ENERGY-EFFICIENT RESOURCE MANAGEMENT FOR HIGH-PERFORMANCE COMPUTING PLATFORMS

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Ziliang Zong

Certificate of Approval:

Drew Hamilton Associate Professor Computer Science and Software Engineering	Xiao Qin, Chair Assistant Professor Computer Science and Software Engineering
Wei-Shinn Ku Assistant Professor Computer Science and Software Engineering	George T. Flowers Interim Dean Graduate School

ENERGY-EFFICIENT RESOURCE MANAGEMENT FOR HIGH-PERFORMANCE COMPUTING PLATFORMS

Ziliang Zong

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
August 9, 2008

ENERGY-EFFICIENT RESOURCE MANAGEMENT FOR

HIGH-PERFORMANCE COMPUTING PLATFORMS

Ziliang Zong

Signature of Author

Date of Graduation

DISSERTATION ABSTRACT

ENERGY-EFFICIENT RESOURCE MANAGEMENT FOR HIGH-PERFORMANCE COMPUTING PLATFORMS

Ziliang Zong

Doctor of Philosophy, August 9, 2008
(M.S. Shandong University, China, 2005)
(B.S. Shandong University, China, 2002)

151 Typed Pages

Directed by Xiao Qin

In the past decade, high-performance computing (HPC) platforms like clusters and computational grids have been widely used to solve challenging and rigorous engineering tasks in industry and scientific applications. Due to extremely high energy cost, reducing energy consumption has become a major concern in designing economical and environmentally friendly HPC infrastructures for many applications. In this dissertation, we first describe a general architecture for building energy-efficient HPC infrastructures, where energy-efficient techniques can be incorporated in each layer of the proposed architecture. Next, we developed an array of energy-efficient scheduling as well as energy-aware load balancing algorithms for high-performance clusters, computational grids, and large-scale storage systems. The primary goal of this dissertation research is to minimize energy consumption while maintaining reasonably high performance by incorporating energy-aware resource management techniques to HPC platforms. We have conducted extensive simulation experiments using both synthetic and real world applications to quantitatively evaluate both energy efficiency and performance of our proposed energy-efficient scheduling and load balancing strategies. Experimental results show that our approaches can reduce energy dissipation in HPC platforms without significantly degrading system performance.

ACKNOWLEDGMENTS

First of all, I would like to express my deep and sincere gratitude to my advisor, Dr. Xiao Qin. Without his wide knowledge, detailed and constructive guidance, generous support and warm encouragement, I would not have completed my PhD study within three years and this dissertation research would have never been possible. His passionate attitude towards research and wonderful personality will have a remarkable influence on my entire career.

I warmly thank Dr. Drew Hamilton and Dr. Wei-Shinn Ku for their valuable advice on my dissertation. The extensive discussions and their insightful comments have significantly helped in improving the quality of this dissertation. I also wish to express my warm and sincere thanks to Dr. Shiwen Mao for serving as the outside reader and proofreading my dissertation.

I also wish to extend my sincere thanks to all the members in our research group led by Dr. Qin. The group members and my friends have helped me and collaborated with me during my study in Auburn University. These research group members include Adam Manzanares, Xiaojun Ruan, Kiranmai Bellam, Tao Xie, and Mais Nijim.

I owe my loving thanks to my wife Shuo Wang, my daughter Elena Zong and my parents-in-law. They not only have been providing me with sufficient time to do research, but also giving me a happy family life in Auburn. It would have been impossible for me to complete this dissertation without their encouragement and understanding. I am deeply grateful to my parents. Without their consistent support and selfless love, I could not get the achievements today.

Style manual: IEEE Standard for Research Papers

Software used: Microsoft Word 2007, Microsoft Excel 2007, Linux GCC Compiler, Microsoft Visio 2007, Eclipse, Adobe Photoshop, C/C++/Java

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

With the advent of powerful processors, fast interconnects, and low-cost storage systems, high performance computing platforms like clusters, grids and large-scale storage systems have served as primary and cost-effective infrastructures for ever complicated scientific and commercial applications. Theses platforms provide powerful computing capability and the applications running in these platforms require intensive data processing and data storage capability in nature. Unfortunately, super-computing power is at the cost of huge energy consumption. How to generate enough power to support these high-performance computing platforms has become a serious problem.

We believe that an efficient way to alleviate the energy crisis caused by high-performance computing platforms is to design green computing techniques and apply these techniques to the super-computing platforms. The objective of this dissertation is to explore energy-efficient resource management technologies to reduce power consumption of high-performance computing platforms built in giant data centers.

This chapter first presents the problem statement in Section 1.1. In Section 1.2, we describe the scope of this research. Section 1.3 highlights the main contributions of this dissertation, and Section 1.4 outlines the dissertation organization.

1.1 Problem Statement

In this section, we start with an overview of new trends in high-performance computing. Section 1.1.2 introduces the serious data center energy crisis that we have to face today and presents the initial motivation for the dissertation research.

1.1.1 The Era of High-Performance Computing

We are now in an era of information explosion. Billions of data is generated in the moment you blink your eyes. In order to process these massive data, large-scale high-performance computing platforms have been widely deployed all over the world. These high-performance computing platforms usually are built in huge data centers. A large fraction of applications running in these high-performance computing platforms are computing-intensive and storage-intensive, since these applications deal with a large amount of data transferred either between memory and storage systems or among hundreds of computing nodes via interconnection networks. Nowadays, we can find the impact of high-performance computing data centers in almost every domain: financial services, scientific computing, bioinformatics, computational chemistry, and weather forecast etc. Without the support of high-performance computing platforms, the implementation of large-scale scientific and commercial projects like human genome sequence programs, universe dark matter observation and Google search engine is almost impossible. There is no doubt that data centers have significantly changed our lives. We are enjoying the great convenience and services provide by data centers every day.

1.1.2 The Data Center Energy Crisis

However, every sword cuts two sides. Increasing evidences have shown that the powerful computing capability of data centers is actually at the cost of huge energy consumption. For example, Energy User News stated that the power requirements of today's data centers range from 75 W/ft^2 to 150-200 W/ft^2 and will increase to 200-300 W/ft2 in the nearest future [1]. The new data center capacity projected for 2005 in U.S. would require approximately 40 TWh ($4B at $100 per MWh) per year to run 24x7 unless they become more efficient [2]. The supercomputing center in Seattle is forecast to increase the city's power demands by 25% [3]. As shown in Figure 1.1, the Environment Protection Agency reported that the total energy consumption of servers and data centers of the United States was 61.4 billion KWh in 2006, which is more than doubled the energy usage for the same purpose in 2000 [4]. Even worse, the EPA predicted that the power usage of servers and data centers will be doubled again within five years if the historical trends are followed [4]. However, most previous research about high-performance computing primarily focused on the improvement of performance, security, and reliability. Energy conservation issue was a forgotten corner. However, organizations of all sizes are currently experiencing significant challenges as a result of energy-related expenses within their data centers. For example, "The data center energy crisis is inhibiting our clients' business growth as they seek to access computing power. Many data centers have now reached full capacity, limiting a firm's ability to grow and make necessary capital investments," said Mike Daniels, senior vice president, IBM Global Technology Services. Our research is motivated by the energy consumption trend and the necessity of energy conservation for high-performance computing platforms.

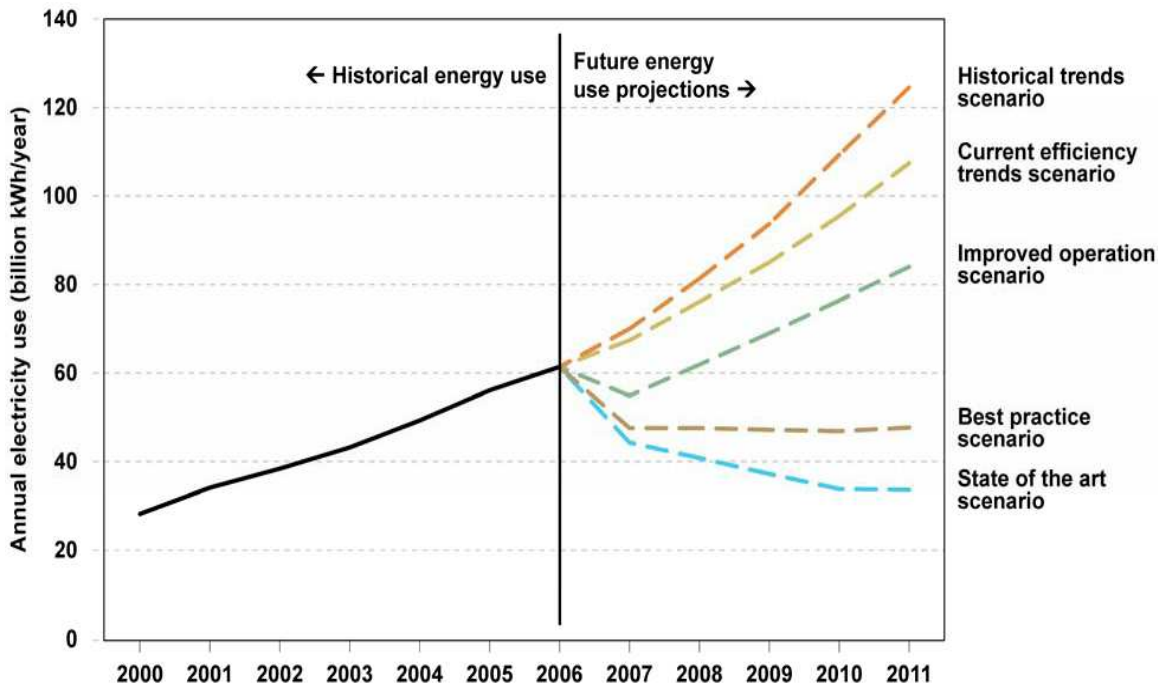


Figure 1.1 2007 EPA report to congress about U.S. data center power usage

1.2 Scope of Research

Our research is focusing on designing new energy-efficient techniques for data centers and incorporating existing techniques to conserve energy in high-performance computing platforms. Since CPUs, network interconnections and storage systems are three primary energy consumers in most high-performance computing platforms, our research focuses on conserving energy for CPUs, interconnections and storage systems.

More specific, the energy conservation for CPUs and interconnections are achieved through energy-efficient scheduling. A buffer disk based architecture (BUD for short) and energy-aware load balancing algorithm are proposed to build energy-efficient parallel storage systems.

1.3 Contributions

The major contributions of this research are summarized as follows:

(1) We propose a general architecture for large scale high-performance computing platforms and discuss the potential possibilities of incorporating energy-efficient techniques to each layer of the proposed architecture.

(2) We design and implement two energy-efficient scheduling algorithms for homogeneous cluster systems.

(3) We design and implement two energy-efficient scheduling algorithms for heterogeneous grid systems.

(4) We design energy-efficient buffer disk based architecture (BUD for short) for storage systems and implement the according energy-aware load balancing algorithm for BUD.

(5) We conduct extensive experiments for large scale clusters, grids, and storage systems. These experimental results could be used for other researchers in the research area of green computing.

1.4 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, related work in the literature is briefly reviewed.

In Chapter 3, we propose the high-performance computing platforms architecture and discuss the potential possibilities of incorporating energy-efficient techniques to each layer of the proposed architecture.

To make the architecture presented in Chapter 3 more practical, we develop two energy-efficient algorithms for parallel jobs running in clusters in Chapter 4.

In Chapter 5, we study the energy-efficient scheduling issue for heterogeneous grids.

In Chapter 6, a buffer disk based energy-efficient storage system is presented and its impact to performance and energy is evaluated.

In Chapter 7, we summarize the main contributions of this dissertation and discuss future directions for this research.

Chapter 2

Literature Review

In this chapter, we briefly summarize the previous literatures which are most relevant to our research in terms of energy-efficient resource management for high-performance computing platforms. Section 2.1 will introduce related work on energy-efficient parallel scheduling, which is highly relevant to our research shown in chapter 4 and 5. Related work on energy-efficient high-performance storage systems will be discussed in section 2.2. This part of related work is closely relevant to our research shown in chapter 6.

2.1 Related Work on Energy-Aware Scheduling

The issue of conserving energy consumption in clusters and grids did not attract enough attention for a long period because researchers primarily concentrate on the performance, reliability, and security issues [5]. Recently, people start to realize that the energy consumption issue is also critical since energy demands of clusters and grids have been steadily growing companied with an increasing number of data centers. However, designing energy-aware scheduling algorithms for homogeneous clusters, especially for heterogeneous grids, is technically challenging because we have to take into account multiple design objectives, including performance (measured by throughput and schedule length), energy efficiency, and heterogeneities.

2.1.1 Energy-Aware Scheduling in Clusters and Grids

A handful of previous studies investigated energy-aware processor and memory design techniques to reduce energy consumption in CPU and memory resources [6] [7] [8]. IBM researchers Elnozahy, Kistler, and Rajamony proposed the Request Batching Policy (RBP), in which servicing of incoming requests is delayed while a web server is kept in a low power state. Incoming requests are accumulated in memory until a request has been kept pending for longer than a specified batching timeout. RBP can save energy because while requests are being accumulated, the processor is placed in a lower power state such as deep sleep [9]. Dynamic power management is designed to achieve requested performance with minimum number of active components or a minimum load on such components [6] [10]. Dynamic power management consists of a collection of energy-efficient techniques, which adaptively turn off system components or reduce their performance when the component is idle or partially unexploited. For example, based on the observation of past idle and busy periods, predictive shutdown policies can make power management decisions when a new idle period starts [11] [12]. Shin and Choi proposed a scheme to slow down a processor when there is a single task eligible for execution [13]. Yao *et al.* developed a static off-line scheduling algorithm [14], whereas Hong *et al.* proposed on-line heuristics scheduling for aperiodic tasks [15]. T. Xie and X. Qin developed a task allocation strategy aiming to minimize overall energy consumption while confining schedule lengths to an ideal range [16].

However, the prior work in the arena of energy-aware scheduling was merely focused on energy consumed by processors. The communication energy consumption was completely ignored. The literature has shown that reducing energy dissipation in interconnects is critical important. For instance, interconnect consumes 33 percent of the total energy in an Avici switch [17] [18], and routers and links consume 37 percent of the total power budget in a Mellanox server blade [19]. The energy consumption in interconnects becomes even more critical for communication-intensive parallel applications, in which large number of data will be transferred among precedence constrained parallel tasks. One of the fundamental differences between our research and previous research is that we consider both CPU and network interconnection power consumption in the context of homogeneous and heterogeneous environment.

2.1.2 Task Partitioning and Task Scheduling

Task allocation strategies, which can be divided into task partitioning and scheduling strategies, play an important role in achieving high-performance for parallel applications on clusters and grids. The goal of a partitioning algorithm is to partition a parallel application into a set of precedence constrained tasks represented in the form of a directed acyclic graph (DAG), whereas a scheduling algorithm is deployed to schedule the DAG onto a set of homogeneous or heterogeneous computational nodes. Scheduling strategies deployed in clusters and grids have large impacts on overall system performance.

Allocation techniques can be generally classified into two types: static and dynamic schemes. The basic idea of static allocation schemes [20] [21] [22] [23] [24] is to assume prior knowledge of applications, including the component tasks, their execution times, and the like. Static allocation tries to find the overall optimized scheduling solution for given objectives at compile time, which is extremely expensive (NP-Complete Problem) in numerous complicated applications. In contrast, dynamic allocation strategies [25] [26] [27] [28], which are much less expensive, provide merely suboptimal results.

Scheduling policies can be generally classified into three categories: priority-based scheduling [29], group-based scheduling, and task-duplication based scheduling algorithms [30]. Priority-based scheduling algorithms involve assignments of priorities to tasks and then maps the tasks to computing nodes based upon assigned priorities. Group-based scheduling algorithms group intercommunicating tasks within a single computing node, thereby eliminating communication overheads [31]. The basic idea behind duplication-based scheduling algorithms is to make use of computing nodes' idle times to replicate predecessor tasks [30] [32]. Many researchers have demonstrated that various strategies regarding task duplications are extremely applicable for reducing total execution times under communication intensive workload conditions [32] [33]. In duplication-based scheduling strategies that exhibit performance improvements over other scheduling methods, redundantly executed tasks either eliminate communication overheads or allow productive utilization of idle processor times. Hagras and Janecek developed a simple yet efficient task-graph scheduling algorithm using the list-based and task-duplication-based scheduling approaches [34]. Siegel *et al.* investigated various mapping and scheduling algorithms in the context of heterogeneous ad hoc grids, where the algorithms are aimed to assign resources in a way to meet applications' execution time and energy constraints [35]. Kishimoto and Ichikawa carried out a case study, attempting to reduce the execution time of the high-performance linpack benchmark on two heterogeneous clusters [36]. Cuenca *et al.* proposed an approach to adapting an application implementing a homogeneous parallel dynamic programming algorithm for efficient execution on a heterogeneous cluster [37].

In our algorithms for grids, we try to seamlessly integrate static and dynamic allocation techniques to guarantee high-performance while conserving energy. Basically, our algorithms contain two phases. In the first phase, we apply a heuristic (a similar approach can be found in [5]) to minimize schedule lengths by clustering the most related parallel tasks together. The static allocation is carried out because we assume the execution and communication times of tasks are already known in priori. In the second phase, our algorithms make use of a dynamic allocation method to obtain an optimal power consumption of a grid computing system by comparing total energy consumption when grouped tasks are allocated to different computational nodes in the grids.

2.2 Related Work on Energy-Efficient Storage Systems

Modern parallel storage systems are able to provide higher performance at the cost of enormous energy consumption. For example, a typical robotic tape system provided by StorageTek would have an aggregate bandwidth of 1200MB/s [38] while a modern disk array could easily provide a peak bandwidth of 2,880,000MB/s. However, reading and storing 1,000TB of information would cost $9,400 to power the tape library system vs. $91,500(almost ten times) to power the disk array [39]. The gap will definitely increase when faster disks with higher power consumption rates appear and are widely deployed. A recent industry report shows that storage devices account for almost 27% of the total energy in a data center [40]. Even worse, this fraction tends to increase as storage requirements are rising by 60% annually [41]. Due to the preceding energy consumption trends, new technologies focused on the design of energy-efficient parallel storage systems are highly desirable.

Several techniques proposed to conserve energy in storage systems include dynamic power management schemes [42], power-aware cache management strategies [43], power-aware prefetching schemes [44], software-directed power management techniques [45], and multi-speed settings [46]. But so far, none of these techniques address the energy conservation and performance issue of buffer-disk based parallel storage systems.

