFactor();
                       if(amountOfLoadAcceptance[count][tempCount] < 0.5 || extraLoad
                                   <= 0)
                           break;
                    }
                 }
                 if(p == NULL || extraLoad <= 0)
                    break;
              }
          }
      }
      nextTimePeriod = nextTimePeriod + EST_PERIOD;
     }
   }


// All jobs processed, so print final values
cout << "\nJob finished " << jobFinished;
cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
cout << "\nTotal Execution Time " << currentSystemTime;
cout << "\n Number of migration " << migration;
cout <<"\n\nResource Utilization\tQueue Length\n";
double avgUtil = 0.0;
double max1 = 0, min = 1;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
     if(node[count].getNodeStatus() == IDLE)
        node[count].incrementNodeIdleTime(currentSystemTime -
              node[count].getLastEventTime());
     double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
     cout << util;
     avgUtil += util;
```

```
        if(util > max1)
          max1 = util;
        if(util < min)
          min = util;
        cout << "\t" << node[count].getJobsWaitingInQueue() << "\n";
    }
    out << min << "\n" << max1 << "\n" << avgUtil / NUMBER_OF_PROCESSORS << "\n\n" <<
      totalEstimation<< "\n";
    cout << "\nAverage Utilization " << avgUtil / NUMBER_OF_PROCESSORS ;
    out.close();
    cout << "\nMax Migration " << max;
    cout << "\nTotal Estimation " << totalEstimation;
    cout << "\n t = " << t;
    getch();
    return 0;
    }


***************************************************************************************
  File : RMELISA.cpp
  Description : This file contains implementation of R-MELISA algorithm.
***************************************************************************************
#include <iostream>
#include <fstream>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

#include "definition.h"
#include "task.h"
#include "node.h"

int main()
{
  /* Declare necessary variables*/
  int t = 0;
  double currentSystemTime = 0.0;       // currentSystemTime will indicate simulation time
  int count, idCount = 0;
  double nextTimePeriod = EST_PERIOD; // nextTimePeriod will be set to Te value
  double statusPeriod = TIME_PERIOD;
  int jobFinished = 0;                  // jobFinised will indicate number of jobs finished
  double totalResponseTime = 0;          // totalResponseTime will find total response time of
 jobFinished jobs
  double totalWaitingTime = 0;      // totalWaitingTime will find total waiting time of jobFinished
 jobs
  int max = 0;
  int totalEstimation = 0;

  srand(RANDOM_SEED);            // This will just set random seed value

  Node *node = new Node[NUMBER_OF_PROCESSORS];
 double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,3,4,2,4,3,4,4,3,4,4,3,4,2,4,4};
 double meanServiceTime[NUMBER_OF_PROCESSORS] = {3,3,2,3,2,1,3,1,4,4,3,3,1,3,3,3};
```

```
ifstream in1("graph.txt",ios::in);
ofstream out("output2.txt",ios::out);

int tempID,tempCount,*tempBuddy;
double *tempCommCost;
double a = 0, s= 0;
/*Following loop will read from file graph.txt and will set topology accordingly.*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
        in1 >> tempID;
        in1 >> tempCount;
        tempBuddy = new int[tempCount];
        int temp;
        for(temp = 0; temp < tempCount; temp++)
              in1 >> tempBuddy[temp];
        tempCommCost = new double[tempCount];
        for(temp = 0; temp < tempCount; temp++)
        {
              in1 >> tempCommCost[temp];
              tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
        }
        node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
        double uVar = getUniformRandomVariable();
        node[count].setSpeedFactor(1.0 + uVar);
        node[count].setEstMeanServiceTime(meanServiceTime[count]);
        delete [] tempBuddy;
}
in1.close();
int migration = 0;
double tempTime = 0;
/* Following loop will generate first arrival event for each processor*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
   tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
   node[count].setNextArrivalEvent(count);
}

struct EventList
{
      char eventType;
      int nodeNo;
      double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;
while(jobFinished < MAX_JOBS)
{
    eventList.eventType = 'T';
    eventList.nodeNo = -1;
    eventList.eventTime = nextTimePeriod;

    /*Following loop will find out which event will happen next : arrival, departure,
     migration,estimation interval. And also for which processor will this happen*/
    for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
    {
        if(node[count].getNextArrivalEvent() < eventList.eventTime)
        {
            eventList.eventType = 'A';
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextArrivalEvent();
        }
        if(node[count].getNextDepartureEvent() < eventList.eventTime)
        {
            eventList.eventType = 'D';
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextDepartureEvent();
        }
        if(node[count].getNextMigrationEvent() < eventList.eventTime)
        {
            eventList.eventType = 'M';
            eventList.nodeNo = count;
            eventList.eventTime = node[count].getNextMigrationEvent();
        }
    }
    int nodeNo = eventList.nodeNo;
    currentSystemTime = eventList.eventTime;
    if(eventList.eventType == 'A')
    {
        /*Next event to happen is arrival for processor nodeNo*/
        node[nodeNo].incrementNoOfArrivalInPeriod();
        /* New job has arrived, so create execution time and program size for this job*/
        double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
        double size =  getNormalRandomVariable(PROGRAM_SIZE_MEAN,
                       PROGRAM_SIZE_VARIANCE);
        Task p(idCount,currentSystemTime,time,size,nodeNo,meanServiceTime[nodeNo]);
        /*add this task in queue*/
        node[nodeNo].addNewTask(p);
        /*If processor is idle, then process job immediately*/
        if(node[nodeNo].getNodeStatus() == IDLE)
        {
            node[nodeNo].setNodeStatus(BUSY);
            Task *j = node[nodeNo].getNextTaskInQueue();
            node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                       node[nodeNo].getSpeedFactor()));
            node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                       node[nodeNo].getLastEventTime());
            if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                TIME_PERIOD));
            else
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
        }
        else
        {
          /*Put job in waiting queue*/
            node[nodeNo].setJobsWaitingInQueue(1);
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        /*Now set time for next arrival event for this processor
        if number of jobs in system is less than MAX_JOBS*/
        time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
        if(idCount < MAX_JOBS)
           node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
        else
           node[nodeNo].setNextArrivalEvent(INFINITY);
        idCount = idCount + 1;
    }
    else if(eventList.eventType == 'D')
    {
       /*Next event is departure for processor nodeNo*/
       Task *j = node[nodeNo].deleteFirstTask();
       if(j->getJobID() > MAX_JOBS)
        t+=1;
       jobFinished = jobFinished + 1;
       /*Calculate response time and waiting time for this processor*/
       totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
       totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                            j->getExecutionTime());
       if(jobFinished%2000 == 0)
       {
          cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
          cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
          cout << "\nTotal Execution Time " << currentSystemTime;
          cout << "\nNumber of migration " << migration;
          cout << "\n\n";
          out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
                   "\t" <<currentSystemTime << "\t" <<migration<<"\n";
       }

       delete j;
       node[nodeNo].incrementNoOfDepartureInPeriod();
       Task *j1 = node[nodeNo].getNextTaskInQueue();
       /*If there is no job to process, then set processor to IDLE state*/
       if(j1 == NULL)
       {
           node[nodeNo].setNodeStatus(IDLE);
           node[nodeNo].setNextDepartureEvent(INFINITY);
       }
       else
       {
          /*If there is at least one job, then process that jon*/
          j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
          node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                   node[nodeNo].getSpeedFactor()));
          // decrement queue length by one
          node[nodeNo].setJobsWaitingInQueue(-1);
       }
       node[nodeNo].setLastEventTime(currentSystemTime);
    }
    else if(eventList.eventType == 'M')
    {
       //Next event is job arrival through migration for processor nodeNo
        Task *p = node[nodeNo].removeFirstMigrationTask();
        node[nodeNo].incrementNoOfArrivalInPeriod();
        //add job in queue
        node[nodeNo].addNewTask(*p);