In 2002, D. Colarelli and D. Grunwald presented a similar framework as compared to our BUD architecture. Their architecture was called “Massive Arrays of Idle Disks” or MAID [39]. However, two important problems remain unsolved in MAID. First, they did not clearly mention about the mapping structure of active drives and passive drives, i.e. which buffer disk should be chosen as the candidate to cache the data whenever there is a data miss. Second, they did not consider the load balancing issue, which very likely could lead to performance penalties.

Another framework similar to MAID, called Popular Data Concentration (PDC), was proposed by E. Pinheiro and R. Bianchini in 2004 [47]. The basic idea of PDC is to migrate data across disks according to frequency of access, or popularity. The goal is to lay data out in such a way that popular and unpopular data are stored on different disks. This layout leaves the disks that store unpopular data mostly idle, so that they can be transitioned to a low-power mode. However, PDC is a static offline algorithm. In some cases, it is impossible for the system to exactly know which data is popular and which is not. This is especially true for the ever-changing workload, in which some data is popular at a particular period but becomes unpopular the next period.

In contrast with both MAID and PDC, we implemented a heat-based algorithm to control data caching and data mapping between data disks and buffer disks in the BUD architecture. The heat-based algorithm was first proposed by P. Scheuermann, G. Weikum and P. Zabback in 1998 [48]. Their algorithm varies from our algorithm in the fact that they calculate the heat of data disks and apply the algorithm in the data partitioning stage. We calculate the heat of buffer disks and apply the algorithm in the data caching stage. They focus on how to partition data to improve throughput, while our focus is how to judiciously cache data to achieve load balancing.

2.3 Summary

The objective of this dissertation is to present energy-aware resource management strategies for high-performance computing platforms, which is based on previous research efforts in scheduling, load balancing and large-scale storage systems. This chapter overviewed a variety of existing techniques related to scheduling, load balancing and high-performance storage systems.

In the first part of this chapter, we discussed the relevant approaches for energy-aware task partitioning and scheduling for clusters and grids. In particular, we talked about the energy-aware techniques for CPU and memory, static and dynamic task allocation and three different scheduling strategies. Moreover, we briefly introduce the characteristics of our scheduling algorithms. In the second part, we surveyed existing energy-aware techniques used in high performance storage systems. These techniques include Massive Arrays of Idle Disks and Popular Data Concentration. In addition, we compare our heat-based algorithms for buffer disk architecture with these two existing algorithms.

Chapter 3

High-Performance Computing Platforms Architecture

In the previous chapter, we summarized the published literatures which are highly related to our research. However, during the course of literature review, we realized that almost all previous studies are in the lower level such as energy-aware scheduling, CPU energy efficiency and Memory energy efficiency etc. Although these works have made great contribution to build energy-aware high-performance computing platforms, comprehensive discussions in the architecture level was ignored.

We believe that the discussions in the architecture level are necessary and valuable because these discussions can help us understand the importance of energy-efficiency for high-performance computing platforms and provide a big picture of this research area. Meanwhile, it can provide meaningful guidance for the follow-up researchers. Therefore, in this chapter, we propose a general architecture for high-performance computing platforms and discuss the possibility of incorporating energy-efficient techniques to each layer of this architecture.

3.1 A General High-Performance Computing Platforms Architecture

Generally, most high-performance computing platforms can be presented by the following four layers: the application layer, the middleware layer, the resource layer and the network layer (See Figure 3.1). Since grid system is one of the most complicated high-performances computing platforms, we will use grids as an example to explain the proposed architecture.

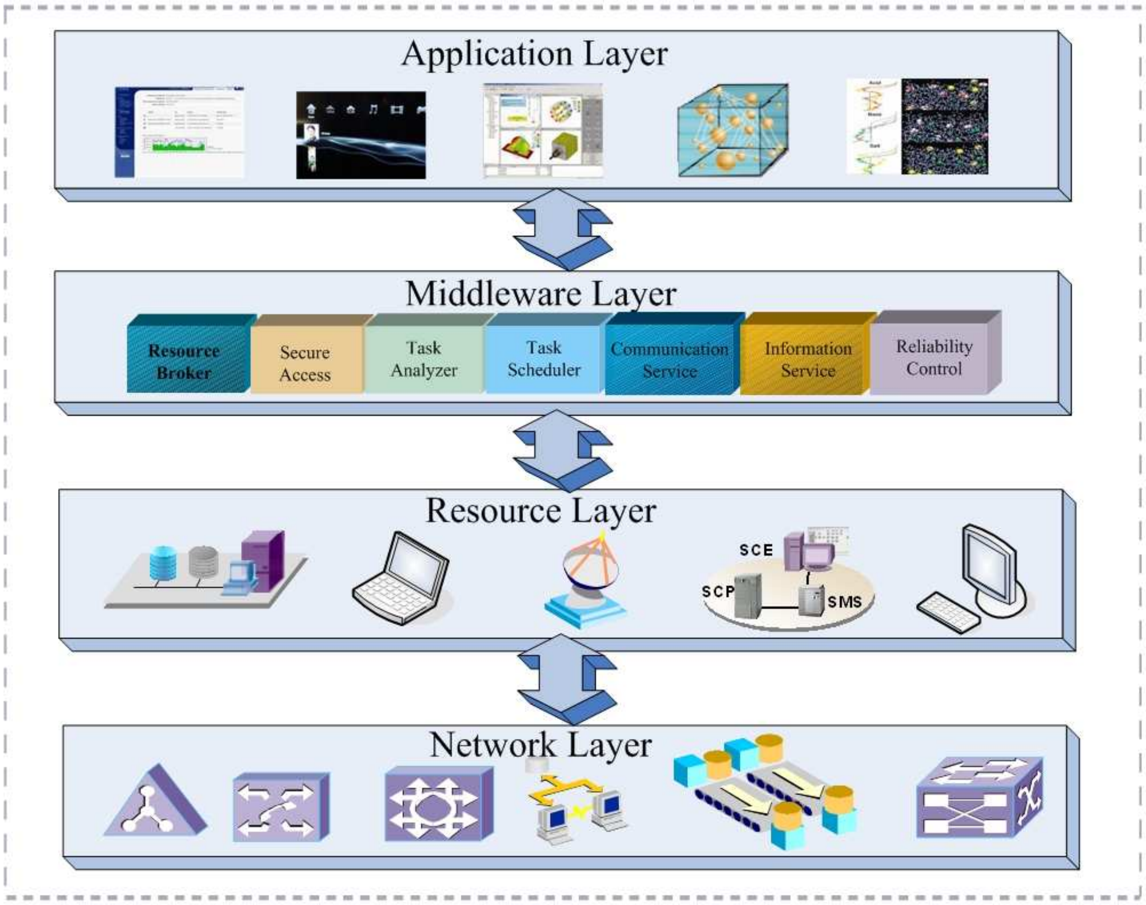


Figure 3.1 High-performance computing platforms architecture

The network layer is responsible for routing and transferring packets and it also has the responsibility of establishing network services for the resource layer. The dynamic network power management technique could be implemented in the network layer to support energy-efficient data transmission by deferring packet transmissions without violating any delay constraints.

On top of the network layer is a resource layer, which consists of a wide range of resources like computing nodes, storage systems, electronic data catalogues, and satellites or other instruments. The resource layer is responsible for manipulating the distributed resources in grid systems. In this layer, the dynamic voltage scaling techniques can be used to conserve energy for computing nodes by dynamically lowering supply voltages when the computing nodes are running faster than specified performance requirements.

Parallel applications running in a grid system do not directly interact with the resource layer. Instead, application programs interact with the middleware layer which provides a sophisticated means of reliability control, security protection, resource allocation, and task scheduling and analysis. The middleware layer contains a set of intelligent modules, including resource broker, security access, task analyzer, task scheduler, communication service, information service, and reliability control. The resource broker allows users to submit their applications to the grid system. The security module is responsible for providing security protection schemes to security-critical grid applications. After a grid job is admitted to the grid system, the task analyzer partitions the job into a number of small tasks with dependency constraints. Next, the task scheduler allocates the tasks to distributed computing resources using specific scheduling strategies. The communication service module has the responsibility for supporting services like remote function calls. The information service module keeps track of detailed information pertinent to the tasks' execution on computing resources. The reliable control module makes the grid system highly reliable and fault tolerant. For example, the reliable control module may reject a submitted job if the job's reliability requirements cannot be guaranteed by resources in the grid system. The middleware layer provides significant opportunities for incorporating energy-efficient techniques, especially for applying energy-efficient scheduling strategies. Our proposed scheduling algorithms in Chapter 4 and Chapter 5 are actually running in this layer.

The application layer handles all types of user applications varying from science, engineering, business, and financial area. Portals and development toolkits are provided to support various grid applications. Although energy-aware software applications are unusual today, they may become the next hotspot in the research area of software engineering with the emerging technology of multi-core microprocessors.

A number of energy efficiency trends for large scale servers and data centers are currently underway. For example, multi-core processors are expected to run at a slower speed and lower voltage but handle more work in parallel than a single-core chip thereby balancing energy efficiency and performance. Replacing several dedicated servers that operate at a low average processor utilization level with a single "host" server that operates at a higher average utilization level is another trend. Hard disk drive storage devices are also expected to become more energy-efficient in part because of a shift to smaller form factor disk drives and increasing use of serial advanced technology attachment drives. Meanwhile, the next generation of power supply systems and site infrastructure systems for grids will become more and more energy efficient. If these trends could be realized and the according techniques could be implemented in different layers, the energy usage caused by high performance computing platforms will be greatly reduced.

3.2 Summary

In this chapter, we have proposed a general architecture for high-performance computing platforms and discussed the possibility of incorporating energy-efficient techniques to each layer of this architecture.

To make this architecture more solid and sound, we will illustrate how to incorporate energy-efficient techniques to three typical high-performance computing platforms in the following three chapters. More specifically, Chapter 4 and Chapter 5 will illustrate energy-efficient scheduling for clusters and grids respectively. Chapter 6 will illustrate energy-efficient resource management for large-scale storage systems.

Chapter 4

Energy-Efficient Scheduling For Clusters

In this chapter, we consider the problem of building energy-efficient cluster systems. A cluster is a type of parallel processing system, which consists of a collection of interconnected stand-alone computers cooperatively working together as a single, integrated computing system (see Figure 4.1). All these loosely coupled computers do not have common memory. They communicate with each other by passing messages.

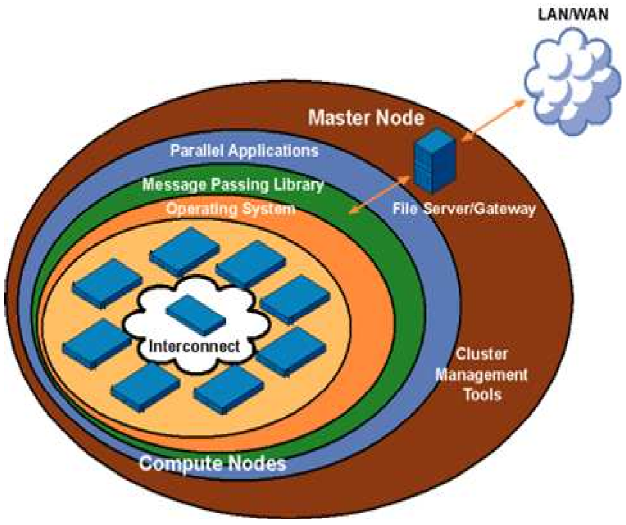


Figure 4.1 System model of high-performance clusters (source: Wikipedia)

When we talk about cluster systems, we have to mention about the parallel computing technologies. Parallel computing is the simultaneous execution of small tasks split up from a complicated application and specially allocated on multiple processors in order to obtain results faster. The combination of cluster systems and parallel computing technology exhibits powerful computing capabilities. Over the last decade, the rapid advancement of high-performance microprocessors, high-speed networks, and standard middleware tools makes cluster computing platforms more powerful and convenient to use. Therefore, cluster computing technology has been extensively deployed and widely used to solve challenging and rigorous engineering problems in industry and scientific areas like molecular design, weather modeling, database systems, universe dark matter observations, and complex image rendering. However, the rapid growth of cluster computing centers introduces a serious problem: excessively high energy consumption. To address this problem, we propose two energy-efficient scheduling algorithms in this chapter for parallel applications running on clusters. The two algorithms are named the Energy-Aware Duplication scheduling algorithm (or EAD for short) and the Performance-Energy Balanced Duplication scheduling algorithm (or PEBD for short).

This chapter is organized as follows. In section 4.1, we introduce the mathematical models used to present cluster systems, including cluster model, parallel tasks model, and energy consumption model. In section 4.2, we present the energy-efficient scheduling algorithms and illustrate how the EAD and PEBD algorithms work using a concrete example. Next, we will prove the time complexity of our algorithms in section 4.3. Experimental environment and simulation results are shown in section 4.4. Finally, section 4.5 concludes this chapter by summarizing the main contributions of the chapter.

4.1 System Models

In this section, we describe mathematical models used to represent clusters, precedence constrained parallel tasks, and energy consumption in CPUs and interconnects.

4.1.1 Cluster Model

A computer cluster is a group of coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. A cluster in our research is characterized by a set $P = \{p_1, p_2,..., p_m\}$ of computational nodes (hereinafter referred to as nodes) connected by a Myrinet-style cluster interconnects. It is assumed that the computational nodes are homogeneous in nature, meaning that all processors are identical in their capabilities. Similarly, the underlying interconnection is assumed to be homogeneous and, thus, communication overhead of a message with fixed data size between any pair of nodes is considered to be the same. Each node communicates with other nodes through message passing, and the communication time between two precedence constrained tasks assigned to the same node is negligible. In our system model, computation and communication can take place simultaneously. This assumption is reasonable because each computational node in a modern cluster has a communication coprocessor that can be used to free the processor in the node from communication tasks.

To simply the system model without loss of generality, we assume that the cluster system is fault free and the page fault service time of each task is integrated into its execution time. With respect to energy conservation, energy consumption rate of each node in the system is measured by Joule per unit time. Each interconnection link is characterized by its energy consumption rate that heavily relies on data size and the transmission rate of the link.

4.1.2 Parallel Tasks Model

A parallel application with a set of precedence-constrained tasks is represented in the form of a *Directed Acyclic Graph* (DAG), which throughout this paper is modeled as a pair (V, E). $V = \{v_1, v_2, ..., v_n\}$ represents a set of precedence constrained parallel tasks, and t_i is the *i*th task's computation requirement showing the number of time units to compute v_i, $0 \leq i \leq 1$. It is assumed that all the tasks in V are non-preemptive and indivisible work units, and a similar assumption can be found in related studies [13][49]. E denotes a set of messages representing communications and precedence constraints among parallel tasks. Thus, $e_{ij} = (v_i, v_j) \in E$ is a message transmitted from task v_i to v_j, and c_{ij} is the communication cost of the message $e_{ij} \in E$. We assume in this study that there is one entry task and one exit task for an application with a set of precedence-constrained tasks. The assumption is reasonable because in case of multiple entry or exit tasks exist, the multiple tasks can always be connected through a dummy task with zero computation cost and zero communication cost messages.

The communication-to-computation ratio or CCR of a parallel application is defined as the ratio between the average communication cost and the average computation cost of the application on a given cluster. Formally, the CCR of an application (V, E) is given by the Eq. (1):

$$CCR(V,E)=\frac{\frac{1}{|E|}\sum_{e_{ij}\in E}c_{ij}}{\frac{1}{|V|}\sum_{i=1}^{|V|}t_i}. \tag{1}$$

A task allocation matrix (e.g., *X*) is an $n\times m$ binary matrix reflecting a mapping of *n* precedence constrained parallel tasks to *m* computational nodes in a cluster. Element x_{ij} in *X* is "1" if task v_i is assigned to node p_j and is "0", otherwise.

4.1.3 Energy Consumption Model

We use a bottom-up approach to derive energy dissipation experienced by a parallel application running on a cluster. In this subsection, we first model energy consumption exhibited by computational nodes in the cluster. Next, we calculate energy dissipation in the interconnection network of the cluster.

Let en_i be the energy consumption caused by task v_i running on a computational node, of which the energy consumption rate is PN_{active}, and the energy dissipation of task v_i can be expressed as Eq. (2)

$$en_i = PN_{active}\times t_i. \tag{2}$$

Given a parallel application with a task set *V* and allocation matrix *X*, we can calculate the energy consumed by all the tasks in *V* using Eq. (3).

$$\begin{aligned}EN_{active} &= \sum_{i=1}^{|V|}en_i = \sum_{i=1}^{n}\left(PN_{active}\cdot t_i\right)\\ &= PN_{active}\sum_{i=1}^{n}t_i.\end{aligned} \tag{3}$$

Let PN_{idle} be the energy consumption rate of a computational node when it is inactive, and f_i be the completion time of task t_i. The energy consumed by an inactive node is a product of the idle energy consumption rate PN_{idle} and an idle period. Thus, we can use Eq. (4) to obtain the energy consumed by the *j*th computational node in a cluster when the node is sitting idle.

$$EN_{idle}^{j} = PN_{idle} \cdot \left(\max_{i=1}^{n} (f_i) - \sum_{i=1}^{n} (x_{ij} \cdot t_i) \right) \tag{4}$$

where $\max_{i=1}^{n}(f_i)$ is the schedule length (also known as makespan time), and $\max_{i=1}^{n}(f_i) - \sum_{i=1}^{n} x_{ij} \cdot t_i$ is the total idle time on the *j*th node. The total energy consumption of all the idle nodes cluster is

$$\begin{aligned} EN_{idle} &= \sum_{j=1}^{m} en_{idle}^{j} = PN_{idle} \cdot \sum_{j=1}^{m} \left(\max_{i=1}^{n} (f_i) - \sum_{i=1}^{n} (x_{ij} \cdot t_i) \right) \\ &= PN_{idle} \cdot \left(m \cdot \max_{i=1}^{n} (f_i) - \sum_{j=1}^{m} \sum_{i=1}^{n} (x_{ij} \cdot t_i) \right). \end{aligned} \tag{5}$$

Consequently, the total energy consumption of the parallel application running on the cluster can be derived from Eqs. (3) and (5) as

$$\begin{aligned} EN &= EN_{active} + EN_{idle} \\ &= PN_{active} \sum_{i=1}^{n} t_i + PN_{idle} \cdot \left(m \cdot \max_{i=1}^{n} (f_i) - \sum_{j=1}^{m} \sum_{i=1}^{n} (x_{ij} \cdot t_i) \right). \end{aligned} \tag{6}$$

We denote el_{ij} as the energy consumed by the transmission of message $(t_i, t_j) \in E$. We can compute the energy consumption of the message as a product of its communication cost and the power PL_{active} of the link when it is active:

$$el_{ij} = PL_{active} \times c_{ij} \tag{7}$$

The cluster interconnect in this study is homogeneous, which implies that all messages are transmitted over the interconnection network at the same transmission rate. The energy consumed by a network link between p_a and p_b is a cumulative energy consumption caused by all messages transmitted over the link. Therefore, the link's energy consumption is obtained by Eq. (8) as follows, where L_{ab} is a set of messages delivered on the link, and L_{ab} can be expressed as $L_{ab} = \{\forall e_{ij} \in E, 1 \le a, b \le m \mid x_{ia} = 1 \wedge x_{jb} = 1\}$.

$$\begin{aligned} EL_{active}^{ab} &= \sum_{e_{ij} \in L_{ab}} el_{ij} = \sum_{e_{ij} \in L_{ab}} \left(PL_{active} \cdot c_{ij}\right) \\ &= \sum_{i=1}^{n} \sum_{j=1, j \ne i}^{n} \left(x_{ia} \cdot x_{jb} \cdot PL_{active} \cdot c_{ij}\right), \end{aligned} \tag{8}$$

The energy consumption of the whole interconnection network is derived from Eq. (8) as the summation of all the links' energy consumption. Thus, we have

$$EL_{active} = \sum_{a=1}^{m} \sum_{b=1, b \ne a}^{m} EL_{active}^{ab}$$

$$= \sum_{i=1}^{n} \sum_{j=1, j \ne i}^{n} \sum_{a=1}^{m} \sum_{b=1, b \ne a}^{m} \left(x_{ia} \cdot x_{jb} \cdot PL_{active} \cdot c_{ij}\right). \tag{9}$$

We can express energy consumed by a link when it is inactive as a product of the consumption rate and the idle period of the link. Thus, we have

$$EL_{idle}^{ab} = PL_{idle} \cdot \left(\max_{i}^{n} (f_i) - \sum_{i=1}^{n} \sum_{j=1, j \ne i}^{n} \left(x_{ia} \cdot x_{jb} \cdot c_{ij}\right) \right) \tag{10}$$

where PL_{idle} is the power of the link when it is inactive, and $\max_{i}^{n} (f_i) - \sum_{i=1}^{n} \sum_{j=1, j \ne i}^{n} \left(x_{ia} \cdot x_{jb} \cdot c_{ij}\right)$ is the total idle time of the link. We can express energy incurred by the whole interconnection network during the idle periods as

$$EL_{idle} = \sum_{a=1}^{m} \sum_{b=1,b\neq a}^{m} EL_{idle}^{ab} = \sum_{a=1}^{m} \sum_{b=1,b\neq a}^{m} PL_{idle} \left(\max_{i}^{n}(f_i) - \sum_{i=1}^{n} \sum_{j=1,j\neq i}^{n} (x_{ia} \cdot x_{jb} \cdot c_{ij}) \right) \cdot \quad (11)$$

Total energy consumption exhibited by the cluster interconnect is derived from Eqs. (9) and (11) as

$$EL = EL_{active} + EL_{idle}, \quad (12)$$

Now, we can compute energy dissipation experienced by a parallel application on a cluster using Eqs. (6) and (12). Hence, we can express the total energy consumption of the cluster executing the application as

$$E = EN + EL = PN_{active} \sum_{i=1}^{n} t_i + PN_{idle} \cdot \left(m \cdot \max_{i=1}^{n}(f_i) - \sum_{j=1}^{m} \sum_{i=1}^{n} (x_{ij} \cdot t_i) \right) \quad (13)$$

$$+ \sum_{i=1}^{n} \sum_{j=1,j\neq i}^{n} \sum_{a=1}^{m} \sum_{b=1,b\neq a}^{m} (x_{ia} \cdot x_{jb} \cdot PL_{active} \cdot c_{ij}) + \sum_{a=1}^{m} \sum_{b=1,b\neq a}^{m} PL_{idle} \left(\max_{i}^{n}(f_i) - \sum_{i=1}^{n} \sum_{j=1,j\neq i}^{n} (x_{ia} \cdot x_{jb} \cdot c_{ij}) \right).$$

4.2 Energy-Efficient Scheduling Algorithms

In this section, we present two energy-aware scheduling algorithms for parallel applications with precedence constraints running on clusters. The two algorithms are named the Energy-Aware Duplication scheduling algorithm (or EAD for short) and the Performance-Energy Balanced Duplication scheduling algorithm (or PEBD for short). The objective of the two scheduling algorithms is to shorten schedule lengths while optimizing energy consumption of clusters. Theoretically, the scheduling problem for clusters is NP-hard problem because it could be mapped to a scheduling problem proven to be an NP-complete [50]. Therefore, the proposed two scheduling algorithms are heuristic in the sense that they can produce suboptimal solutions in polynomial-time. The EAD and PEBD algorithms consist of three major steps delineated in sections 4.2.1 -- 4.2.3.

4.2.1 Original Task Sequence Generation

Precedence constraints of a set of parallel tasks have to be guaranteed by executing predecessor tasks before successor tasks. To achieve this goal, the first step in our algorithms is to construct an ordered task sequence using the concept of level, which of each task is defined as the length in computation time of the longest path from the task to the exit task. There are alternative ways to generate the task sequence for a DAG, including critical path-based priority schemes [30] and other priority-based schemes [51]. In this study, we use a similar approach as proposed by Srinivasan and Jha [5] to define the level *L(v$_i$)* of task *v$_i$* as below

$$L(v_i) = \begin{cases} t_i, & \text{if successor(i)} = \Phi \\ \max\underbrace{(level(k))}_{k \in succ(i)} + t_i & otherwise \end{cases} \cdot \tag{14}$$

The levels of the tasks which have no successor are equal to their execution time. The levels of other tasks can be obtained in a bottom-up fashion by specifying the level of the exit task as its execution time and then recursively applying the second term on the right-hand side of Eq. (14) to calculate the levels of all the other tasks. Next, all the tasks are placed in a queue in an increasing order of the levels.

4.2.2 Duplication Parameters Calculation

The second phase in the EAD and PEBD algorithms is to calculate some important parameters, which the algorithms rely on. The important notation and parameters are listed in Table 4.1.

Table 4.1 Important notations and parameters

Notation	Definition
EST(v_i)	Earliest start time of task v_i
ECT(v_i)	Earliest completion time of task vi
FP(v_i)	Favorite predecessor of task vi
LACT(v_i)	Latest allowable completion time of task vi
LAST(v_i)	Latest allowable start time of task vi

The earliest start time of the entry task is 0 (see the first term on the right side of Eq. (15). The earliest start times of all the other tasks can be calculated in a top-down manner by recursively applying the second term on the right side of Eq. (15).

$$EST(v_i) = \begin{cases} 0, & \text{if predecessor(i)} = \Phi \\ \min\limits_{e_{ji}\in E}\left(\max\limits_{e_{ki}\in E, v_k \neq v_j} \left(ECT(v_j), ECT(v_k) + c_{ki} \right) \right), & \text{otherwise} \end{cases}. \quad (15)$$

The earliest completion time of task v_i is expressed as the summation of its earliest start time and execution time. Thus, we have

$$ECT(v_i) = EST(v_i) + t_i. \quad (16)$$

Allocating task v_i and its favorite predecessor *FP*(v_i) on the same computational node can lead to a shorter schedule length. As such, the favorite predecessor *FP*(v_i) is defined as below

$$FP(v_i) = v_j, \text{where } \forall e_{ji} \in E, e_{ki} \in E, j \neq k \left| ECT(v_j) + c_{ji} \geq ECT(v_k) + c_{ki} \right. .$$

(17)

As shown by the first term on the right-hand side of Eq. (18), the latest allowable completion time of the exit task equals to its earliest completion time. The latest allowable completion times of all the other tasks are calculated in a top-down manner by recursively applying the second term on the right-hand side of Eq. (18).

$$LACT(v_i) = \begin{cases} ECT(v_i), & \text{if successor}(i) = \Phi \\ \min\left(\min\limits_{e_{ij} \in E, v_i \neq FP(v_j)} \left(LAST(v_j) - c_{ij} \right), \min\limits_{e_{ij} \in E, v_i = FP(v_j)} \left(LAST(v_j) \right) \right), & \text{otherwise} \end{cases} . \quad (18)$$

The latest allowable start time of task v_i is derived from its latest allowable completion time and execution time. Hence, the *LAST*(v_i) can be written as

$$LAST(v_i) = LACT(v_i) - t_i. \quad (19)$$

4.2.3 Energy-Efficient Scheduling: EAD and PEBD

Given a parallel application presented in form of a DAG, the EAD algorithm in this phase allocates each parallel task to a computational node in a way to aggressively shorten the schedule length of the DAG while conserving energy consumption. The pseudocode in Figure 4.2 shows the details of this phase in the EAD algorithm, which aims to provide the greatest energy savings when it reaches the point to duplicate a task. Most existing duplication-based scheduling schemes merely optimize schedule lengths without addressing the issue of energy conservation. As such, the existing duplication-based approaches tend to yield minimized schedule lengths at the cost of high energy consumption. To make tradeoffs between energy savings and schedule lengths, we design the EAD algorithm in which task duplications are strictly forbidden if the duplications do not exhibit energy conservation (see Steps 9-10). In other words, duplications are not allowed if they result in a significant increase in energy consumption (e.g., the increase exceeds a threshold) and, are avoided in EAD. Consequently, the EAD algorithm ensures that schedule lengths are minimized using task duplication without adversely affecting energy conservation.

```
1.  v = first waiting task of scheduling queue;
2.  i = 0;
3.  assign v to P_i;
4.  while (not all tasks are allocated to computational nodes) do
5.    u = FP(v);
6.    if (u has already been assigned to another processor) then
7.      if (LAST(v) - LACT(u)<c_uv) then /* if duplicate u, we can shorten the schedule length */
8.        moreenergy = en_u – el_uv; /*energy increase*/
9.        if (moreenergy ≤ threshold h) then /* increased energy less than our threshold*/
10.         assign u to P_i; /*duplicate u*/
11.         if v has another predecessor z ≠ u has not yet been allocated to any node then
12.           u = z;
13.         else
14.           if u is entry task then
15.             u = the next task that has not yet been assigned to a node;
16.             i++;
17.       else
18.         for another predecessor z of v, z ≠ u,
19.         if (ECT(u)+cc_uv = ECT(z) + cc_zv) and z hasn't been allocated) then
20.           u = z; /* do not duplicate*/
21.     else
22.       for another predecessor z of v, z≠ u,
23.       if (ECT(u)+cc_uv = ECT(z) + cc_zv) and z hasn't been allocated) then
24.         u = z; /* do not duplicate*/
25.   else allocate u to P_i;
26.   v = u;
27.   if v is entry task then
28.     v = the next task that has not yet been allocated to a computational node;
29.     i++;
30.     assign v to P_i;
31. return schedule list;
```

Figure 4.2 Pseudo code of phase 3 in the EAD algorithm

Before this phase starts, phase 1 sorts all the tasks in a waiting queue, followed by phase 2 to calculate the important parameters. In phase 3 EAD strives to group communication-intensive parallel tasks together and have them allocated to the same computational node. Once multiple task groups are constructed, each group of tasks is assigned to a different node in the cluster. The process of grouping tasks is repeated from the first task in the queue by performing a depth-first style search, which traces the path from the first task to the entry task. Steps 5 and 6 choose a favorite predecessor if it has not been allocated a computational node. Otherwise, EAD may or may not replicate the favorite predecessor on the current node. For example, we assume that vj is the favorite predecessor of the current task vi, and vj has been allocated to another node. If duplicating vj on the current node to which vi is allocated can improve performance without sacrificing energy conservation, Step 12 makes a duplication of vj.

Please note that the generation of a task group terminates once the path reaches the entry task. The next task group starts from the first unassigned task in the queue. If all tasks are assigned to the computation nodes, then the EAD algorithm terminates.

The third phase of the PEBD algorithm is similar as that of EAD except that PEBD seamlessly integrate the approach to minimizing schedule lengths with the process of energy optimization (see Figure 4.3). Unlike EAD, the development of PEBD is motivated by the needs of making the right tradeoff between performance and energy conservation. Thus, the PEBD algorithm is geared to efficiently reduce schedule lengths while providing the greatest energy savings. Energy consumption incurred by duplicating a task involves judging whether the duplication is profitable or not. To facilitate the construction of PEBD, we introduce a concept of cost ratio of a duplication, which is defined as the ratio between the energy saving and schedule length reduction (see Step 10). While the energy saving of the duplication is obtained in Step 8, the reduction in schedule length is computed in Step 9. The PEBD algorithm is, of course, conducive to maintaining cost ratios at a low level, thereby efficiently shortening schedule lengths with low energy consumption. This feature is accomplished by Steps 11-12, which duplicate a task in case the cost ratio of such duplication is smaller than a given threshold.

```
1.  $v$ = first waiting task of scheduling queue;
2.  $i = 0$;
3.  assign $v$ to $P_i$;
4.  **while** (not all tasks are allocated to computational nodes) **do**
5.    $u = FP(v)$;
6.    **if** ($u$ has already been assigned to another node) **then**
7.      **if** ($LAST(v) - LACT(u) < c_{uv}$) **then** /* if duplicate $u$, we can shorten the execution
    time*/
8.        moreenergy = $en_u - el_{uv}$; /*energy increase*/
9.        lesstime = $LACT(u) + c_{uv} - LAST(v)$; /* schedule length is reduced */
10.       cost ratio = moreenergy / lesstime;    /*value of ratio: the smaller the better*/
11.       **if** (ratio $\leq$ threshold $h$) **then** /* significantly shorten schedule length */
12.         assign $u$ to $P_i$;  /*duplicate $u$*/
13.         **if** $v$ has another predecessor $v \neq u$ has not yet been assigned to any node **then**
14.           $u = v$;
15.         **else**
16.           **if** $u$ is entry task **then**
17.             $u$ = the next task that has not yet been allocated to a computational node;
18.             $i$++;
19.       **else**
20.         for another predecessor $z$ of $v$, $z \neq u$,
21.         **if** ($ECT(u)+cc_{uv} = ECT(z) + cc_{zv}$) and $z$ has not been allocated) **then**
22.           $u = z$; /*do not duplicate*/
23.     **else**
24.        for another predecessor $z$ of $v$, $z \neq u$,
25.        **if** ($ECT(u)+cc_{uv} = ECT(z) + cc_{zv}$) and $z$ has not been allocated) **then**
26.          $u = z$; /*do not duplicate*/
27.   **else** assign $u$ to $P_i$;
28.   $v = u$;
29.   **if** $v$ is entry task **then**
30.     $v$ = the next task that has not yet been allocated;
31.     $i$++;
32.     allocate $v$ to $P_i$;
33. **return** schedule list
```

Figure 4.3 Pseudo code of phase 3 in the PEBD algorithm

4.2.4 A Case Study

Now we run the proposed scheduling algorithms using a sample task graph delineated in Figure 4.4. In this example, we choose Intel Core2 Duo E6300 as the CPU of each computing node and high-speed Merynet as interconnection. Recall that the energy consumption of the task graph is determined by Eq. (13), where PN_{active} and PL_{active} are set to 44W and 33.6W, respectively.

In the task DAG plotted in Figure 4.4, each task is represented by (en_i, t_i) and each message is denoted by (el_{ij}, c_{ij}). Recall that en_i and el_{ij}, computed by Eqs. (2) and (7), are the energy consumption of task v_i and communication between task v_i and v_j. The running trace of EAD and PEBD is given as follows:

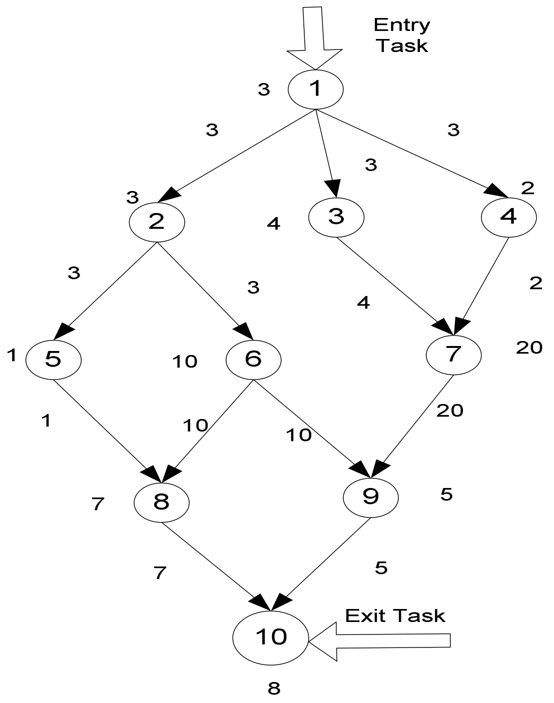


Figure 4.4 A typical DAG

Phase 1. Generate a task sequence by computing levels: The levels of tasks can be calculated using Eq. (14). For instance, the level of task v_{10} is 8, since v_{10} is the exit task without any successor. The level of v_8 is 8 + 7 = 15 because v_8 has only one successor task. The level of task v_2 is max{$L(v_5)$ + 3, $L(v_6)$ + 3} = 28, since v_2 has two successors - v_5 and v_6. All the tasks are placed in a queue in the non-increasing order of levels. Thus, we have a list of tasks as {10, 9, 8, 5, 6, 2, 7, 4, 3, 1}

Phase 2. Calculate important parameters:

Phase 2.1 Compute *EST* and *ECT* : The *EST* and *ECT* values of each task can be computed by applying Eqs. (15) and (16). For example, task v_1 is the entry task and, therefore, $EST(v_1) = 0$. In accordance with Eq. 16, we have $ECT(v_1) = 0 + t_1 = 3$. Since v_2, v_3, and v_4 are unable to start until v_1 finishes and, thus, we have $\mathrm{EST}(v_2) = \mathrm{EST}(v_3) = \mathrm{EST}(v_4) = \mathrm{ECT}(v_1) = 3$. Similarly, EST of v7 is computed as below

$$\begin{aligned}\mathrm{EST}(\mathrm{v}_7) &= \min\{\max(\mathrm{ECT}(\mathrm{v}_4), \mathrm{ECT}(\mathrm{v}_3) + c_{37}), \max(\mathrm{ECT}(\mathrm{v}_3), \mathrm{ECT}(\mathrm{v}_4) + c_{47})\} \\ &= \min\{\max(5, 7+4), \max(7, 5+2)\} = 7.\end{aligned}$$

Correspondingly, the ECT of v7 is $\mathrm{ECT}(v_7) = \mathrm{EST}(v_7) + t_7 = 7 + 20 = 27$.