        //if processor is idle,then process this job directly
        if(node[nodeNo].getNodeStatus() == IDLE)
        {
            node[nodeNo].setNodeStatus(BUSY);
            Task *j = node[nodeNo].getNextTaskInQueue();
            node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                        node[nodeNo].getSpeedFactor()));
            node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                node[nodeNo].getLastEventTime());
            if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                TIME_PERIOD));
            else
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
        }
        else
        {
          //place job in waiting queue
            node[nodeNo].setJobsWaitingInQueue(1);
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        //set time for next migration event for this processor
        p = node[nodeNo].getNextMigrationTask();
        if(p == NULL)
            node[nodeNo].setNextMigrationEvent(INFINITY);
        else
            node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
    }
    else
    {
      //it is either estimation time period or status exchange period
      if((int)(currentSystemTime)%(TIME_PERIOD) == 0)
      {
        statusPeriod += TIME_PERIOD;
      // this is status exchange period
      // every processor will calculate its load, mean arrival time and service time
        for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
        {
            if(node[count].getNodeStatus() == IDLE)
            {
                double time = currentSystemTime - node[count].getLastEventTime();
                if(time > TIME_PERIOD)
                    node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
                else
                    node[count].incrementLastIdleTimePeriod(time);
            }
            node[count].calculateMeanArrivalTime();
            node[count].calculateMeanServiceTime();
            node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
            node[count].calculateEstCurrentLoad(currentSystemTime);
        }
    // Following loop will pass this information to all its buddy set
    // so this is status exchange communication
   for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
   {
         for(tempCount = 0; tempCount < node[count].getBuddySetCount(); tempCount++)
         {
                  int buddyId = node[count].getBuddyId(tempCount);
                  node[count].setEstBuddyArrivalTime(tempCount,
                         node[buddyId-1].getEstMeanArrivalTime());
                  node[count].setEstBuddyServiceTime(tempCount,
                         node[buddyId-1].getEstMeanServiceTime());
                  node[count].setEstBuddyLoad(tempCount,
                         node[buddyId-1].getEstCurrentLoad());
         }
   }
  }
else
{
   // this is estimation period, so every processor will find load on its buddy processor
   for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
   {
         node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
         node[count].calculateEstBuddyLoad(EST_PERIOD);
         totalEstimation = totalEstimation + node[count].getBuddySetCount();
   }
}
/* Now starts Load balancing code*/
double amountOfLoadAcceptance[NUMBER_OF_PROCESSORS]
                                          [NUMBER_OF_PROCESSORS];
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      for(int count1 = 0; count1 < NUMBER_OF_PROCESSORS; count1++)
         amountOfLoadAcceptance[count][count1] = 0.0;
}
double avgLoad[NUMBER_OF_PROCESSORS];
// Following loop will do balancing for each processor if it load is
// above avg load of its buddy processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      // first find out avg load in buddy set
      int buddyCount = node[count].getBuddySetCount();
      avgLoad[count] = node[count].getJobsWaitingInQueue() *
                node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
      double totalWeight = node[count].getSpeedFactor();
      for(tempCount = 0; tempCount < buddyCount; tempCount++)
      {
          int buddyId = node[count].getBuddyId(tempCount);
          avgLoad[count] += (node[count].getEstBuddyLoad(tempCount) *
                                   node[count].getEstBuddyServiceTime(tempCount) *
                              node[buddyId-1].getSpeedFactor());
          totalWeight += node[buddyId - 1].getSpeedFactor();
      }
      avgLoad[count] = avgLoad[count] / totalWeight;
```

```
        double myLoad = node[count].getJobsWaitingInQueue() *
                        node[count].getEstMeanServiceTime() *
                        node[count].getSpeedFactor();
        double commLoad = avgLoad[count] * node[count].getSpeedFactor();
        // if my load is less than avg load, then accept load
        if(commLoad > myLoad)
        {
            double extraLoad = commLoad - myLoad;
            double availableCapacity = 0;
            // find out how much i can receive from my buddy processor
            for(tempCount = 0; tempCount < buddyCount; tempCount++)
            {
                int buddyId = node[count].getBuddyId(tempCount);
                double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                        node[count].getEstBuddyServiceTime(tempCount) *
                        node[buddyId - 1].getSpeedFactor();
                if(buddyLoad > avgLoad[count] * node[buddyId - 1].getSpeedFactor())
                {
                        availableCapacity += (buddyLoad - avgLoad[count] *
                                node[buddyId - 1].getSpeedFactor());
                }
            }
            for(tempCount = 0; tempCount < buddyCount; tempCount++)
            {
                int buddyId = node[count].getBuddyId(tempCount);
                double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                                        node[count].getEstBuddyServiceTime(tempCount)
                                       * node[buddyId - 1].getSpeedFactor();
                if(buddyLoad > avgLoad[count] * node[buddyId - 1].getSpeedFactor())
                {
                        double bCapacity = buddyLoad - avgLoad[count] *
                                       node[buddyId - 1].getSpeedFactor();
                        amountOfLoadAcceptance[buddyId - 1][count] = bCapacity *
                                       extraLoad /  availableCapacity;
                }
            }
        }
}
//Now tranfer load to buddy processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
        double myLoad = node[count].getJobsWaitingInQueue()
                *node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
        if(myLoad > avgLoad[count] * node[count].getSpeedFactor())
        {
            double extraLoad = myLoad - avgLoad[count] * node[count].getSpeedFactor();
            for(tempCount = 0; tempCount < NUMBER_OF_PROCESSORS;
                tempCount++)
            {
                // if i can tranfer to buddy having ID tempCount, then tranfer load
                Task *p = node[count].getNextTaskInQueue();
                p = p->next;
                if(amountOfLoadAcceptance[count][tempCount] > 0.5)
                {
                        double bMeanArrivalTime =
                                node[tempCount].getEstMeanArrivalTime();
                        double bMeanServiceTime =
                                node[tempCount].getEstMeanServiceTime();
                        double bLoad = node[tempCount].getEstCurrentLoad();
                        double bSpeedFactor = node[tempCount].getSpeedFactor();
                        double commCost = node[count].getCommSpeed(tempCount + 1);
                        while(p != NULL)
                        {
                          p = node[count].checkForMigration(bMeanArrivalTime,
                                bMeanServiceTime, bLoad,p,commCost,bSpeedFactor);
                          if(p == NULL)
                            break;
                          Task *q = p;
                          //remove task from waiting queue
                          p = node[count].removeTask(p,currentSystemTime);
                          double transferTime = q->getTaskSize() / commCost;
                          q->incrementNoOfTimesMigrated();
                          migration += 1;
                          if(max < q->getNoOfTimesMigrated())
                              max = q->getNoOfTimesMigrated();
                          q->setArrivalTime(currentSystemTime + transferTime);
                          // insert task in buddy list after tranfer time
                          node[tempCount].insertMigrationTask(q);
                          amountOfLoadAcceptance[count][tempCount] -=
                                node[count].getEstMeanServiceTime() *
                                node[count].getSpeedFactor();
                          node[count].incrementEstBuddyLoad(tempCount + 1,
                                node[count].getEstMeanServiceTime() *
                                node[count].getSpeedFactor());
                          extraLoad -= node[count].getEstMeanServiceTime() *
                                node[count].getSpeedFactor();
                          if(amountOfLoadAcceptance[count][tempCount] < 0.5 || extraLoad
                                <= 0)
                              break;
                        }
                      }
                      if(p == NULL || extraLoad <= 0)
                        break;
                  }
              }
          }
          nextTimePeriod = nextTimePeriod + EST_PERIOD;
        }
    }


// All jobs processed, so print final values
cout << "\nJob finished " << jobFinished;
cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
cout << "\nTotal Execution Time " << currentSystemTime;
cout << "\n Number of migration " << migration;
cout <<"\n\nResource Utilization\tQueue Length\n";
double avgUtil = 0.0;
double max1 = 0, min = 1;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      if(node[count].getNodeStatus() == IDLE)
         node[count].incrementNodeIdleTime(currentSystemTime -
                  node[count].getLastEventTime());
      double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
      cout << util;
      avgUtil += util;
      if(util > max1)
        max1 = util;
      if(util < min)
        min = util;
      cout << "\t" << node[count].getJobsWaitingInQueue() << "\n";
}
out << min << "\n" << max1 << "\n" << avgUtil / NUMBER_OF_PROCESSORS << "\n\n" <<
  totalEstimation<< "\n";
cout << "\nAverage Utilization " << avgUtil / NUMBER_OF_PROCESSORS ;
out.close();
cout << "\nMax Migration " << max;
cout << "\nTotal Estimation " << totalEstimation;
cout << "\n t = " << t;
getch();
return 0;
}


*********************************************************************************
  File : LBA.cpp
  Description : This file contains implementation of load balancing on each arrival. It estimates
  expected starting time for each new arrival and if expected starting time on buddy processor is less
  than starting time on that processor, then job migration decision will be taken.
*********************************************************************************
#include <iostream>
#include <fstream>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

#include "definition.h"
#include "task.h"
#include "node.h"

int main()
{
  int t =0;
  double currentSystemTime = 0.0;       // it indicates current simulation time
  int count, idCount = 0;
  double nextTimePeriod = TIME_PERIOD;
  int jobFinished = 0;                  // it indicates how many jobs have finished
  double totalResponseTime = 0;         // it is total response time of jobFinished jobs
  double totalWaitingTime = 0;          // it is total waiting time of jobFinished jobs
  int totalEstimation = 0;
```

```
srand(RANDOM_SEED);           // this will set random seed

Node *node = new Node[NUMBER_OF_PROCESSORS];
double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
double meanServiceTime[NUMBER_OF_PROCESSORS] = {6,6,6,6,3,4,2,2,3,4,3,4,3,4,2,2};

ifstream in1("graph.txt",ios::in);
ofstream out("output1.txt",ios::out);

int tempID,tempCount,*tempBuddy;
double *tempCommCost;
double a = 0, s = 0;
/*Following loop will read from file graph.txt and will set topology accordingly.*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      in1 >> tempID;
      in1 >> tempCount;
      tempBuddy = new int[tempCount];
      int temp;
      for(temp = 0; temp < tempCount; temp++)
            in1 >> tempBuddy[temp];
      tempCommCost = new double[tempCount];
      for(temp = 0; temp < tempCount; temp++)
      {
            in1 >> tempCommCost[temp];
            tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
      }
      node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
      double uVar = getUniformRandomVariable();
      node[count].setSpeedFactor(1.0 + uVar);
      node[count].setEstMeanServiceTime(meanServiceTime[count]);
      delete [] tempBuddy;
}
in1.close();
int migration = 0;
double tempTime = 0;
/* Following loop will generate first arrival event for each processor*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
  tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
  node[count].setNextArrivalEvent(count);
}

struct EventList
{
     char eventType;
     int nodeNo;
     double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;

while(jobFinished < MAX_JOBS)
{
```

```
eventList.eventType = 'T';
eventList.nodeNo = -1;
eventList.eventTime = nextTimePeriod;
/*Following loop will find out which event will happen next : arrival, departure,
  migration,estimation interval. And also for which processor will this happen*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
    if(node[count].getNextArrivalEvent() < eventList.eventTime)
    {
        eventList.eventType = 'A';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextArrivalEvent();
    }
    if(node[count].getNextDepartureEvent() < eventList.eventTime)
    {
        eventList.eventType = 'D';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextDepartureEvent();
    }
    if(node[count].getNextMigrationEvent() < eventList.eventTime)
    {
        eventList.eventType = 'M';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextMigrationEvent();
    }
}
int nodeNo = eventList.nodeNo;
currentSystemTime = eventList.eventTime;
if(eventList.eventType == 'A')
{
    /*Next event to happen is arrival for processor nodeNo*/
    /* New job has arrived, so create execution time and program size for this job*/
    double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
    double size =  getNormalRandomVariable(PROGRAM_SIZE_MEAN,
                   PROGRAM_SIZE_VARIANCE);
    Task p(idCount,currentSystemTime,time,size,nodeNo,meanServiceTime[nodeNo]);
    /*If node is idle, then directly process this job*/
    if(node[nodeNo].getNodeStatus() == IDLE)
    {
        node[nodeNo].incrementNoOfArrivalInPeriod();
        node[nodeNo].addNewTask(p);
        node[nodeNo].setNodeStatus(BUSY);
        Task *j = node[nodeNo].getNextTaskInQueue();
        node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                node[nodeNo].getSpeedFactor()));
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                node[nodeNo].getLastEventTime());
        if((nextTimePeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
            node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                        (nextTimePeriod - TIME_PERIOD));
        else
            node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                        node[nodeNo].getLastEventTime());

    }
```

```
else
{
    /*Node is not idle*/
     time = currentSystemTime - nextTimePeriod + TIME_PERIOD;
     /*Calculate total load on this processor*/
     node[nodeNo].calculateEstCurrentLoad(currentSystemTime);
     /*This will be job's expected starting time on this processor*/
     double totalLoad = node[nodeNo].getEstCurrentLoad() *
       node[nodeNo].getEstMeanServiceTime();
     /*THRESHOLD value is not used, so it is set to zero*/
     if(totalLoad < THRESHOLD)
     {
        node[nodeNo].incrementNoOfArrivalInPeriod();
        node[nodeNo].addNewTask(p);
        node[nodeNo].setJobsWaitingInQueue(1);
     }
     else
     {
        double minTime = totalLoad + p.getExecutionTime() /
                                node[nodeNo].getSpeedFactor();
        int migrateToNode = nodeNo;
        /*Now find number of arrivals and departures that can take place
        on buddy processors in time (transferTime + time)*/
        /*Loop for each buddy processor in set*/
        for(tempCount = 0; tempCount < node[nodeNo].getBuddySetCount();
               tempCount++)
        {
            int buddyId = node[nodeNo].getBuddyId(tempCount);
            double commCost = node[nodeNo].getCommSpeed(buddyId);
            double transferTime = p.getTaskSize() / commCost;
            double totalTime = transferTime + time;
            double bMeanArrivalTime =
                       node[nodeNo].getEstBuddyArrivalTime(tempCount);
            double bMeanServiceTime =
                       node[nodeNo].getEstBuddyServiceTime(tempCount);
            double factor = totalTime / bMeanArrivalTime;
            double ePower = exp((-1) * factor);
            double prob = ePower;
            /*First find number of arrivals which can be possible*/
            int arrival = 1;
            while(prob < ACCURACY)
            {
                 prob += ((ePower * power(factor,arrival))/factorial(arrival));
                 arrival++;
            }
            factor = totalTime / bMeanServiceTime;
            ePower = exp((-1) * factor);
            prob = ePower;
            int departure = 1;
            /*Now find number of departures which can be possible*/
            while(prob < ACCURACY )
            {
                 prob += ((ePower * power(factor,departure)) / factorial(departure));
                 departure++;
            }
```