Phase 2.2 Compute favorite predecessors: The favorite predecessor of a task is determined by using Eq. (17). For example, the favorite predecessor of task v_2, v_3, and v_4 is v_1, simply because these three tasks have only one predecessor. The favorite predecessor of v_8 is v_6 because $ECT(v_6) + c_{68} = 16 + 10 = 26 > ECT(v_5) + c_{58} = 7 + 1 = 8$.

Phase 2.3 Compute LAST and LACT: The LACT and ECT values of the exit task v_{10} equal to 40 and, thus, we have $LAST(v_{10}) = LACT(v_{10}) - t_{10} = 40 – 8 = 32$. In case of $\mathrm{LACT}(v_6)$, we have to consider two successors, namely, v_8 (not in critical path) and v_9

(in critical path). We obtain

$\mathrm{LACT}(v_6) = \min\{\min(\mathrm{LAST}(v_9) - c_{69}, \min(\mathrm{LAST}(v_8)))\} = \min\{(27-10), 18)\} = 17$ and

$LAST(v_6) = LACT(v_6) - t_6 = 17 – 10 = 7$

Table 4.2 shows the final results of all important parameters.

Table 4.2 Final results of parameters

Task	level	est	ect	last	lact	fpred
1	40	0	3	0	3	--
2	28	3	6	4	7	1
3	37	3	7	3	7	1
4	35	3	5	3	5	1
5	16	6	7	16	17	2
6	25	6	16	7	17	2
7	33	7	27	7	27	3
8	15	16	23	18	25	6
9	13	27	32	27	32	7
10	8	32	40	32	40	9

Phase 3. Task allocation and duplication phase:

The EAD algorithm. Given a threshold h = 25, EAD generates the first group of tasks by starting from the first task in the task list obtained in Phase 1. The first task group containing tasks v_1, v_3, v_7, v_9, and v_{10} is allocated to node 1. Next, EAD attempts to allocate the first unassigned task in the list. In this case, the unassigned task is task v_8. Tasks v_8, v_6 and v_2 are allocated to node 2, and the next task to be assigned is task v_1. Since v_1 has been allocated to node 1, EAD has to decide whether there is an incentive to duplicate v_1 on node 2. The condition in step 7 (see Figure 4.2) is satisfied, because we have $LAST(v_2) - LACT(v_1) = 4 – 3 = 1 < cc_{12} = 3$. Therefore, duplicating v1 on node 2 can shorten the schedule length. However, the increase in energy consumption is $en_1 – el_{12}$ = 44w×3 – 33.6w×3 = 31.2J (see step 8 in Figure 4.2), which is greater than the threshold. Thus, there is no any incentive to duplicate the task due to the high energy overhead, signifying that the duplication of v_1 must be avoided. EAD assigns task v_5 to node 3, followed by task v_2, and v_1, which are not duplicated on node 3 because we can not shorten the schedule length ($LAST(v_5)$ - $LACT(v2)=16\text{-}7=9> cc_{25}=3$). Task v_4 is the only task allocated on node 4, and v1 is not duplicated because the increase in energy consumption is significant.

Therefore, the final scheduling decision of EAD is as follows:

Processor 1: Task 10→ Task 9→ Task 7→ Task 3→ Task 1
Processor 2: Task 8→ Task 6→ Task 2
Processor 3: Task 5
Processor 4: Task 4

The PEBD algorithm. The behavior of PEBD is similar to that of EAD except that energy-performance tradeoffs are determined by a ratio between the energy consumption of replicas and the decrease in schedule length by virtue of replicas. Given a threshold h = 25, PEBD first allocates v_1, v_3, v_7, v_9, and v_{10} to node 1 and then it will meet the same situation as EAD, in which PEBD has to decide whether or not to duplicate v_1. Once again, PEBD will calculate $LAST(v_2)$ - $LACT(v_1)$ = 4 – 3 = 1 < cc_{12} = 3. Thus, if duplicate T1, the scheduling length can be shortened by 2 seconds. However, the energy consumption will be increased by $en_1 - el_{12}$ = 44w×3 – 33.6w×3 = 31.2J. Now PEBD will decide based on the result of ratio (31.2/2 =15.6<Threshold=25) to duplicate T1. The duplication of v_1 is made possible by PEBD because the replica helps in reducing the schedule length without significantly increasing energy consumption. And then, in the next iteration, EAD assigns task v_5 to node 3, followed by task v_2, and v_1, which are not duplicated on node 3 because we cannot shorten the schedule length ($LAST(v_5)$ - $LACT(v2)=16\text{-}7=9> cc_{25}=3$). The final scheduling decision of PEBD is:

Processor 1: Task 10→ Task 9→ Task 7→ Task 3→ Task 1
Processor 2: Task 8→ Task 6→ Task 2→ Task 1
Processor 3: Task 5
Processor 4: Task 4→ Task 1

4.3 Time Complexity Analysis

In this subsection, we will analyze the time complexity of the EAD and PEBD algorithms.

Theorem 1. The time complexity of EAD and PEBD is $O(|V|^2)$.

Proof. The EAD and PEBD algorithms perform the three main phases respectively described in Sections 4.2. In the first phase, EAD and PEBD traverse all the tasks of the DAG to compute the levels of the tasks. The time complexity to calculate the levels is $O(|E|)$, where $|E|$ is the number of messages. This is because all the messages have to be examined in the worst case. It takes $O(|V|\log|V|)$ time to sort the tasks in the non-increasing order of the levels, where $|V| = n$ is the number of tasks. Therefore, the time complexity of phase 1 is $O(|E| + |V|\log|V|)$.

The second phase is performed to obtain all the important parameters like *EST*, *ECT*, *FP*, *LACT*, and *LAST*. Phase 2 calculates these parameters by applying the depth first search with the complexity of $O(|V| + |E|)$.

Recall that in phase 3 the tasks are allocated to the computational nodes. First, all the tasks are checked and allocated to one or more nodes in the while loop based on duplication strategies. In the worst case, all the tasks in the critical path must be duplicated, meaning that the time complexity is $O(h|V|)$time, where h is the height of the DAG. Since h is less than or equal to $|V|$, the complexity of the third phase is $O(|V|^2)$.

Consequently, the overall time complexities of EAD and PEBD are O(2|E| + |V|(lg|V|+1) + |V^2| = O(|E|+|V|^2). For a dense DAG, the number of messages are proportional to O(|V|^2). Hence, the time complexities of EAD and PEBD are O(|V|^2).

4.4 Simulation Results

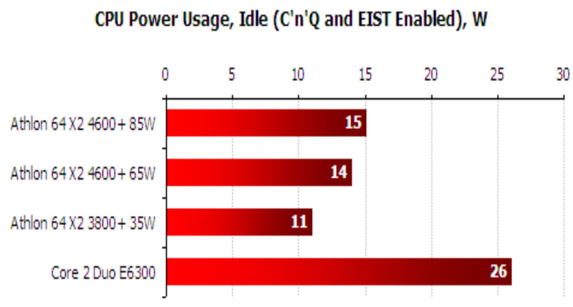
Now we are in the position to evaluate the effectiveness of the proposed energy-aware duplication scheduling algorithms. In this section, we compare EAD and PEBD with two existing scheduling algorithms: the non-duplication-based scheduling heuristic (NDS or MCP) [52], and the task duplication-based scheduling algorithm (TDS) [49]. In order to fairly compare our scheduling algorithms with existing algorithms, we set the same evaluation metrics and parameter tune rule for all simulation results of different algorithms. Additionally, we choose popular processors of AMD and Intel companies and popular interconnections like Myrinet and Infiniband network as our simulation platform, which can make our simulation results more practical and acceptable to industry people.

4.4.1 Simulation Metrics and Parameters

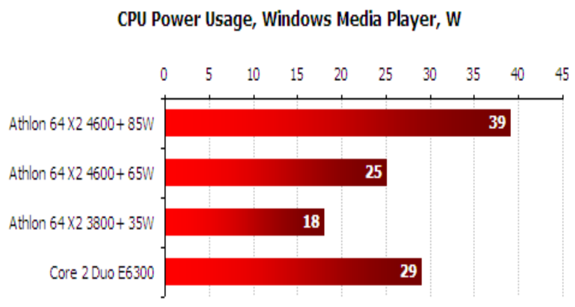
Schedule length and energy consumption are the major two metrics used in our simulation to evaluate the performance of different algorithms. The basic but important rule we followed in our simulations is OTOP (Once Tuning One Parameter). In other words, parameters in the same simulation group results are exactly the same except one parameter is different. By tuning only one parameter, we can clearly observe its impact to clusters and easily find out the system sensitivity to this specific parameter. The important parameters tuned in our simulations include Communication-to-Computation Ratio (CCR), energy threshold, interconnection type and processor type. It is to be noted that CCR is an overall average time parameter to measure the communication time and computation time, which is defined in equation (1). Generally speaking, data transfer intensive applications have higher CCR, whereas the CCR of computation-intensive applications is lower.

The processors used in our simulator are AMD Athlon 64 X2 4600+ with 85W TDP, AMD Athlon 64 X2 4600+ with 65W TDP, AMD Athlon 64 X2 3800+ with 35W TDP, Intel Core 2 Duo E6300 processor. Figure 4.5 demonstrates the energy consumption rate of each processor in idle, light, busy and heavy working mode. The data source is from the latest test report of Xbit Lab (**http://www.xbitlabs.com**).

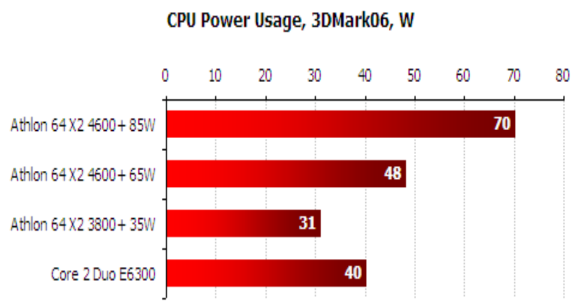
Myrinet and Infiniband network are the interconnections used in our simulations. The energy consumption parameters used for Myrinet and Infiniband are 33.6w and 65w respectively, which are based on the products technical report from Myricom and Qlogic company.



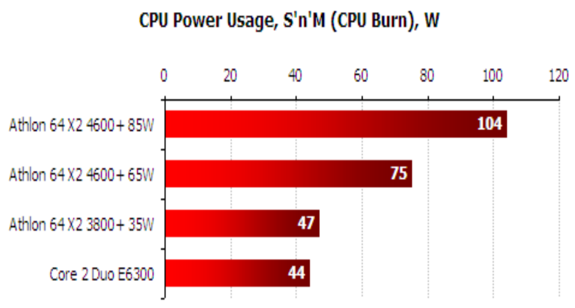
(a) Energy parameter in idle mode



(b) Energy parameter in light mode



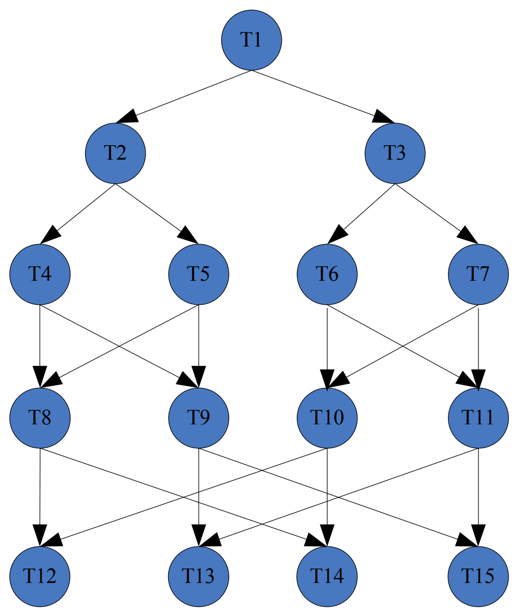
(c) Energy parameter in busy mode



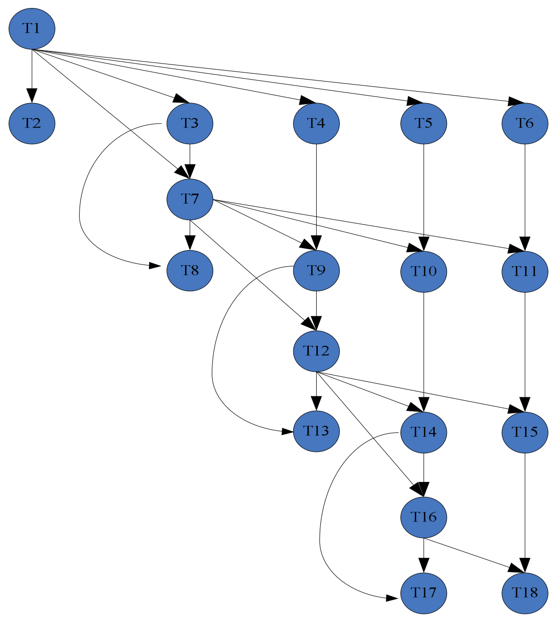
(d) Energy parameter in heavy mode

Figure 4.5 Energy consumption parameters in different working modes

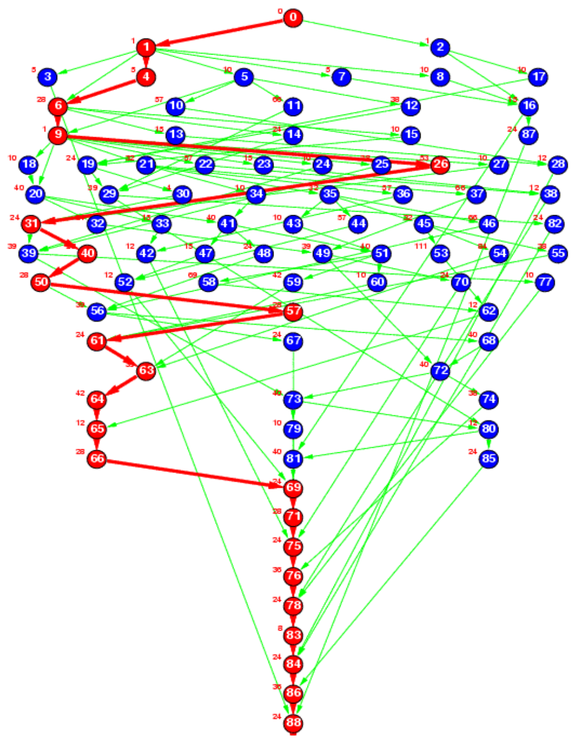
We simulated four DAGs, which include Fast Fourier Transform Tree (15 tasks), Gaussian Elimination Tree(18 tasks), Robot Control application (88 tasks) and Sparse Matrix Solver application (96 tasks). The detailed tree structures are shown in Figure 4.6 and the tree structure files of two actual applications (Robot and Sparse) can be downloaded at Standard Task Graph website [53]. Robot Control DAGs represents a task graph for Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator [54]. Sparse Matrix Solver DAGs represents a task graph for a random sparse matrix solver of an electronic circuit simulation that was generated using a symbolic generation technique and the OSCAR FORTRAN compiler [55] [56].



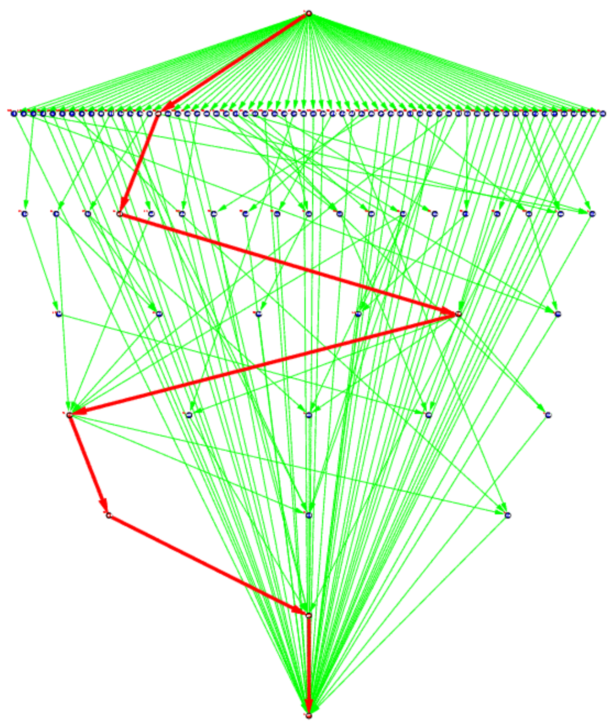
(a) Fast Fourier Transform



(b) Gaussian Elimination



(c) Robot Control



(d) Sparse Matrix Solver

Figure 4.6 Structure of simulated trees and applications

4.4.2 Impact of Processor Types to Energy

Processors play an important role in the computing capacity and energy consumption of clusters. In order to study impacts of processors on the performance of EAD and PEBD, we choose three different AMD processors and one Intel processor as CPUs used in our simulated clusters. All the power consumption parameters of these four types of processors are listed in Figure 4.5. Table 4.3 shows the simulation environment and according parameters of the clusters which we collect data for Figures 4.7.