```
                    /*Now estimate expected starting time of this job on buddy processor*/
                    arrival += node[nodeNo].getEstBuddyLoad(tempCount);
                    double expFinishTime =  (arrival - departure) * bMeanServiceTime;
                    /*If this time is less than transfer time, then set time to transfer time*/
                    if(expFinishTime < transferTime)
                        expFinishTime = transferTime;
                    expFinishTime += p.getExecutionTime() / node[buddyId - 1].getSpeedFactor();
                    totalEstimation++;
                    /*If expected starting time is less than current mintime,
                    then set this time as minTime and this node as destination node*/
                    if(minTime > expFinishTime)
                    {
                           minTime = expFinishTime;
                           migrateToNode = buddyId - 1;
                           break;
                    }
                }
                /*If migrateToNode != nodeNo, then we have taken decision of migrating a job*/
                if(migrateToNode != nodeNo)
                {
                       p.incrementNoOfTimesMigrated();
                       p.setExpectedFinishTime(minTime);
                       migration += 1;
                       double commCost = node[nodeNo].getCommSpeed(migrateToNode + 1);
                       double transferTime = p.getTaskSize() / commCost;
                       /*set arrival time for this job after tranfer time*/
                       p.setArrivalTime(currentSystemTime + transferTime);
                       Task *q = new Task(p);
                       /*insert this job in migration queue of destination processor*/
                       node[migrateToNode].insertMigrationTask(q);
                       node[nodeNo].incrementEstBuddyLoad(migrateToNode +
                               1,p.getExecutionTime()/node[migrateToNode].getSpeedFactor());
                }
                else
                {
                    /*we are not migrgating this job, so add it to waiting queue of this processor*/
                       node[nodeNo].incrementNoOfArrivalInPeriod();
                       node[nodeNo].addNewTask(p);
                       node[nodeNo].setJobsWaitingInQueue(1);
                }
            }
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
        /*set next arrival event for this processor*/
        if(idCount < MAX_JOBS)
           node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
        else
           node[nodeNo].setNextArrivalEvent(INFINITY);
        idCount = idCount + 1;
    }
    else if(eventList.eventType == 'D')
    {
        /*departure event is taken place for node nodeNo*/
        Task *j = node[nodeNo].deleteFirstTask();
    if(j->getJobID() > MAX_JOBS)
    t+=1;
    /*increment no of job finished and calculate response time and waiting time
    of this processor*/
    jobFinished = jobFinished + 1;
    totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
    totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                          j->getExecutionTime());
    if(jobFinished%2000 == 0)
    {
       cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
       cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
       cout << "\nTotal Execution Time " << currentSystemTime;
       cout << "\nNumber of migration " << migration;
       cout << "\n\n";
       out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
                  "\t" << currentSystemTime << "\t" << migration << "\n";
    }
    delete j;
    node[nodeNo].incrementNoOfDepartureInPeriod();
    /*Check for new task in queue*/
    Task *j1 = node[nodeNo].getNextTaskInQueue();
    if(j1 == NULL)
    {
        /*No job in queue, so set node status to IDLE*/
        node[nodeNo].setNodeStatus(IDLE);
        node[nodeNo].setNextDepartureEvent(INFINITY);
    }
    else
    {
       /*There is at least one job in queue, so start processing jon*/
       j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
       node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                  node[nodeNo].getSpeedFactor()));
       node[nodeNo].setJobsWaitingInQueue(-1);
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
 }
 else if(eventList.eventType == 'M')
 {
    /*Migration event has taken place for node nodeNo*/
    Task *p = node[nodeNo].removeFirstMigrationTask();
    /*this node is idle, so start processing this migrated job*/
    if(node[nodeNo].getNodeStatus() == IDLE)
    {
         node[nodeNo].incrementNoOfArrivalInPeriod();
         node[nodeNo].addNewTask(*p);
         node[nodeNo].setNodeStatus(BUSY);
         Task *j = node[nodeNo].getNextTaskInQueue();
         node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                  node[nodeNo].getSpeedFactor()));
         node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                  node[nodeNo].getLastEventTime());
         if((nextTimePeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
        node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
            (nextTimePeriod - TIME_PERIOD));
    else
        node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
            node[nodeNo].getLastEventTime());
}
/*If this job is already migrated upto MIGRATION_LIMIT,
then place it in waiting queue as it can not be further migrated*/
else if(p->getNoOfTimesMigrated() >= MIGRATION_LIMIT)
{
    node[nodeNo].incrementNoOfArrivalInPeriod();
    node[nodeNo].addNewTask(*p);
    node[nodeNo].setJobsWaitingInQueue(1);
}
else
{
    /*Try to find out any buddy processor which can process this job faster than me.*/
    double time = currentSystemTime - nextTimePeriod + TIME_PERIOD;
    node[nodeNo].calculateEstCurrentLoad(currentSystemTime);
    double totalLoad = node[nodeNo].getEstCurrentLoad() *
      node[nodeNo].getEstMeanServiceTime();
    if(totalLoad < THRESHOLD)
    {
       node[nodeNo].incrementNoOfArrivalInPeriod();
       node[nodeNo].addNewTask(*p);
       node[nodeNo].setJobsWaitingInQueue(1);
    }
    else
    {
       double minTime = totalLoad + p->getExecutionTime() /
                   node[nodeNo].getSpeedFactor();
       int migrateToNode = nodeNo;
       /*This will loop for each buddy processor in set*/
       for(tempCount = 0; tempCount < node[nodeNo].getBuddySetCount();
            tempCount++)
       {
           int buddyId = node[nodeNo].getBuddyId(tempCount);
           double commCost = node[nodeNo].getCommSpeed(buddyId);
           double transferTime = p->getTaskSize() / commCost;
           double totalTime = transferTime + time;
           double bMeanArrivalTime =
              node[nodeNo].getEstBuddyArrivalTime(tempCount);
           double bMeanServiceTime =
              node[nodeNo].getEstBuddyServiceTime(tempCount);
           double factor = totalTime / bMeanArrivalTime;
           double ePower = exp((-1) * factor);
           double prob = ePower;
           /*Find number of possible arrivals */
           int arrival = 1;
           while(prob < ACCURACY)
           {
                prob += ((ePower * power(factor,arrival))/factorial(arrival));
                arrival++;
           }
           factor = totalTime / bMeanServiceTime;
```

```
                ePower = exp((-1) * factor);
                prob = ePower;
                /*find number of possible departures*/
                int departure = 1;
                while(prob < ACCURACY )
                {
                      prob += ((ePower * power(factor,departure)) / factorial(departure));
                      departure++;
                }
                //estimate expected starting time of job on buddy processor*/
                arrival += node[nodeNo].getEstBuddyLoad(tempCount);
                double expFinishTime =  (arrival - departure) * bMeanServiceTime;
                /*if expected starting time is less transfer time,then set it to transfer time*/
                if(expFinishTime < transferTime)
                       expFinishTime = transferTime;
                expFinishTime += p->getExecutionTime() /
                                              node[buddyId - 1].getSpeedFactor();
               totalEstimation++;
               /*If expected starting time is less than my time, then transfer job */
               if(minTime > expFinishTime)
               {
                      minTime = expFinishTime;
                      migrateToNode = buddyId - 1;
                      break;
               }
           }
           if(migrateToNode != nodeNo)
           {
               //we will migrate this job on processor havin ID migrateToNode
                p->incrementNoOfTimesMigrated();
                p->setExpectedFinishTime(minTime);
                migration += 1;
                double commCost = node[nodeNo].getCommSpeed(migrateToNode + 1);
                double transferTime = p->getTaskSize() / commCost;
                p->setArrivalTime(currentSystemTime + transferTime);
                //place this job on migration queue of destination processor
                node[migrateToNode].insertMigrationTask(p);
                node[nodeNo].incrementEstBuddyLoad(migrateToNode + 1,
                    p->getExecutionTime() / node[migrateToNode].getSpeedFactor());
           }
           else
           {
             //there is no processor which can start processing job faster than this
             // so we will not migrate this job on any processor
                node[nodeNo].incrementNoOfArrivalInPeriod();
                node[nodeNo].addNewTask(*p);
                node[nodeNo].setJobsWaitingInQueue(1);
           }
       }
   }
   node[nodeNo].setLastEventTime(currentSystemTime);
   p = node[nodeNo].getNextMigrationTask();
   if(p == NULL)
       node[nodeNo].setNextMigrationEvent(INFINITY);
   else
```

```
                node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
            }
            else
            {
              /*This is status exchange period
              Every prcoessor will calculate its load, arrival time and service time*/
              for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
              {
                    if(node[count].getNodeStatus() == IDLE)
                    {
                         double time = currentSystemTime - node[count].getLastEventTime();
                         if(time > TIME_PERIOD)
                              node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
                         else
                              node[count].incrementLastIdleTimePeriod(time);
                    }
                    node[count].calculateMeanArrivalTime();
                    node[count].calculateMeanServiceTime();
                    node[count].calculateEstCurrentLoad(currentSystemTime);
              }
              /*Now every processor will pass information to its buddy processor.
              Follwoing loop will do the same task*/
              for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
              {
                    for(tempCount = 0; tempCount < node[count].getBuddySetCount(); tempCount++)
                    {
                         int buddyId = node[count].getBuddyId(tempCount);
                         node[count].setEstBuddyArrivalTime(tempCount,
                                   node[buddyId-1].getEstMeanArrivalTime());
                         node[count].setEstBuddyServiceTime(tempCount,
                                   node[buddyId-1].getEstMeanServiceTime());
                         node[count].setEstBuddyLoad(tempCount,
                                   node[buddyId-1].getEstCurrentLoad());
                    }
              }
              nextTimePeriod = nextTimePeriod + TIME_PERIOD;
            }
        }
cout << "\njob finished " << jobFinished;
cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
cout << "\nTotal Execution Time " << currentSystemTime;
cout << "\nNumber of migration " << migration;
cout <<"\n\nResource Utilization\tQueue Length\n";
double avgUtil = 0;
double max1 = 0, min = 1;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      if(node[count].getNodeStatus() == IDLE)
         node[count].incrementNodeIdleTime(currentSystemTime -
                                   node[count].getLastEventTime());
      double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
      cout << util;
      avgUtil += util;
      if(util > max1)
```

```
            max1 = util;
         if(util < min)
            min = util;
           cout << "\t" << node[count].getJobsWaitingInQueue() << "\n";
}
out << min << "\n" << max1 << "\n" << avgUtil / NUMBER_OF_PROCESSORS << "\n\n" <<
  totalEstimation<< "\n";
cout<<"\n\n";
cout << min << "\t" << max1 << "\t" << avgUtil / NUMBER_OF_PROCESSORS << "\t" <<
  totalEstimation<< "\n";
out.close();
cout <<"\nAvg Utilization "<<avgUtil/NUMBER_OF_PROCESSORS;
cout << "\nTotal Estimation "<<totalEstimation;
cout << "\n t = " << t;
getch();
return 0;
}


*********************************************************************************
  File : PIA.cpp
  Description : This file contains implementation of PIA algorithm.
*********************************************************************************
#include <iostream>
#include <fstream>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

#include "definition.h"
#include "task.h"
#include "node.h"

int main()
{
  double currentSystemTime = 0.0;
  int count, idCount = 0;
  double nextTimePeriod = TIME_PERIOD;
  int jobFinished = 0;
  double totalResponseTime = 0;
  double totalWaitingTime = 0;
  int totalEstimation = 0;

  srand(RANDOM_SEED);

  Node *node = new Node[NUMBER_OF_PROCESSORS];
  double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
  double meanServiceTime[NUMBER_OF_PROCESSORS] = {6,6,6,6,3,4,2,2,3,4,3,4,3,4,2,2};

  ifstream in1("graph.txt",ios::in);
  ofstream out("output3.txt", ios::out);

  int tempID,tempCount,*tempBuddy;
```

```
 double *tempCommCost;
 double a = 0, s = 0;

 for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
 {
        in1 >> tempID;
        in1 >> tempCount;
        tempBuddy = new int[tempCount];
        int temp;
        for(temp = 0; temp < tempCount; temp++)
             in1 >> tempBuddy[temp];
        tempCommCost = new double[tempCount];
        for(temp = 0; temp < tempCount; temp++)
        {
             in1 >> tempCommCost[temp];
             tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
        }
        node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
        double uVar = getUniformRandomVariable();
        node[count].setSpeedFactor(1.0 + uVar);
        node[count].setEstMeanArrivalTime(meanArrivalTime[count]);
        node[count].setEstMeanServiceTime(meanServiceTime[count]);
        delete [] tempBuddy;
 }

in1.close();
int migration = 0;
double tempTime = 0;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
  tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
  node[count].setNextArrivalEvent(count);
}

struct EventList
{
     char eventType;
     int nodeNo;
     double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;
while(jobFinished < MAX_JOBS)
{
   eventList.eventType = 'T';
   eventList.nodeNo = -1;
   eventList.eventTime = nextTimePeriod;

   for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
   {
       if(node[count].getNextArrivalEvent() < eventList.eventTime)
       {
          eventList.eventType = 'A';
          eventList.nodeNo = count;
          eventList.eventTime = node[count].getNextArrivalEvent();
```

```
    }
    if(node[count].getNextDepartureEvent() < eventList.eventTime)
    {
       eventList.eventType = 'D';
       eventList.nodeNo = count;
       eventList.eventTime = node[count].getNextDepartureEvent();
    }
    if(node[count].getNextMigrationEvent() < eventList.eventTime)
    {
       eventList.eventType = 'M';
       eventList.nodeNo = count;
       eventList.eventTime = node[count].getNextMigrationEvent();
    }
}
int nodeNo = eventList.nodeNo;
currentSystemTime = eventList.eventTime;
if(eventList.eventType == 'A')
{
    double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
    double size =  getNormalRandomVariable(PROGRAM_SIZE_MEAN,
                   PROGRAM_SIZE_VARIANCE);
    Task p(idCount,currentSystemTime,time,size,nodeNo, meanServiceTime[count]);
    if(node[nodeNo].getNodeStatus() == IDLE)
    {
         node[nodeNo].incrementNoOfArrivalInPeriod();
         node[nodeNo].addNewTask(p);
         node[nodeNo].setNodeStatus(BUSY);
         Task *j = node[nodeNo].getNextTaskInQueue();
         node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                   node[nodeNo].getSpeedFactor()));
         node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                   node[nodeNo].getLastEventTime());
         if((nextTimePeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                   (nextTimePeriod - TIME_PERIOD));
         else
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                   node[nodeNo].getLastEventTime());
    }
    else
    {
         time = currentSystemTime - nextTimePeriod + TIME_PERIOD;
         node[nodeNo].calculateExactCurrentLoad(currentSystemTime);
         double totalLoad = node[nodeNo].getExactCurrentLoad();
         if(totalLoad < THRESHOLD)
         {
            node[nodeNo].incrementNoOfArrivalInPeriod();
            node[nodeNo].addNewTask(p);
            node[nodeNo].setJobsWaitingInQueue(1);
         }
         else
         {
            double minTime = totalLoad + p.getExecutionTime() /
                   node[nodeNo].getSpeedFactor();;
            int migrateToNode = nodeNo;
```

```
        for(tempCount = 0; tempCount < node[nodeNo].getBuddySetCount();
            tempCount++)
        {
            int buddyId = node[nodeNo].getBuddyId(tempCount);
            double commCost = node[nodeNo].getCommSpeed(buddyId);
            double transferTime = p.getTaskSize() / commCost;
            double totalTime = transferTime + time;
            double bMeanArrivalTime = node[buddyId - 1].getEstMeanArrivalTime();
            double bMeanServiceTime = node[buddyId - 1].getEstMeanServiceTime();
            double factor = transferTime / bMeanArrivalTime;
            double ePower = exp((-1) * factor);
            double prob = ePower;
            int arrival = 1;
            while(prob < ACCURACY)
            {
                prob += ((ePower * power(factor,arrival))/factorial(arrival));
                arrival++;
            }
            factor = transferTime / bMeanServiceTime;
            ePower = exp((-1) * factor);
            prob = ePower;
            int departure = 1;
            while(prob < ACCURACY)
            {
                prob += ((ePower * power(factor,departure)) / factorial(departure));
                departure++;
            }
            node[buddyId – 1].
              .calculateExactCurrentLoad(currentSystemTime,transferTime);
            double expFinishTime = (arrival - departure) * bMeanServiceTime +
              node[buddyId - 1].getExactCurrentLoad();
            if(expFinishTime < transferTime)
                expFinishTime = transferTime;
            expFinishTime += p.getExecutionTime() / node[buddyId - 1].getSpeedFactor();
            if(minTime > expFinishTime)
            {
                minTime = expFinishTime;
                migrateToNode = buddyId - 1;
            }
            totalEstimation++;
      }
      if(migrateToNode != nodeNo)
      {
            p.incrementNoOfTimesMigrated();
            p.setExpectedFinishTime(minTime);
            migration += 1;
            double commCost = node[nodeNo].getCommSpeed(migrateToNode + 1);
            double transferTime = p.getTaskSize() / commCost;
            p.setArrivalTime(currentSystemTime + transferTime);
            Task *q = new Task(p);
            node[migrateToNode].insertMigrationTask(q);
      }
      else
      {
            node[nodeNo].incrementNoOfArrivalInPeriod();
```