Table 4.3 Simulation environment of processor impact

Simulation environment	
Processor type	**Athlon 4600+ 85W, Athlon 4600+ 65W Athlon 3800+ 35W, Intel Core2 Duo E6300**
Processor working mode	**Heavy**
Interconnection	**Myrinet**
Simulated Trees or Applications	**Gaussian Elimination, Fast Fourier Transform**
CCR	**(0.4, 4)**

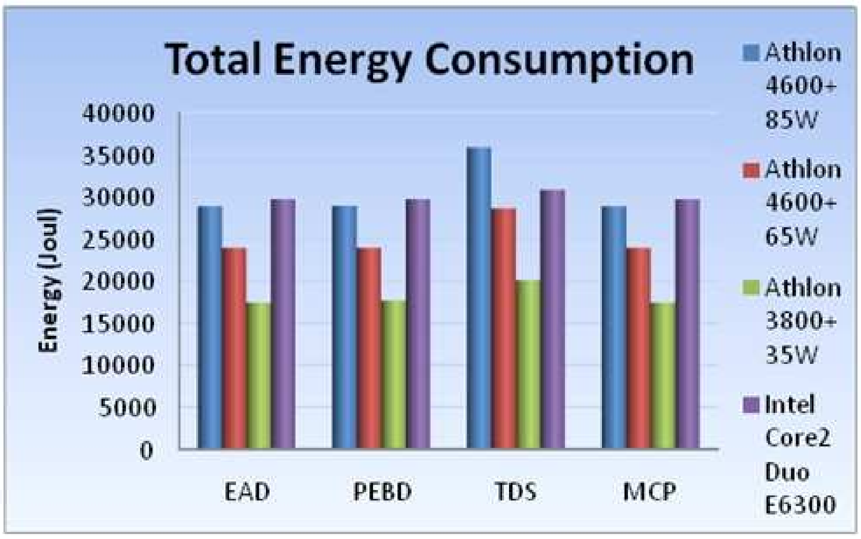


Figure 4.7(a) Energy consumption for different processors (Gaussian, CCR=0.4)

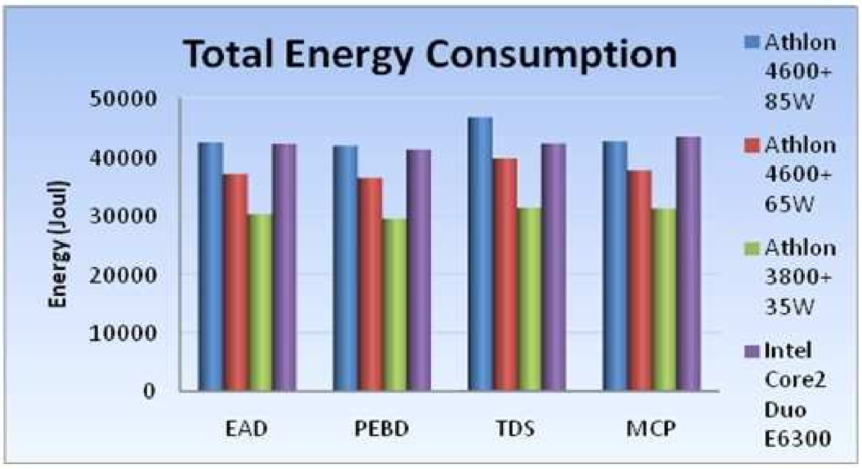


Figure 4.7(b) Energy consumption for different processors (Gaussian, CCR=4)

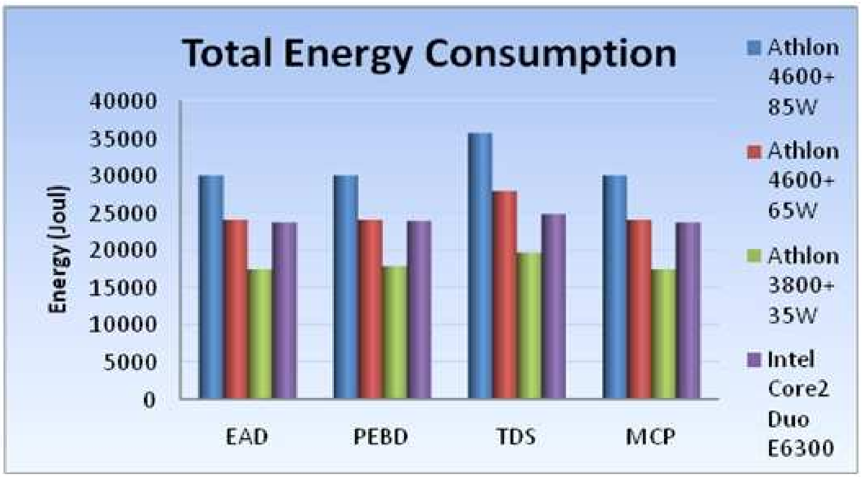


Figure 4.7(c) Energy consumption for different processors (FFT, CCR=0.4)

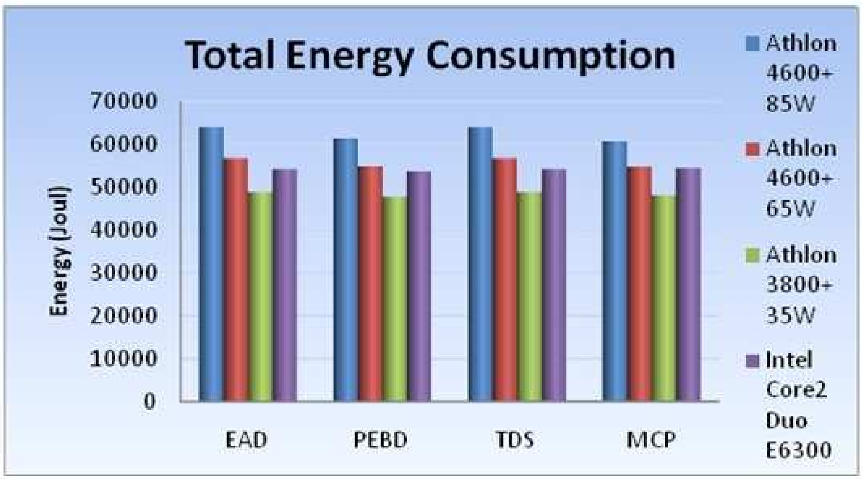


Figure 4.7(d) Energy consumption for different processors (FFT, CCR=4)

We observe from Figures 4.7 that EAD and PEBD can provide significant performance improvements for these four kiMCP of processors. In general, EAD and PEBD perform much better on Athlon 4600+ 85W than Intel Core2 Duo E6300. An intriguing result for EAD or PEBD is that a larger discrepancy between CPU_heavy and CPU_idle leads to a more pronounced performance enhancements. For instance, the gap between CPU_heavy and CPU_idle (i.e., 104W – 15W = 89) in Athlon 4600+ 85W, which is bigger than that (i.e., 44W – 26W = 18W) of Intel Core2 Duo E6300; EAD and PEBD outperform TDS by 19.47% and 19.36% in Athlon 4600+ 85W whereas the percentage drops down to 3.73% and 3.76% respectively in Intel Core2 Duo E6300. We did exactly the same experiments in FFT tree (results shown in Figures 4.7(c) and (d)) and found very similar trend. The implication of the result is that processors with large descrepency between CPU_heavy and CPU_idle can benefit greatly from EAD and PEBD, regardless of the value of CCR. This implication provides a useful suggestion to users what kind of processor is more suitable for our algorithms.

4.4.3 Impact of Interconnection Types to Energy

Network energy consumption is a second critical factor affecting total energy dissipation in clusters. In this subsection, our goal is to study the impacts of different interconnections on the performance of the EAD and PEBD algorithms. The underneath interconnections used in this group of simulation results are Myrinet and Infiniband, which are two of the popular networks implemented in modern clusters. Table 4.4 shows the simulation environment and according parameters of the clusters which we collect data for Figures 4.8.

Table 4.4. Simulation environment of interconnection impact

Simulation environment	
Processor type	**Intel Core2 Duo E6300**
Processor working mode	**Heavy**
Interconnection	**Myrinet , Infiniband**
Simulated Trees or Applications	**Robot Control , Sparse Matrix Solver**
CCR	**(0.1, 0.5, 1, 5, 10)**

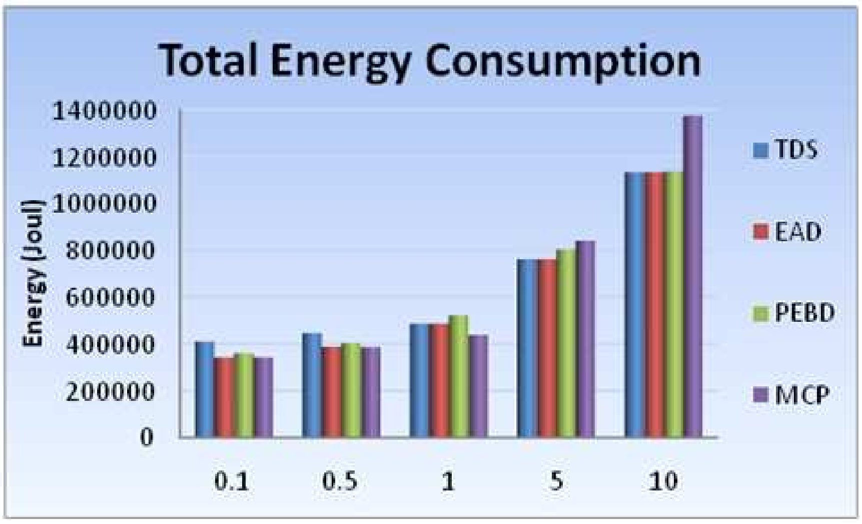


Figure 4.8(a) Total energy consumption (Robot Control, Myrinet)

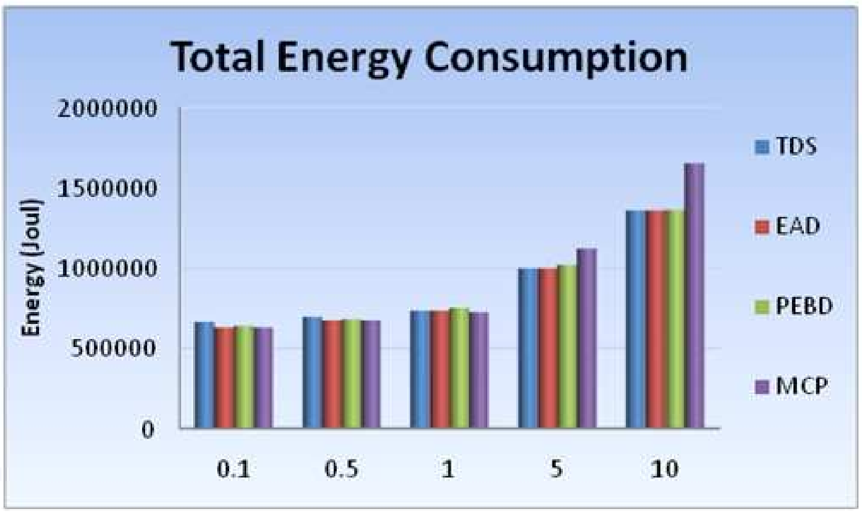


Figure 4.8(b) Total energy consumption (Robot Control, Infiniband)

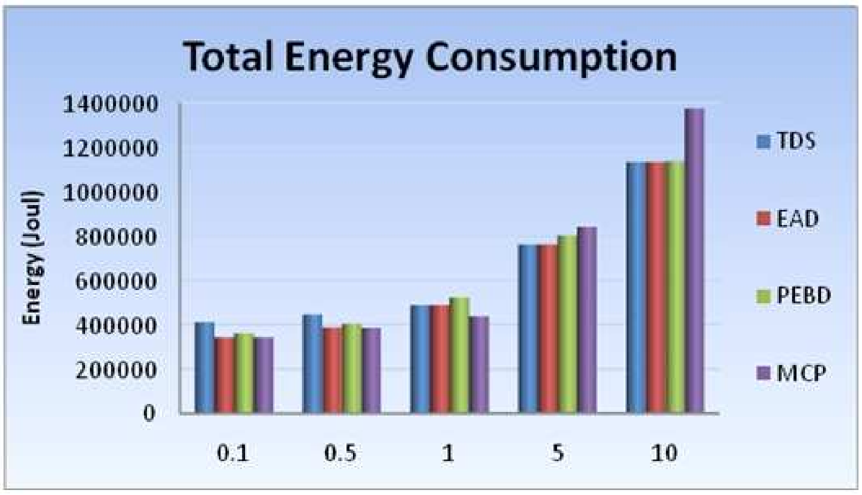


Figure 4.8(c) Total energy consumption (Sparse Matrix Solver, Myrinet)

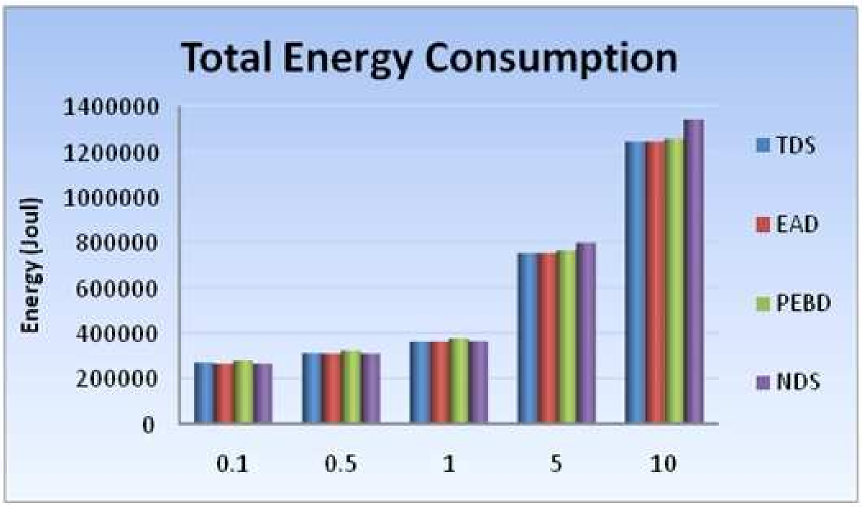


Figure 4.8(d) Total energy consumption (Sparse Matrix Solver, Infiniband)

From Figures 4.8, we can find out that the overall performance of EAD and PEBD are better than TDS and MCP. Another interesting observation is that both EAD and PEBD work better, i.e. save more energy, when the interconnection is Myrinet. For example, for the same Robot Control application, EAD outperformance TDS in terms of energy conservation for 16.65% (CCR=0.1) and 13.25% (CCR=0.5) if we use Myrinet, whereas the numbers will change to 5% (CCR=0.1) and 3.14% (CCR=0.5) when we choose Infiniband. Similary, for the same Sparse Matrix Solver application, PEBD outperformance MCP in terms of power consumption for 4.64% (CCR=5) and 17.25% (CCR=10) if we use Myrinet, whereas the numbers will change to 4.17% (CCR=5) and 6.35% (CCR=10) when we choose Infiniband. Since the interconnection power consumption rate used in our siumlations for Myrinet and Infiniband are 33.6w and 65w respectively, we can see that the efficiency of our algorithms are somehow degraded by the high interconnection power consumption. In other words, less portion of network energy consumption is a positive factor to make our algorithms have better performance.

4.4.4 Impact of Application Types to Energy

Will the application type affect the efficiency of EAD and PEBD? If it does, what is the most important factor? In order to answer these questions, we simulated Robot Control and Sparse Matrix Solver applications under exactly the same environments, which means we have same processor, same interconnections, same CCRs and even same energy threshold. Figures 4.9 shows the simulation results which illustrate the different efficiency of both EAD and PEBD for different applications. Table 4.5 shows the simulation environment of Figures 4.9.

Table 4.5 Simulation environment of application impact

Simulation environment	
Processor type	**Intel Core2 Duo E6300** **Athlon 3800+ 35W**
Processor working mode	**Heavy**
Interconnection	**Myrinet**
Simulated Trees or Applications	**Robot Control , Sparse Matrix Solver**
CCR	**(0.1, 0.5, 1, 5, 10)**

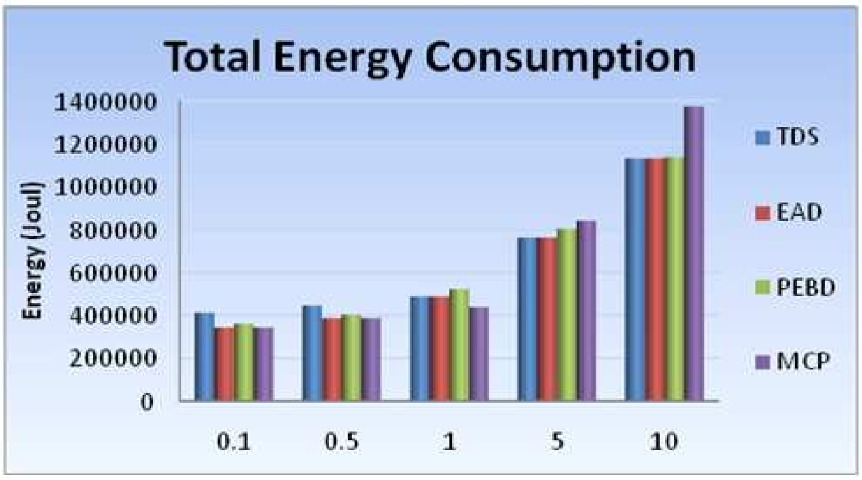


Figure 4.9(a) Energy of Intel Core2 Duo E6300 (Robert Control, Myrinet)

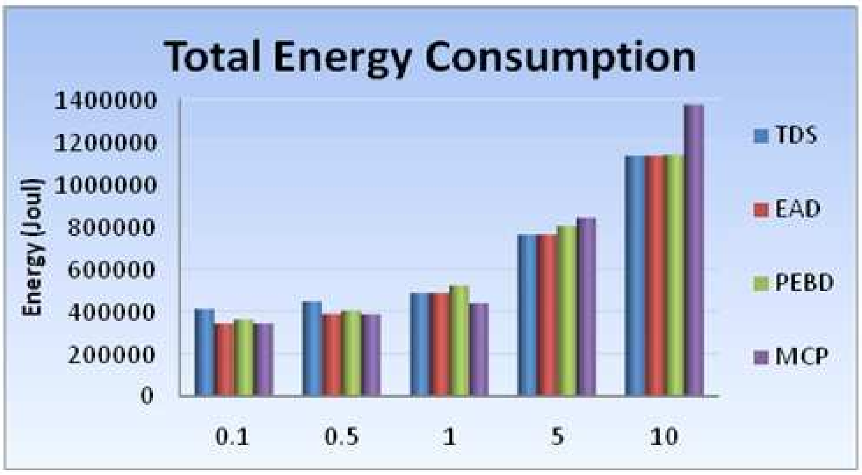


Figure 4.9(b) Energy of Intel Core2 Duo E6300 (Sparse Matrix Solver, Myrinet)