```
                node[nodeNo].addNewTask(p);
                node[nodeNo].setJobsWaitingInQueue(1);
            }
        }
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
    time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
    if(idCount < MAX_JOBS)
      node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
    else
      node[nodeNo].setNextArrivalEvent(INFINITY);
    idCount = idCount + 1;
}
else if(eventList.eventType == 'D')
{
   Task *j = node[nodeNo].deleteFirstTask();
   jobFinished = jobFinished + 1;
   totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
   totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                           j->getExecutionTime());
   if(jobFinished%2000 == 0)
   {
      cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
      cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
      cout << "\nTotal Execution Time " << currentSystemTime;
      cout << "\nNumber of migration " << migration;
      cout << "\n\n";
      out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
                "\t" << currentSystemTime << "\t" <<migration << "\n";
   }
   delete j;
   node[nodeNo].incrementNoOfDepartureInPeriod();
   Task *j1 = node[nodeNo].getNextTaskInQueue();
   if(j1 == NULL)
   {
       node[nodeNo].setNodeStatus(IDLE);
       node[nodeNo].setNextDepartureEvent(INFINITY);
   }
   else
   {
     j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
     node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                node[nodeNo].getSpeedFactor()));
    node[nodeNo].setJobsWaitingInQueue(-1);
   }
   node[nodeNo].setLastEventTime(currentSystemTime);
}
else if(eventList.eventType == 'M')
{
    Task *p = node[nodeNo].removeFirstMigrationTask();
    if(node[nodeNo].getNodeStatus() == IDLE)
    {
        node[nodeNo].incrementNoOfArrivalInPeriod();
        node[nodeNo].addNewTask(*p);
        node[nodeNo].setNodeStatus(BUSY);
        Task *j = node[nodeNo].getNextTaskInQueue();
        node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                node[nodeNo].getSpeedFactor()));
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                node[nodeNo].getLastEventTime());
        if((nextTimePeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
            node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
            (nextTimePeriod - TIME_PERIOD));
        else
            node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
    }
    else if(p->getNoOfTimesMigrated() >= MIGRATION_LIMIT)
    {
        node[nodeNo].incrementNoOfArrivalInPeriod();
        node[nodeNo].addNewTask(*p);
        node[nodeNo].setJobsWaitingInQueue(1);
    }
    else
    {
        double time = currentSystemTime - nextTimePeriod + TIME_PERIOD;
        node[nodeNo].calculateExactCurrentLoad(currentSystemTime);
        double totalLoad = node[nodeNo].getExactCurrentLoad();
        if(totalLoad < THRESHOLD)
        {
           node[nodeNo].incrementNoOfArrivalInPeriod();
           node[nodeNo].addNewTask(*p);
           node[nodeNo].setJobsWaitingInQueue(1);
        }
        else
        {
           double minTime = totalLoad + p->getExecutionTime() /
                   node[nodeNo].getSpeedFactor();
           int migrateToNode = nodeNo;
           for(tempCount = 0; tempCount < node[nodeNo].getBuddySetCount();
                   tempCount++)
           {
               int buddyId = node[nodeNo].getBuddyId(tempCount);
               double commCost = node[nodeNo].getCommSpeed(buddyId);
               double transferTime = p->getTaskSize() / commCost;
               double bMeanArrivalTime = node[buddyId - 1].getEstMeanArrivalTime();
               double bMeanServiceTime = node[buddyId - 1].getEstMeanServiceTime();
               double factor = transferTime / bMeanArrivalTime;
               double ePower = exp((-1) * factor);
               double prob = ePower;
               int arrival = 1;
               while(prob < ACCURACY)
               {
                   prob += ((ePower * power(factor,arrival))/factorial(arrival));
                   arrival++;
               }
               factor = transferTime / bMeanServiceTime;
               ePower = exp((-1) * factor);
               prob = ePower;
               int departure = 1;
```

```
                while(prob < ACCURACY)
                {
                    prob += ((ePower * power(factor,departure)) / factorial(departure));
                    departure++;
                }
                node[buddyId - 1].
                        calculateExactCurrentLoad(currentSystemTime,transferTime);
                double expFinishTime = (arrival - departure) * bMeanServiceTime +
                        node[buddyId - 1].getExactCurrentLoad();
                if(expFinishTime < transferTime)
                      expFinishTime = transferTime;
                expFinishTime += p->getExecutionTime() /
                        node[buddyId - 1].getSpeedFactor();
                if(minTime > expFinishTime)
                {
                      minTime = expFinishTime;
                      migrateToNode = buddyId - 1;
                }
                totalEstimation++;
            }
            if(migrateToNode != nodeNo)
            {
                  p->incrementNoOfTimesMigrated();
                  p->setExpectedFinishTime(minTime);
                  migration += 1;
                  double commCost = node[nodeNo].getCommSpeed(migrateToNode + 1);
                  double transferTime = p->getTaskSize() / commCost;
                  p->setArrivalTime(currentSystemTime + transferTime);
                  node[migrateToNode].insertMigrationTask(p);
            }
            else
            {
                  node[nodeNo].incrementNoOfArrivalInPeriod();
                  node[nodeNo].addNewTask(*p);
                  node[nodeNo].setJobsWaitingInQueue(1);
            }
        }
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
    p = node[nodeNo].getNextMigrationTask();
    if(p == NULL)
        node[nodeNo].setNextMigrationEvent(INFINITY);
    else
        node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
}
else
{
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
        if(node[count].getNodeStatus() == IDLE)
        {
            double time = currentSystemTime - node[count].getLastEventTime();
            if(time > TIME_PERIOD)
                node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
            else
```

```
                        node[count].incrementLastIdleTimePeriod(time);
                    }
                }
              nextTimePeriod = nextTimePeriod + TIME_PERIOD;
            }
        }
    out.close();
    cout << "\nAvg Response Time " << totalResponseTime / MAX_JOBS;
    cout << "\nAvg Waiting Time " << totalWaitingTime / MAX_JOBS;
    cout << "\nTotal Execution Time " << currentSystemTime;
    cout << "\nNumber of migration " << migration;
    cout <<"\n\nResource Utilization\tQueue Length\n";
    double avgUtil = 0;
    for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
    {
          if(node[count].getNodeStatus() == IDLE)
            node[count].incrementNodeIdleTime(currentSystemTime -
            node[count].getLastEventTime());
          double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
          cout << util;
          avgUtil += util;
          cout << "\t\t\t" << node[count].getJobsWaitingInQueue() << "\n";
    }
    cout <<"\nAvg Utilization "<<avgUtil/NUMBER_OF_PROCESSORS;
    cout << "\nTotal Estimation " << totalEstimation;
    getch();
    return 0;
    }


    ******************************************************************************
      File : LBLS.cpp
      Description : This file contains implementation of LBLS algorithm.
    ******************************************************************************
    #include <iostream>
    #include <fstream>
    #include <conio.h>
    #include <stdlib.h>
    #include <math.h>
    using namespace std;

    /*definition.h contains declaration of all necessary parameters. */
    #include "definition.h"
    /*task.h file contains implementation of class task*/
    #include "task.h"
    /*node.h file contains implementation of class node*/
    #include "node.h"

    int main()
    {
      /* Declare necessary variables*/
      int t = 0;
      double currentSystemTime = 0.0;       // currentSystemTime will indicate simulation time
      int count, idCount = 0;
      double nextTimePeriod = EST_PERIOD; // nextTimePeriod will be set to Te value
```

```
 double statusPeriod = TIME_PERIOD;
 int jobFinished = 0;              // jobFinised will indicate number of jobs finished
 double totalResponseTime = 0;   // totalResponseTime will find total response time of jobFinished
jobs
 double totalWaitingTime = 0;    // totalWaitingTime will find total waiting time of jobFinished jobs
 int max = 0;
 int totalEstimation = 0;

 srand(RANDOM_SEED);            // This will just set random seed value

 Node *node = new Node[NUMBER_OF_PROCESSORS];
double meanArrivalTime[NUMBER_OF_PROCESSORS] = {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
 double meanServiceTime[NUMBER_OF_PROCESSORS] = {6,6,6,6,3,4,2,2,3,4,3,4,3,4,2,2};

 ifstream in1("graph.txt",ios::in);
 ofstream out("output1.txt",ios::out);

 int tempID,tempCount,*tempBuddy;
 double *tempCommCost;
 double a = 0, s= 0;
 /*Following loop will read from file graph.txt and will set topology accordingly.*/
 for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
 {
        in1 >> tempID;
        in1 >> tempCount;
        tempBuddy = new int[tempCount];
        int temp;
        for(temp = 0; temp < tempCount; temp++)
              in1 >> tempBuddy[temp];
        tempCommCost = new double[tempCount];
        for(temp = 0; temp < tempCount; temp++)
        {
              in1 >> tempCommCost[temp];
              tempCommCost[temp] = tempCommCost[temp] * 1024.0 * 1024.0;
        }
        node[count].setBuddySet(tempID,tempCount,tempBuddy,tempCommCost);
        double uVar = getUniformRandomVariable();
        node[count].setSpeedFactor(1.0 + uVar);
        node[count].setEstMeanServiceTime(meanServiceTime[count]);
        delete [] tempBuddy;
 }
in1.close();
int migration = 0;
double tempTime = 0;
/* Following loop will generate first arrival event for each processor*/
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
   tempTime = getExponentialRandomVariable(meanArrivalTime[count]);
   node[count].setNextArrivalEvent(count);
}

 struct EventList
 {
      char eventType;
      int nodeNo;
```

```
    double eventTime;
}eventList;

idCount = NUMBER_OF_PROCESSORS;
while(jobFinished < MAX_JOBS)
{
  eventList.eventType = 'T';
  eventList.nodeNo = -1;
  eventList.eventTime = nextTimePeriod;

  /*Following loop will find out which event will happen next : arrival, departure,
    migration,estimation interval. And also for which processor will this happen*/
  for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
  {
     if(node[count].getNextArrivalEvent() < eventList.eventTime)
     {
        eventList.eventType = 'A';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextArrivalEvent();
     }
     if(node[count].getNextDepartureEvent() < eventList.eventTime)
     {
        eventList.eventType = 'D';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextDepartureEvent();
     }
     if(node[count].getNextMigrationEvent() < eventList.eventTime)
     {
        eventList.eventType = 'M';
        eventList.nodeNo = count;
        eventList.eventTime = node[count].getNextMigrationEvent();
     }
  }
  int nodeNo = eventList.nodeNo;
  currentSystemTime = eventList.eventTime;
  if(eventList.eventType == 'A')
  {
     /*Next event to happen is arrival for processor nodeNo*/
     node[nodeNo].incrementNoOfArrivalInPeriod();
     /* New job has arrived, so create execution time and program size for this job*/
     double time = getExponentialRandomVariable(meanServiceTime[nodeNo]);
     double size =
    getNormalRandomVariable(PROGRAM_SIZE_MEAN,PROGRAM_SIZE_VARIANCE);
     Task p(idCount,currentSystemTime,time,size,nodeNo,meanServiceTime[nodeNo]);
     /*add this task in queue*/
     node[nodeNo].addNewTask(p);
     /*If processor is idle, then process job immediately*/
     if(node[nodeNo].getNodeStatus() == IDLE)
     {
        node[nodeNo].setNodeStatus(BUSY);
        Task *j = node[nodeNo].getNextTaskInQueue();
        node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                  node[nodeNo].getSpeedFactor()));
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                  node[nodeNo].getLastEventTime());
```

```
            if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                TIME_PERIOD));
            else
                node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                node[nodeNo].getLastEventTime());
        }
        else
        {
          /*Put job in waiting queue*/
            node[nodeNo].setJobsWaitingInQueue(1);
        }
        node[nodeNo].setLastEventTime(currentSystemTime);
        /*Now set time for next arrival event for this processor
        if number of jobs in system is less than MAX_JOBS*/
        time = getExponentialRandomVariable(meanArrivalTime[nodeNo]);
        if(idCount < MAX_JOBS)
          node[nodeNo].setNextArrivalEvent(currentSystemTime + time);
        else
          node[nodeNo].setNextArrivalEvent(INFINITY);
        idCount = idCount + 1;
    }
    else if(eventList.eventType == 'D')
    {
       /*Next event is departure for processor nodeNo*/
       Task *j = node[nodeNo].deleteFirstTask();
       if(j->getJobID() > MAX_JOBS)
        t+=1;
       jobFinished = jobFinished + 1;
       /*Calculate response time and waiting time for this processor*/
       totalResponseTime += (currentSystemTime - j->getActualArrivalTime());
       totalWaitingTime += (currentSystemTime - j->getActualArrivalTime() –
                               j->getExecutionTime());
       if(jobFinished%2000 == 0)
       {
          cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
          cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
          cout << "\nTotal Execution Time " << currentSystemTime;
          cout << "\nNumber of migration " << migration;
          cout << "\n\n";
          out << totalResponseTime / jobFinished << "\t" << totalWaitingTime / jobFinished <<
                    "\t" <<currentSystemTime << "\t" <<migration<<"\n";
       }
       delete j;
       node[nodeNo].incrementNoOfDepartureInPeriod();
       Task *j1 = node[nodeNo].getNextTaskInQueue();
       /*If there is no job to process, then set processor to IDLE state*/
       if(j1 == NULL)
       {
           node[nodeNo].setNodeStatus(IDLE);
           node[nodeNo].setNextDepartureEvent(INFINITY);
       }
       else
       {
         /*If there is at least one job, then process that jon*/
```