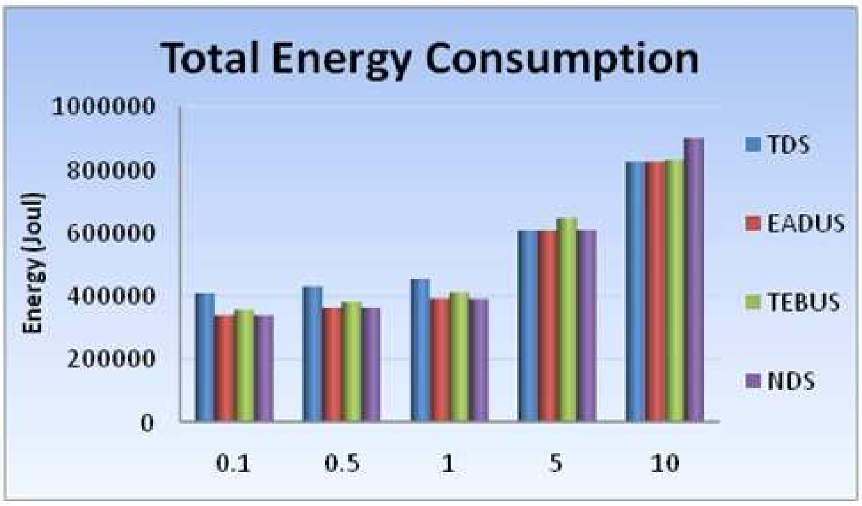


Figure 4.9(c) Energy of Athlon 3800+ 35W (Robert Control, Myrinet)

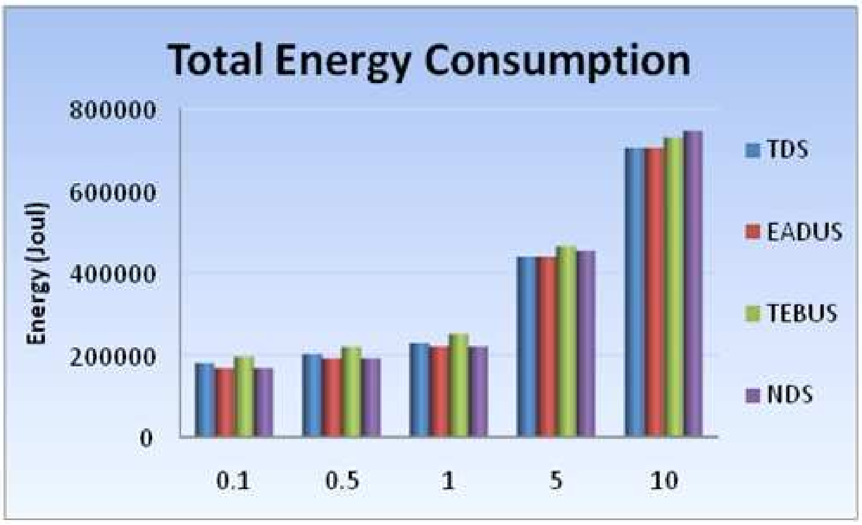


Figure 4.9(d) Energy of Athlon 3800+ 35W (Sparse Matrix Solver, Myrinet)

From Figures 4.9, we can see that EAD and PEBD can save more energy in the Robot Control applications. For example, in the Robot Control application, EAD can save more energy than TDS up to 17.07% (CCR=0.1, Athlon 3800+ 35W) and 15.78% (CCR=0.5, Athlon 3800+ 35W), whereas the numbers will drop down to 6.89% (CCR=0.1, Athlon 3800+ 35W) and 5.37% (CCR=0.5, Athlon 3800+ 35W) for Sparse Matrix Solver application. Since all the other parameters are exactly the same except the application structures (see Figures 4.9(c) and (d)), we can draw the conclusion that application types do affect the efficiency of our algorithms. Based on the data provided by Standard Task Graph Set website [53], the parallelism of Robert Control and Sparse Matrix Solver applications are 4.363796 and 15.868853 respectively, which means Robert Control has more task dependencies thus there exists more possibility for EAD and PEBD to consume energy by judiciously duplicating tasks. In other words, the task dependencies and parallelism level are the key points to decide the efficiency of our algorithms.

4.4.5 Impact of CCR to Energy

Group Figures 4.10 illustrate the CCR impact to processor energy, interconnection energy and total energy. Four observations are evident from this group of experimental results. First, the overall performance of EAD and PEBD outperforms MCP and TDS. Second, both EAD and PEBD are very sensitive to CCR. For example, when CCR is 0.1, EAD and PEBD perform 11.33% and 8.33% better than TDS. However, the performance drops down to 9.39% and 6.85% if we tune the CCR to 0.5. Third, MCP provides the greatest energy savings if CCR is less than 1. This is because energy cost due to interconnection is extremely low with a small CCR value. Finally yet importantly, the communication energy cost will dramatically increase when CCR going higher and become the major power consumer of whole system.

Table 4.6 Simulation environment of CCR impact

Processor type:	**Athlon 3800+ 35W**
Processor working mode:	Busy
Interconnection:	Myrinet
Simulated Application:	Robot Control
CCR:	(0.1, 0.5, 1, 5, 10)

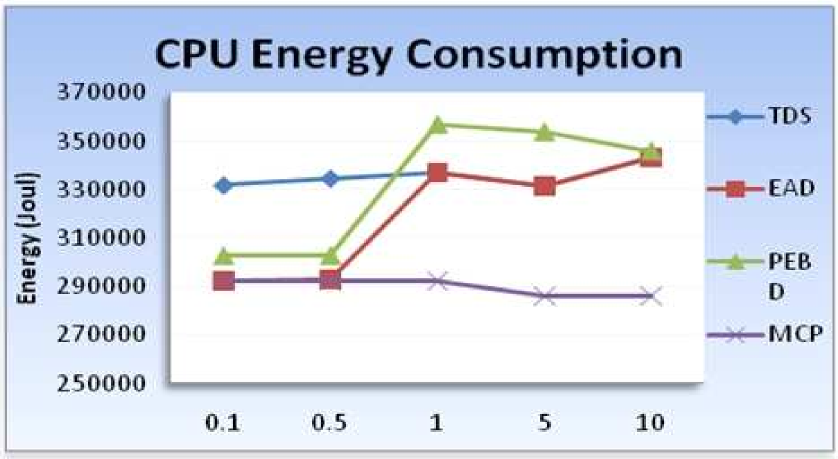


Figure 4.10(a) CPU energy consumption under different CCRs

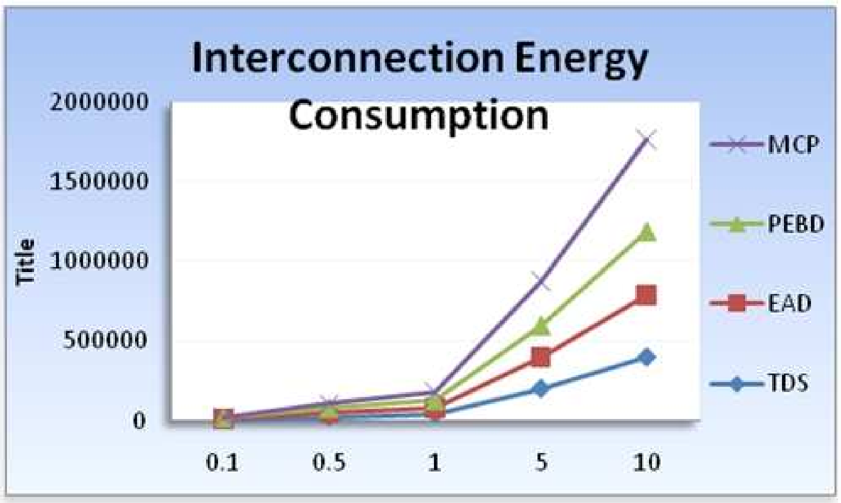


Figure 4.10(b) Interconnection energy under different CCRs

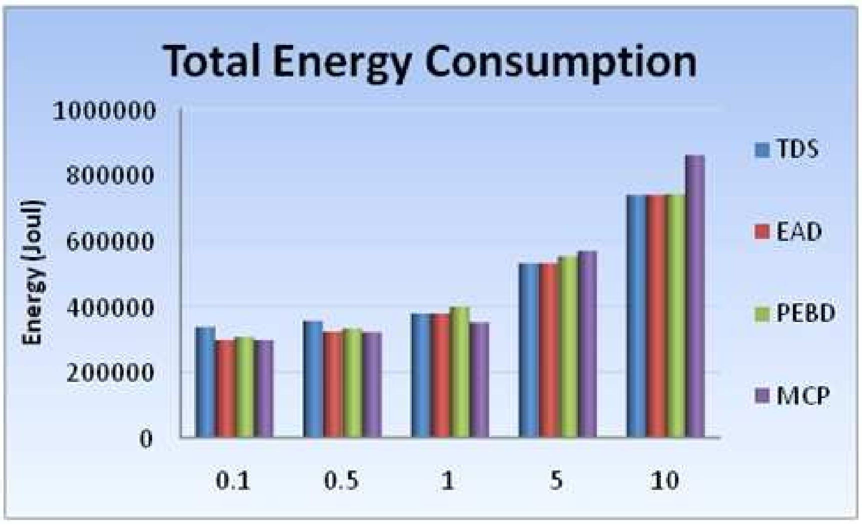


Figure 4.10(c) Total energy consumption under different CCRs

4.4.6 Impact of Processor Status to energy

Processors may have different working modes like idle, not busy, busy and extremely busy. The energy consumption rate is different under different modes. In order to speculate the impact of processor status to energy consumption, we simulated three working modes, for AMD Athlon 3800+ 35W processor. When processor is running applications like widows media player, 3D graph generation, CD burn, it is in light, busy and heavy modes respectively. The simulation results are shown in Figures 4.11 and the corresponding energy consumption parameters for each working mode could be found in Figures 4.5.

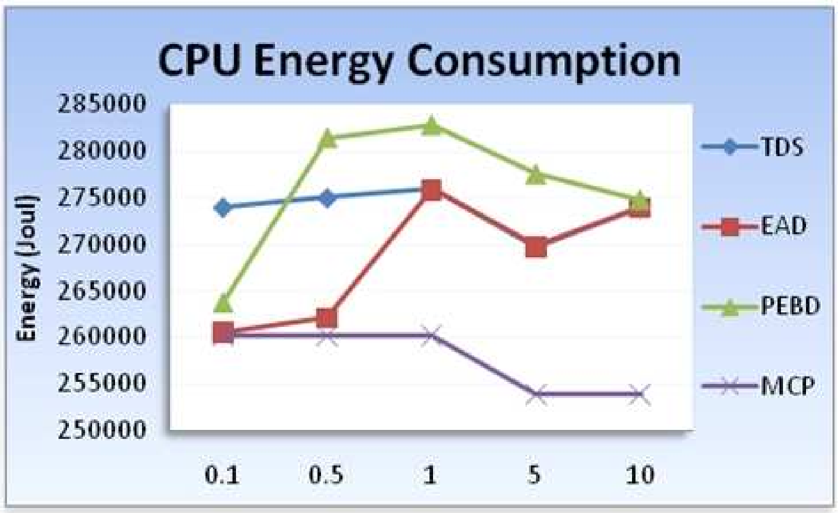


Figure 4.11(a) CPU energy consumption under light mode

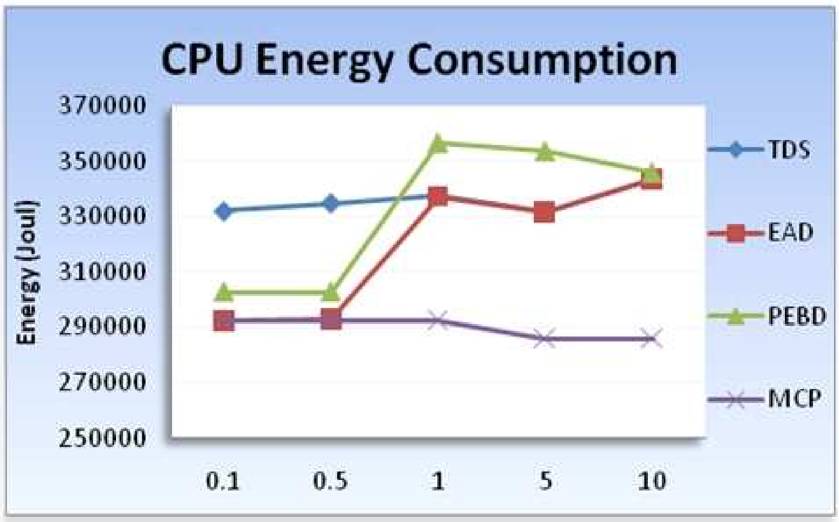


Figure 4.11(b) CPU energy consumption under busy mode

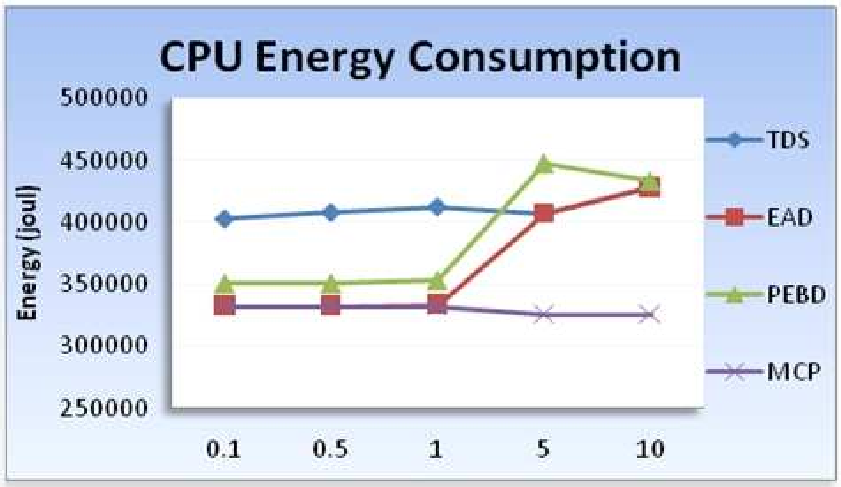


Figure 4.11(c) CPU energy consumption under heavy mode

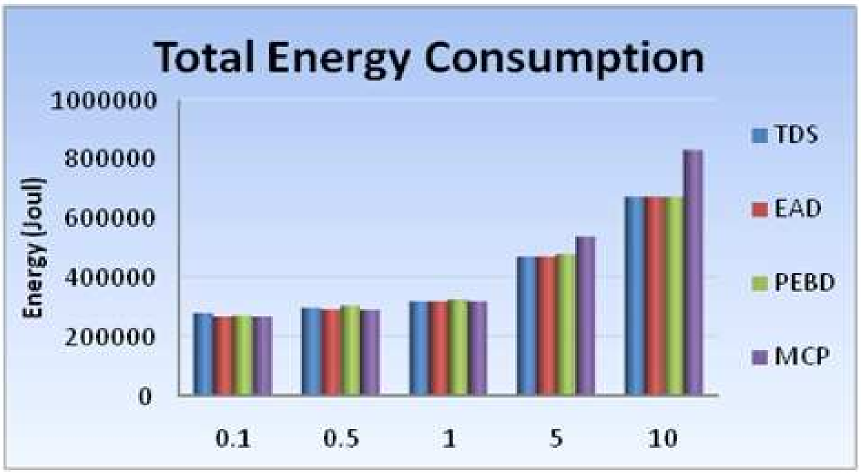


Figure 4.11(d) Total energy consumption under light mode

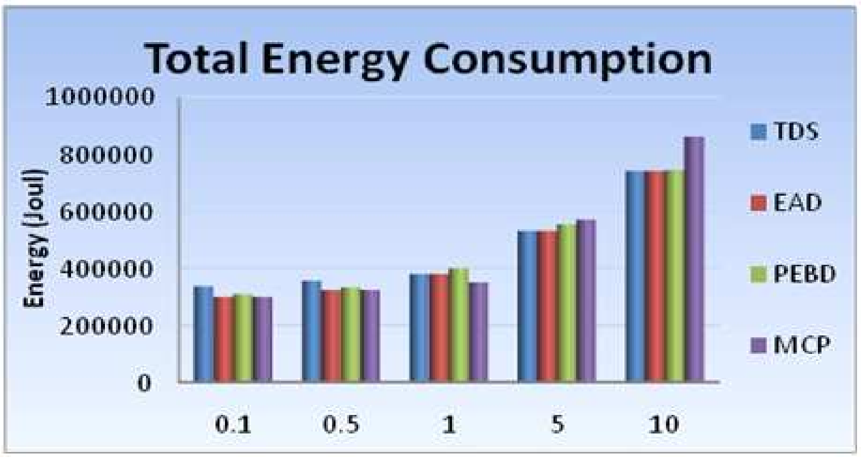


Figure 4.11(e) Total energy consumption under busy mode

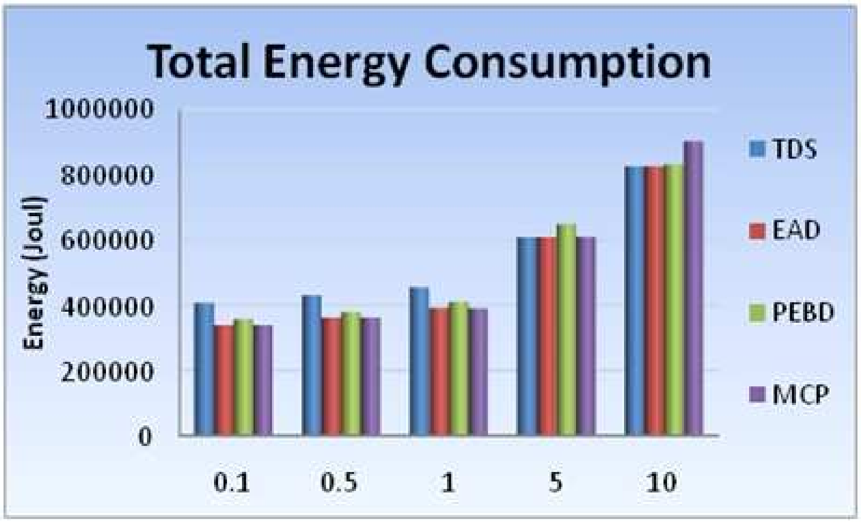


Figure 4.11(f) Total energy consumption under heavy mode

If we look at Figures 4.11(a) (b) (c) together, you will find the CPU energy consumption of EAD and PEBD are various under different modes, which indicates EAD and PEBD have different duplication decisions. If we compared the results shown in Fig.4.11 (d) (e) (f), we can easily find that EAD and PEBD work more efficiently under heavy mode. For example, EAD and PEBD can conserve 17.07%, 12.6% more energy than TDS in heavy mode, whereas these numbers will become 11.33%, 8.33% in busy mode and 4.43%, 3.23% in light mode. Recall that in the heavy mode, the processor has the biggest energy consumption gap between CPU idle and CPU working, we can easily find out the same conclusion as section 6.2, which tells us processors with large energy consumption descrepency betweent CPU_working and CPU_idle can benefit greatly from EAD and PEBD, regardless of the value of CCR.