```
      j1->setWaitingTime(currentSystemTime - j1->getArrivalTime());
      node[nodeNo].setNextDepartureEvent(currentSystemTime + (j1->getExecutionTime() /
                  node[nodeNo].getSpeedFactor()));
      // decrement queue length by one
      node[nodeNo].setJobsWaitingInQueue(-1);
    }
    node[nodeNo].setLastEventTime(currentSystemTime);
  }
  else if(eventList.eventType == 'M')
  {
    //Next event is job arrival through migration for processor nodeNo
     Task *p = node[nodeNo].removeFirstMigrationTask();
     node[nodeNo].incrementNoOfArrivalInPeriod();
     //add job in queue
     node[nodeNo].addNewTask(*p);
     //if processor is idle,then process this job directly
     if(node[nodeNo].getNodeStatus() == IDLE)
     {
        node[nodeNo].setNodeStatus(BUSY);
        Task *j = node[nodeNo].getNextTaskInQueue();
        node[nodeNo].setNextDepartureEvent(currentSystemTime + (j->getExecutionTime() /
                  node[nodeNo].getSpeedFactor()));
        node[nodeNo].incrementNodeIdleTime(currentSystemTime -
                  node[nodeNo].getLastEventTime());
        if((statusPeriod-TIME_PERIOD) > node[nodeNo].getLastEventTime())
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime - (statusPeriod -
                  TIME_PERIOD));
        else
              node[nodeNo].incrementLastIdleTimePeriod(currentSystemTime -
                  node[nodeNo].getLastEventTime());
     }
     else
     {
       //place job in waiting queue
        node[nodeNo].setJobsWaitingInQueue(1);
     }
     node[nodeNo].setLastEventTime(currentSystemTime);
     //set time for next migration event for this processor
     p = node[nodeNo].getNextMigrationTask();
     if(p == NULL)
        node[nodeNo].setNextMigrationEvent(INFINITY);
     else
        node[nodeNo].setNextMigrationEvent(p->getArrivalTime());
  }
  else
  {
    //it is either estimation time period or status exchange period
    if((int)(currentSystemTime)%(TIME_PERIOD) == 0)
    {
      statusPeriod += TIME_PERIOD;
     // this is status exchange period
     // every processor will calculate its load, mean arrival time and service time
      for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
      {
         if(node[count].getNodeStatus() == IDLE)
        {
            double time = currentSystemTime - node[count].getLastEventTime();
            if(time > TIME_PERIOD)
                node[count].incrementLastIdleTimePeriod(TIME_PERIOD);
            else
                node[count].incrementLastIdleTimePeriod(time);
        }
        node[count].calculateMeanArrivalTime();
        node[count].calculateMeanServiceTime();
        node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
        node[count].calculateEstCurrentLoad(currentSystemTime);
    }
    // Following loop will pass this information to all its buddy set
    // so this is status exchange communication
    for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
    {
        for(tempCount = 0; tempCount < node[count].getBuddySetCount(); tempCount++)
        {
            int buddyId = node[count].getBuddyId(tempCount);
            node[count].setEstBuddyArrivalTime(tempCount,
                node[buddyId-1].getEstMeanArrivalTime());
            node[count].setEstBuddyServiceTime(tempCount,
                node[buddyId-1].getEstMeanServiceTime());
            node[count].setEstBuddyLoad(tempCount,
                node[buddyId-1].getEstCurrentLoad());
        }
    }
  }
else
{
    // this is estimation period, so every processor will find load on its buddy processor
    for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
    {
        node[count].calculateExpectedFinishTimeofTasks(currentSystemTime);
        node[count].calculateEstBuddyLoad(EST_PERIOD);
        totalEstimation = totalEstimation + node[count].getBuddySetCount();
    }
}
/* Now starts Load balancing code*/
double amountOfLoadAcceptance[NUMBER_OF_PROCESSORS]
                             [NUMBER_OF_PROCESSORS];
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
    for(int count1 = 0; count1 < NUMBER_OF_PROCESSORS; count1++)
        amountOfLoadAcceptance[count][count1] = 0.0;
}
double avgLoad[NUMBER_OF_PROCESSORS];
// Following loop will do balancing for each processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
    // first find out avg load in buddy set
    int buddyCount = node[count].getBuddySetCount();
    double totalWeight = node[count].getSpeedFactor();
    avgLoad[count] = node[count].getJobsWaitingInQueue() *
            node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
        for(tempCount = 0; tempCount < buddyCount; tempCount++)
        {
            int buddyId = node[count].getBuddyId(tempCount);
            totalWeight += node[buddyId - 1].getSpeedFactor();
            avgLoad[count] += (node[count].getEstBuddyLoad(tempCount) *
                    node[count].getEstBuddyServiceTime(tempCount) *
                    node[buddyId-1].getSpeedFactor());
        }
        avgLoad[count] = avgLoad[count] / totalWeight;
        double myLoad = node[count].getJobsWaitingInQueue() *
                    node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
        double commLoad = avgLoad[count] * node[count].getSpeedFactor();
        // if my load is greated than avg load, then transfer load
        if(commLoad < myLoad)
        {
            double extraLoad = myLoad - commLoad;
            double availableCapacity = 0;
            // find out how much i can tranfer to my buddy processor
            for(tempCount = 0; tempCount < buddyCount; tempCount++)
            {
                int buddyId = node[count].getBuddyId(tempCount);
                double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                                node[count].getEstBuddyServiceTime(tempCount) *
                                node[buddyId - 1].getSpeedFactor();
                if(buddyLoad < avgLoad[count] * node[buddyId - 1].getSpeedFactor())
                {
                        availableCapacity += (avgLoad[count]*
                                node[buddyId - 1].getSpeedFactor() - buddyLoad);
                }
            }
            // amountOfLoadAcceptance array will indicate how many jobs can be transferred
              to buddy processor
            for(tempCount = 0; tempCount < buddyCount; tempCount++)
            {
                int buddyId = node[count].getBuddyId(tempCount);
                double buddyLoad = node[count].getEstBuddyLoad(tempCount) *
                node[count].getEstBuddyServiceTime(tempCount)*
                        node[buddyId - 1].getSpeedFactor();
                if(buddyLoad < avgLoad[count]*node[buddyId - 1].getSpeedFactor())
                {
                        double bCapacity = avgLoad[count]*
                                node[buddyId - 1].getSpeedFactor() - buddyLoad;
                        amountOfLoadAcceptance[count][buddyId - 1] = bCapacity *
                                extraLoad / availableCapacity;
                }
            }
        }
}
//Now tranfer load to buddy processor
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      double myLoad = node[count].getJobsWaitingInQueue() *
        node[count].getEstMeanServiceTime() * node[count].getSpeedFactor();
      if(myLoad > avgLoad[count] * node[count].getSpeedFactor())
      {
```

```
                double extraLoad = myLoad - avgLoad[count]* node[count].getSpeedFactor();
                for(tempCount = 0; tempCount < NUMBER_OF_PROCESSORS;
                    tempCount++)
                {
                    // if i can tranfer to buddy having ID tempCount, then tranfer load
                    Task *p = node[count].getNextTaskInQueue();
                    p = p->next;
                    if(amountOfLoadAcceptance[count][tempCount] > 0.5)
                    {
                        double bMeanArrivalTime =
                                    node[tempCount].getEstMeanArrivalTime();
                        double bMeanServiceTime =
                                    node[tempCount].getEstMeanServiceTime();
                        double bLoad = node[tempCount].getEstCurrentLoad();
                        double bSpeedFactor = node[tempCount].getSpeedFactor();
                        while(p != NULL)
                        {
                            p = node[count].checkForMigration(bMeanArrivalTime,
                                    bMeanServiceTime, bLoad,p,bSpeedFactor);
                            if(p == NULL)
                               break;
                            Task *q = p;
                            //remove task from waiting queue
                            p = node[count].removeTask(p,currentSystemTime);
                            double commCost = node[count].getCommSpeed(tempCount + 1);
                            double transferTime = q->getTaskSize() / commCost;
                            q->incrementNoOfTimesMigrated();
                            migration += 1;
                            if(max < q->getNoOfTimesMigrated())
                                 max = q->getNoOfTimesMigrated();
                            q->setArrivalTime(currentSystemTime + transferTime);
                            // insert task in buddy list after tranfer time
                            node[tempCount].insertMigrationTask(q);
                            amountOfLoadAcceptance[count][tempCount] -=
                                    node[count].getEstMeanServiceTime()*
                                    node[count].getSpeedFactor();
                            node[count].incrementEstBuddyLoad(tempCount + 1,
                                    node[count].getEstMeanServiceTime()*
                                    node[count].getSpeedFactor());
                            extraLoad -= node[count].getEstMeanServiceTime()*
                                    node[count].getSpeedFactor();
                            if(amountOfLoadAcceptance[count][tempCount] < 0.5 || extraLoad
                                    <= 0)
                                break;
                        }
                    }
                    if(p == NULL || extraLoad <= 0)
                        break;
                }
            }
        }
      nextTimePeriod = nextTimePeriod + EST_PERIOD;
    }
}
```

```
// All jobs processed, so print final values
cout << "\nJob finished " << jobFinished;
cout << "\nAvg Response Time " << totalResponseTime / jobFinished;
cout << "\nAvg Waiting Time " << totalWaitingTime / jobFinished;
cout << "\nTotal Execution Time " << currentSystemTime;
cout << "\n Number of migration " << migration;
cout <<"\n\nResource Utilization\tQueue Length\n";
double avgUtil = 0.0;
double max1 = 0, min = 1;
for(count = 0; count < NUMBER_OF_PROCESSORS; count++)
{
      if(node[count].getNodeStatus() == IDLE)
         node[count].incrementNodeIdleTime(currentSystemTime -
        node[count].getLastEventTime());
      double util = (currentSystemTime - node[count].getNodeIdleTime()) / currentSystemTime;
      cout << util;
      avgUtil += util;
      if(util > max1)
        max1 = util;
      if(util < min)
        min = util;
     cout << "\t" << node[count].getJobsWaitingInQueue() << "\n";
}
out << min << "\n" << max1 << "\n" << avgUtil / NUMBER_OF_PROCESSORS << "\n\n" <<
  totalEstimation<< "\n";
cout << "\nAverage Utilization " << avgUtil / NUMBER_OF_PROCESSORS ;
out.close();
cout << "\nMax Migration " << max;
cout << "\nTotal Estimation " << totalEstimation;
cout << "\n t = " << t;
getch();
return 0;
}
```