4.4.7 Impact to Schedule Length

Group Figures 4.12 depict the experimental results used to evaluate the overall performance of the four scheduling algorithms in term of schedule length. Figures

4.12(a) and (b) show the scheduling lengths of schedules made by the four algorithms for the Gaussian Elimination and Fast Fourier Transform applications. The results show that EAD and PEBD efficiently reduce energy consumption without adversely affecting performance of the applications. For example, on average the schedule lengths of Gaussian Elimination produced by EAD and PEBD are merely 5.7% and 2.2% larger than those generated by TDS. Similarly, on average the schedule lengths of Fast Fourier Transform yielded by EAD and PEBD are only 2.92% and 2.02% longer than that of TDS. These results suggest that it is worth trading a marginal degradation in schedule length for a significant reduction in energy dissipation for cluster computing systems.

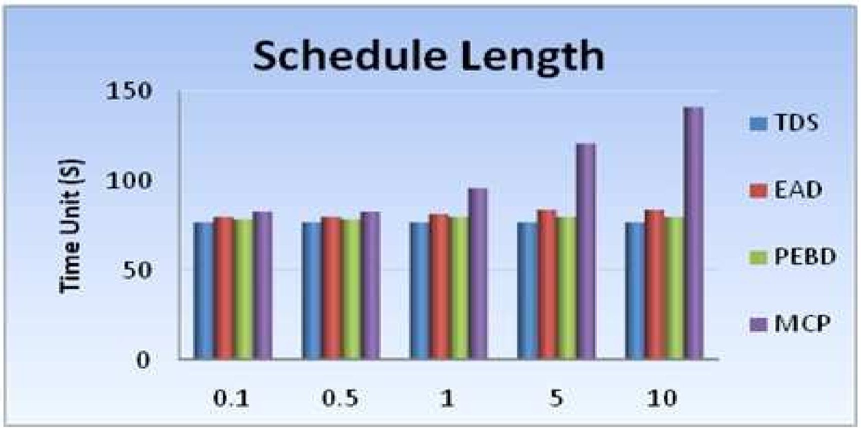


Figure 4.12(a) Schedule length of Gaussian Elimination

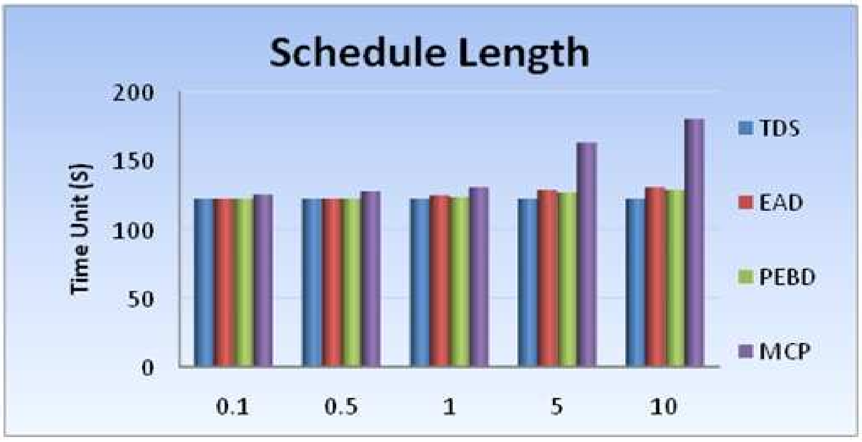


Figure 4.12(b) Schedule length of Sparse Matrix Solver

4.5 Summary

In this chapter, we addressed the issue of allocating tasks of parallel applications running on clusters with an objective of shortening schedule lengths while conserving energy. Specifically, we proposed two improved duplication-based scheduling algorithms, namely the Energy-Aware Duplication algorithm (or EAD) and the Performance-Energy Balanced Duplication algorithm (or PEBD). EAD and PEBD are designed and implemented to provide energy savings in clusters by duplicating tasks on more than one computational node. While EAD is able to aggressively provide the greatest energy savings by making use of task replicas to eliminate energy-consuming messages, PEBD aims at making tradeoffs between energy conservation and performance.

To facilitate the presentation of EAD and PEBD, we built mathematical models to describe a cluster system framework, parallel applications with precedence constraints, and energy consumption model. We conducted extensive experiments and our experimental results show that EAD and PEBD are more energy-efficient compared with other two existing allocation schemes called MCP(or NDS) and TDS. Our conclusion is that EAD and PEBD are capable of trading a marginal degradation in schedule length for a significant reduction in energy consumption for homogeneous cluster computing systems.

Chapter 5

Energy-Efficient Scheduling For Grids

In the previous chapter, we have designed two energy-efficient scheduling algorithms for homogeneous clusters, which comprise a set of identical characteristics in terms of CPU speed, memory capacity, power consumption rate and interconnections. However, these algorithms cannot be directly used for heterogeneous high performance computing platforms like grids. In this chapter, we propose two energy-aware scheduling algorithms, called Energy-Efficient Task Duplication Scheduling (EETDS) and Heterogeneous Energy-Aware Duplication Scheduling (HEADUS), which attempt to make the best tradeoffs between performance and energy savings for parallel applications running on heterogeneous grids.

This chapter is organized as follows. Section 5.1 presents the motivation of this study. In section 5.2, we define the mathematical models used in our grid systems, which include a grid model, parallel tasks model, and an energy consumption model. Next, in section 5.3, we discuss the job scheduling in grid systems. In section 5.4, we present the proposed EETDS and HEADUS scheduling algorithms in detail and illustrate how they work using a concrete example. Section 5.5 proves that time complexity of EETDS and HEADUS. Experimental results with qualitative comparisons to other two existing approaches are analyzed in section 5.6. Finally, section 5.7 summarize the entire chapter.

5.1 Motivation

Although it is common that a new and stand-alone cluster system is homogeneous in nature, upgraded clusters or networked clusters are likely to be heterogeneous in practice. In other words, heterogeneity of a variety of resources such as CPU, memory, and interconnection, may exist in cluster systems. This is because, to improve performance and support more users, new nodes that might have different characteristics than the original ones may be added to the systems or several smaller clusters of different characteristics may be connected via a high-speed network to form a bigger one. Accordingly, heterogeneity may exist in a variety of resources such as CPU, memory, and interconnection etc.

Computing grids are one of the typical distributed systems with heterogeneity. A computational grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across multiple administrative domains based on the resources’ availability, capacity, performance, cost and users' quality-of-service requirements. Literally speaking, a large-scale distributed system that qualifies the following three conditions could be envisioned as a computational grid [57]. (1) Computing resources are not administered centrally; (2) open standards are used; and (3) non-trivial quality of service is achieved. Grid applications distinguish themselves from traditional distributed applications because they not only simultaneously use large number of resources, but also have stringent performance requirements, dynamic resource requirements, and complex communication structures [58]. As our economy shifts from paper-based to digital information management, large-scale grid computing platforms have been widely deployed to support the complicated scientific and commercial applications which require intensive data processing and data storage in nature. As you can imagine, the powerful computing capability of grids is actually in the cost of huge energy consumption. Therefore, designing energy-efficient algorithms for grids becomes highly desirable.

The research shown in this chapter is motivated by the above reasons. However, we realized that the design is much more challenging compared with the design for homogeneous clusters. In the study shown in this chapter, we take into account multiple design objectives, including performance (measured by throughput and schedule length), energy efficiency, and heterogeneities.

5.2 System Model

In this section, we describe mathematical models used to represent heterogeneous grids, parallel applications with precedence constraints. Since the energy consumption model is the same to the model used in cluster systems, we do not explain it again in this chapter. Please refer to section 4.1.3 for details.

5.2.1 Grid Systems Model

A grid system consists of a set $P = \{p_1, p_2,..., p_m\}$ of heterogeneous computing nodes (hereinafter referred to as nodes) connected by a high-speed interconnect like fast Ethernet, gigabit Ethernet, SCI, FDDI or Myrinet. A heterogeneous grid can be represented by a graph, where computing nodes are vertices. There exists a weighted edge if a pair of corresponding nodes can communicate with each other. An $n \times m$ binary allocation matrix X is used to reflect a mapping of n tasks to m heterogeneous nodes. Thus, element x_{ij} in X is "1" if task t_i is assigned to node p_j and is "0", otherwise. Since our scheduling algorithms will be verified in a heterogeneous environment, it is imperative to define the following constraints for our heterogeneous grid system model. First, different nodes have different preference with respect to tasks, meaning that a node offering task t_i a shorter execution time does not necessarily run faster for another task t_j. Thus, different nodes in a heterogeneous cluster favor different kinds of tasks. Second, execution times of tasks on different nodes may various because the nodes may have various clock speed and processing capabilities. Third, the transmission rates of network interconnections depend on underlying network types. Last, energy consumption rates of the nodes and interconnections may not necessarily be identical.

To simply the system model without loss of generality, we assume that all nodes are fully connected with dedicated and reliable network interconnections. Each node communicates with other nodes through message passing; communication time between two tasks assigned to the same node is negligible. In addition, we assume computation and communication can take place simultaneously in our system model. This assumption is reasonable because each computing node in a modern cluster has a communication coprocessor that can be used to free the processor in the node from communication tasks. Since we primarily focus on energy consumption, each node in the system has an energy consumption rate measured by Joule per unit time. Furthermore, each network link is characterized by its energy consumption rate that heavily relies on the link's transmission rate, which is modeled by weight w_{ij} of the edge between nodes p_i and p_j.

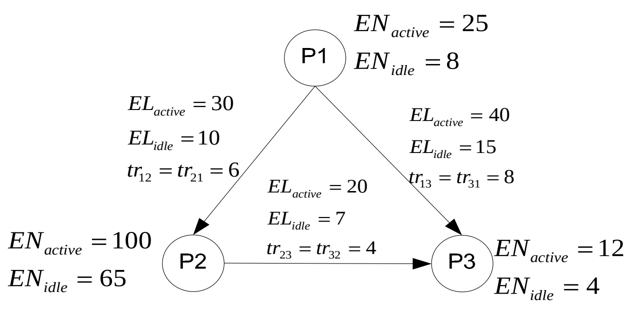
5.2.2 Parallel Tasks Model

Parallel tasks with dependencies are represented by *Directed Acyclic Graphs (DAGs)* in this study. Throughout this paper, a collaborative application is specified as a pair, i.e, (T, E), where $T = \{t_1, t_2, ..., t_n\}$ represents a set of parallel tasks, E is a set of weighted and directed edges representing communication cost among tasks, e.g., $(t_i, t_j) \in E$ is a message transmitted from task t_i to t_j. Precedence constrains of the parallel tasks are represented by all edges in set E. Communication time spent in delivering a message $(t_i, t_j) \in E$ from task t_i on node p_u to t_j on p_v is determined by s_{ij}/b_{uv}, where s_{ij} is the message's data size and b_{uv} is the transmission rate of a link connecting node p_u and p_v. The execution times of task t_i running on a set of heterogeneous computing nodes are modeled by a vector, i.e., $c_i = \left(c_i^1, c_i^2, \cdots, c_i^m\right)$, where c_i^j represents the execution time of t_i on the jth computing node. If task t_i cannot be executed on node $p_{j,}$ the corresponding execution time c_i^j in the vector c_i is marked as ∞. We define a task as an entry task if it does not have any predecessor tasks and; similarly, a task is called an exit task if there is no task following behind it.

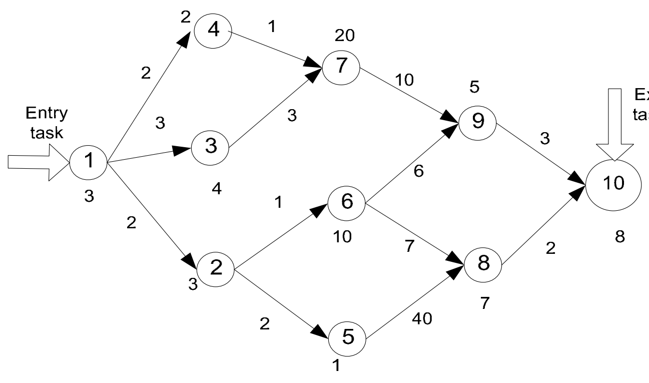
An Example. Figure 5.1 illustrates the task description of a parallel application represented by a task graph, a mapping matrix, and a cluster with three heterogeneous computing nodes. The task graph contains ten tasks; the computing node graph (or processor graph) has three heterogeneous computing nodes. EN^{i}_{active} is the energy consumption rate of the *i*th computing node in the busy mode, and EN^{i}_{idle} is the energy consumption of the *i*th computing node in the idle mode. Similarly, EL^{ij}_{active} and EL^{ij}_{idle} is the energy consumption rate of the link between the *i*th and *j*th nodes when data is being transferred and when no data are being transmitted. For example, the energy consumption rates of the network link between nodes p_1 and p_2 are $EL^{12}_{active} = 30$ and EL^{12}_{idle} = 10 when the link is in the busy and idle modes, respectively. The energy consumption rates of node p_1 is EN^{1}_{active} = 25 and $EN^{1}_{idle} = 8$ when it is active and idle, respectively. We assume that the transmission rate between two computing nodes is same in both directions. The execution time vector of each task on the three processors is illustrated in Figure 5.1(d). For example, the execution times of task t_1 on nodes p_1, p_2, and p_3 are 6.7, 3.9, and 2.0 time units, respectively. Here we should note that task t_9 couldn't be allocated to p_1 because the corresponding item in the mapping matrix is marked as ∞.

Task Description:
TaskSet {T1, T2, …, T9, T10 }
T1 is the entry task;
T10 is the exit task;
T2, T3 and T4 can not start until T1 finished;
T5 and T6 can not start until T2 finished;
T7 can not start until both T3 and T4 finished;
T8 can not start until both T5 and T6 finished;
T9 can not start until both T6 and T7 finished;
T10 can not start until both T8 and T9 finished;

(a) An example task description



(b) A heterogeneous processor graph



(c) A DAG based on description in (a)

$$\begin{array}{c|ccc} & P1 & P2 & P3 \\ \hline T1 & 6.7 & 3.9 & 2.0 \\ T2 & 9.0 & 11.4 & 1.8 \\ T3 & 9.3 & 5.2 & 4.5 \\ T4 & 8.8 & 4.9 & 2.4 \\ T5 & 7.5 & 7.6 & 3.0 \\ T6 & 5.9 & 1.4 & 5.0 \\ T7 & 7.8 & 6.5 & 7.2 \\ T8 & 9.6 & 4.1 & 3.2 \\ T9 & 8.1 & 2.3 & 1.7 \\ T10 & \infty & 10.2 & 9.9 \\ & 11.2 & 1.3 & 2.4 \end{array}$$

(d) A mapping matrix

Figure 5.1 Example task graph and heterogeneous processor graph

5.3 Job Scheduling in Grids

Computational grids are complex multivariate environments, which are made up of numerous grid entities needed to be managed. The job scheduling module plays a key role in making coherent and coordinated use of ubiquitous and heterogeneous resources in a grid system.

The responsibility of the scheduling module includes resource allocations and task scheduling. Figure 5.2 and Figure 5.3 depict the process of job scheduling in grids from the prospective of system and task level respectively. In system view, the job scheduler of grids contains two parts, a global scheduler and several local schedulers. The global level scheduler (or grid level scheduler) coordinates multiple local schedulers while choosing the most appropriate resources for grid applications. It is worth noting that the global level scheduler in most cases has no direct control over grid resources. Consequently, the global level schedule has to communicate with and precisely trigger local level schedulers to complete tasks of jobs submitted by users. The local level schedulers in turn directly handle resources by accessing to local resources. Moreover, the global level scheduler is responsible for collaborating with other fundamental middleware modules such as information services, communication services, and reliability controllers.

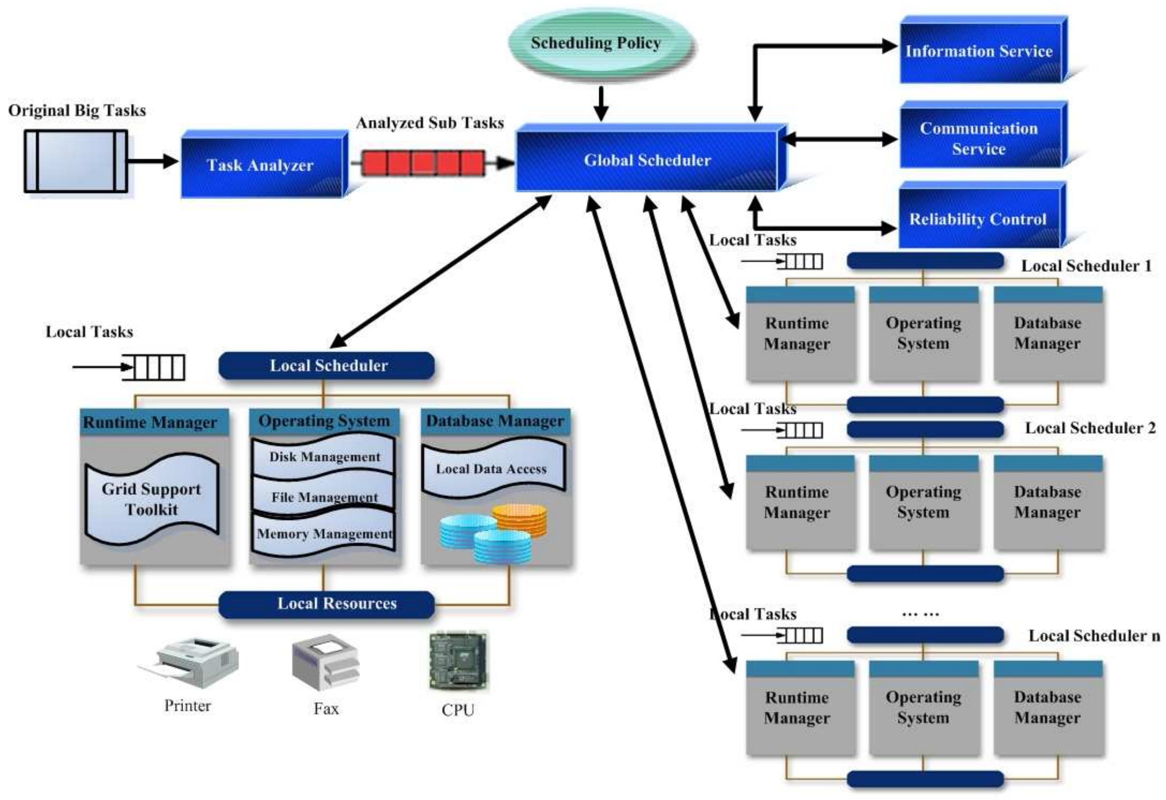


Figure 5.2 The system view of scheduling in a computational grid

The grid level scheduler not only implements energy-efficient scheduling policies but also deals with resource heterogeneities. The grid level scheduler has the following unique attributes.

Reclamation of allocations

Target resources may be reclaimed by the local administrator so that the reclaimed resources can be allocated to tasks with higher priorities. In this case, the scheduler must be able to reclaim allocated resources and reallocate resources to corresponding tasks.

Task and data migrations

This attribute signifies that any task can be interrupted in computing node and the task along with its corresponding data can be migrated to another node. The scheduling module leverages the task and data migrations to improve the performance and reliability of grid systems.

Exclusive allocations

It is common that some computing resources might have particular preferences or exclusiveness for different types of tasks. For example, a computing node offering a shorter execution time for a task does not necessarily run faster for another task. Even worse, some computing nodes may be exclusive to specific types of tasks. Thus, the scheduling module has to resolve conflicts between tasks and resources.

Tentative allocations

To make scheduling decisions with high energy efficiency, it is imperative for the scheduling module to calculate and compare task allocation cost by tentatively allocating tasks to a wide range of available resources. The scheduler is able to efficiently complete revocation of tentative allocations.

Dependent task allocations

A grid application may be consist of multiple dependent tasks, whose dependencies must be handled by the scheduling module. In our framework, a task analyzer provides detailed information of tasks to scheduler; the scheduler makes an effort to first allocate tasks with high dependencies to the same computing resources to significantly reduce communication overheads.

From the task view, once the scheduling module has collected all the information of currently available resources, the module can allocate shared resources to parallel tasks after judiciously choosing target recourses in accordance with specific scheduling policies. Figure 5.3 outlines the job scheduling flow in computational grids. During the course of jobs' execution, a result collector periodically checks randomly returned sub-results and transfers the sub-results to the grid level scheduler. The grid level scheduler further passes on the latest information to all tasks, thereby guaranteeing that the tasks with dependencies can immediately be executed their precedence constraints are met (i.e., sub-results become available).

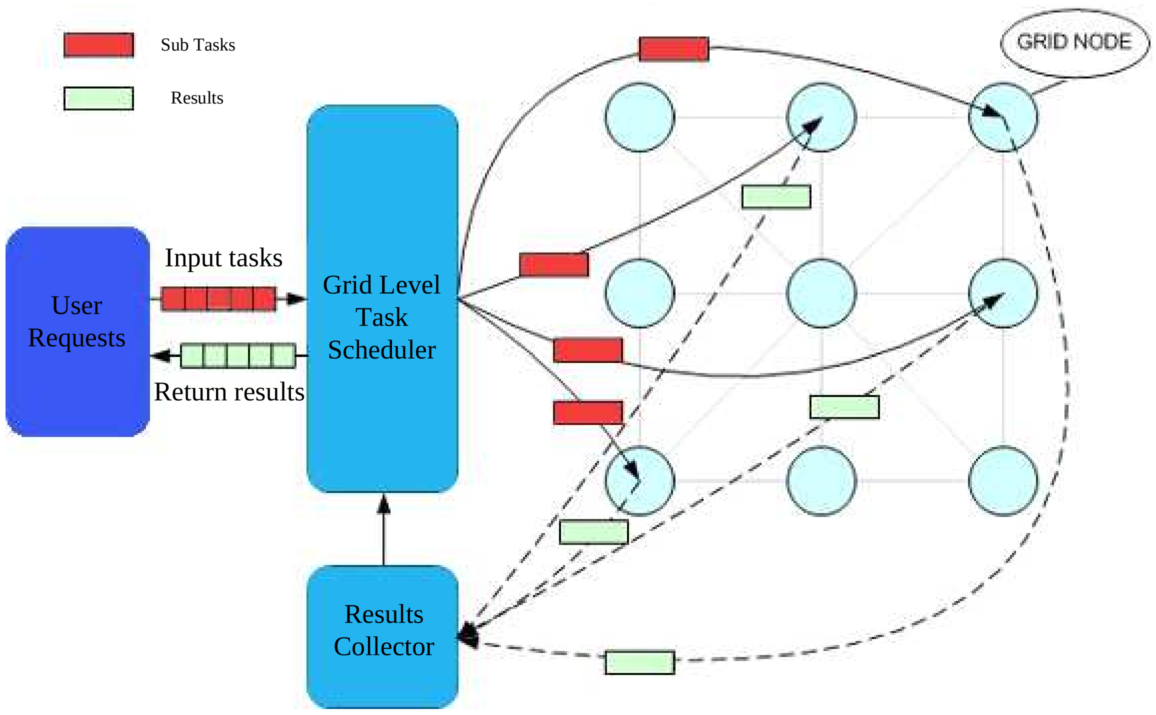


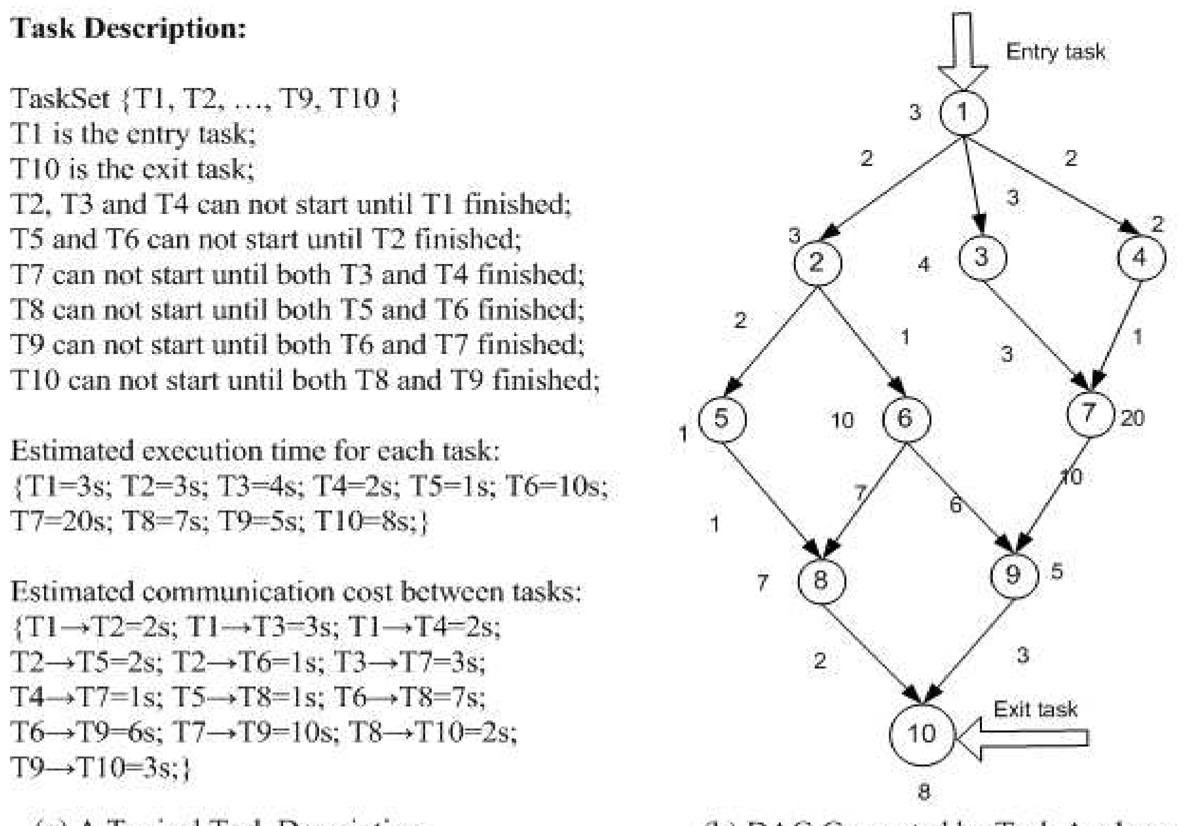
Figure 5.3 The task view of scheduling in a computational grid

5.4 Energy-Efficient Scheduling Algorithms

In this section, we present the details of scheduling algorithms used in our Grid scheduling framework. First, we will explain how the task analyzer can provide information about task dependencies. Next, the proposed EETDS and HEADUS will be explained in three major phases. The first phase is called grouping, in which tasks with highest dependency will be grouped together. Phase two is called task duplication, which aims to duplicate as many tasks as possible if the energy cost will not be significantly increased. In phase three, the scheduler will tentatively allocate the grouped tasks to different available resources and calculate the energy cost. Finally, the scheduler will make its final (energy-performance balanced) decision and completed the real allocations.

5.4.1 The Task Analyzer

The task analyzer is responsible for analyzing tasks characteristic and task dependencies. In addition, the task analyzer has to accurately estimate execution times of tasks based on task types or information provided by users. In our framework, parallel tasks with dependencies are represented by *Directed Acyclic Graphs (DAGs)*. Throughout this paper, a grid application is specified as a pair, i.e, *(T, E)*, where $T = \{t_1, t_2, ..., t_n\}$ represents a set of parallel tasks, E is a set of weighted and directed edges representing communication cost among tasks, e.g., $(t_i, t_j) \in E$ is a message transmitted from task t_i to t_j. Task dependencies among the parallel tasks are represented by all edges in set E. Communication time spent in delivering a message $(t_i, t_j) \in E$ from task t_i on node p_u to t_j on p_v is determined by s_{ij}/b_{uv}, where s_{ij} is the message's data size and b_{uv} is the transmission rate of a link connecting node p_u and p_v. The execution times of task t_i running on a set of heterogeneous computing nodes are modeled by a vector, i.e., $c_i = (c_i^1, c_i^2, \cdots, c_i^m)$, where c_i^j represents the execution time of t_i on the jth computing node. If task t_i cannot be executed on node p_j, the corresponding execution time c_i^j in the vector c_i is marked as ∞. We define a task as an entry task if it does not have any predecessor tasks and; similarly, a task is called an exit task if there is no task following behind it. The task analyzer will take the user request (usually it contains the necessary task description information) as input and generate DAGs as output. Figure 5.4 illustrates a typical task description file and the DAG generated by the task analyzer.



Task Description:

TaskSet {T1, T2, …, T9, T10 }
T1 is the entry task;
T10 is the exit task;
T2, T3 and T4 can not start until T1 finished;
T5 and T6 can not start until T2 finished;
T7 can not start until both T3 and T4 finished;
T8 can not start until both T5 and T6 finished;
T9 can not start until both T6 and T7 finished;
T10 can not start until both T8 and T9 finished;

Estimated execution time for each task:
{T1=3s; T2=3s; T3=4s; T4=2s; T5=1s; T6=10s;
T7=20s; T8=7s; T9=5s; T10=8s;}

Estimated communication cost between tasks:
{T1→T2=2s; T1→T3=3s; T1→T4=2s;
T2→T5=2s; T2→T6=1s; T3→T7=3s;
T4→T7=1s; T5→T8=1s; T6→T8=7s;
T6→T9=6s; T7→T9=10s; T8→T10=2s;
T9→T10=3s;}

(a) A Typical Task Description (b) DAG Generated by Task Analyzer

Figure 5.4 A directed acyclic graph (DAG) analyzed by the task analyzer

5.4.2 Grouping Phase

The grouping phase of our algorithms is to associate the most relevant tasks (i.e. tasks in the same critical paths) into groups. Given a parallel application modeled as a task graph or DAG, the grouping phase yields a group-based graph of the DAG. Since all tasks in a group are allocated to the same computing node where there are no waiting times among the tasks within the group, we can reduce communication overheads of highly dependent tasks with intensive communications. Additionally, a task-duplication strategy is applied in the process of grouping to further improve system performance by replicating tasks into multiple computing nodes if schedule lengths can be shortened.

More specifically, the grouping phase can be finely divided into two sub-steps, namely, original task sequence generating and parameters calculating. Since these two steps are quite similar with the first two steps used for EAD and PEBD. Please refer to section 4.2.1 and section 4.2.2 for details.

5.4.3 Task Duplication Phase

After the grouping phase, the original task sequence should be generated and all important parameters should be calculated. Once the original task sequence and important parameters are available, we are ready to apply the duplication strategy to complete the last step of the grouping phase. Figure 5.5 illustrates the main idea of the duplication strategy using a simple example. The left part of Figure 5.5 shows a DAG for four tasks with precedence constraints. The execution times of task $T_{1,}$ $T_{2,}$ $T_{3,}$ T_4 are 8s, 10s, 15s, and 6s. The communication times among the tasks are 6s, 5s, 2s, and 4s, respectively. The right part of Figure 5.5 provides three schedules made by the linear scheduling strategy, the non-duplication strategy, and the duplication strategy, respectively. The linear schedule has the longest schedule length because all the tasks allocated to one computing nodes have to be executed in a sequential order. The non-duplication schedule reduces the schedule length by allowing T_2 and T_3 running in parallel on two computing nodes. The duplication schedule further improves the performance by redundantly executing T_1 on the second node. Thus, the duplication strategy trades CPU times for communication overheads.

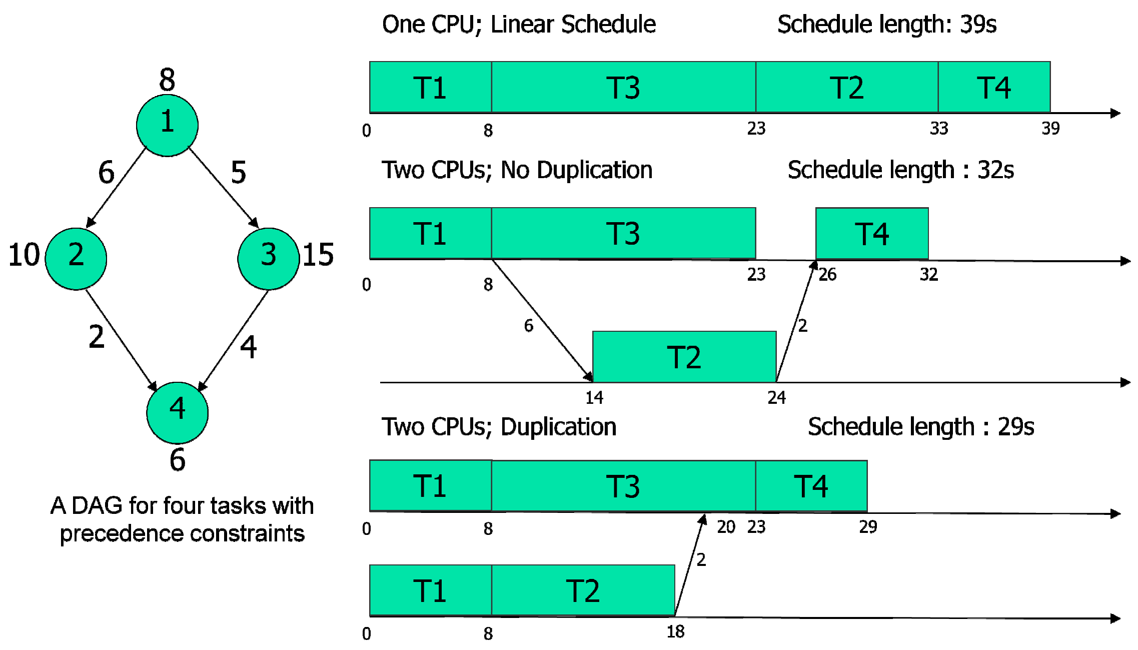


Figure 5.5 An example of duplication scheduling strategy

Figures 5.6 and 5.7 illustrate in details the implementation of EETDS and HEADUS with respect to the process of forming a final task group graph. Initially, no task is marked as "grouped" and the list of clusters is initialized to be empty. Next, the algorithms consider the first task and insert it to a newly formed group called G1. Then in the following iterations, the algorithms go through all the tasks along the favorite predecessor chain, attempting to add all the tasks in the critical path to the same group. Once a task is added to a group, it will be immediately marked as "grouped". If the task being processed is the entry task, the current iteration will end and a new iteration will start in the next loop by choosing the first ungrouped task from the original task sequence generated in step 1. During the process of walking through multiple critical paths, we may find some tasks have been added to a group. At this point, the duplication strategy is responsible to make the decision whether or not to duplicate this task to multiple groups by comparing the value of LAST(t) - LACT(t') and the communication time cc(t, t'). A task will be duplicated if the schedule length can be reduced and the task will not be duplicated otherwise. At the end of the process, the task graph has been divided into groups. Finally, the group graph is generated by creating edges among all the groups communicating with each other. The algorithms then set a weight for each edge to represent corresponding communication cost.

```
1.  $t$ = first waiting task of original task sequence;
2.  $i$ = 1;
1.  add $t$ to $G_i$; /* mark $t$ as "grouped" */
2.  while (not all tasks are grouped) do
3.    $t'$ = $FP(t)$;
4.    if (t' has already been added to one cluster) then
5.      if ($LAST(t) - LACT(t') < cc(t, t')$) then /* if duplicate $t'$, we can shorten the schedule
    length */
6.        add $t'$ to $G_i$; /*duplicate $t'$, mark $t'$ grouped */
7.        if $t$ has another predecessor $z \neq t'$ has not yet been grouped then
8.          $t'$ = $z$;
9.        else
10.         if $t'$ is entry task then
11.           $t'$ = the next task that has not yet been grouped;
12.           $i$++;
13.     else
14.       for another predecessor $z$ of $x$, $z \neq t'$,
15.       if ($ECT(t')+cc_{uv} = ECT(z) + cc(t, t')$) and $z$ hasn't been grouped) then
16.         $t'$ = $z$; /* do not duplicate*/
17.   else allocate $t'$ to $G_i$; /*also mark $t'$ as "grouped" */
18.   $t$ = $t'$;
19.   if $t$ is entry task then
20.     $t$ = the next task that has not yet been added to a group;
21.     $i$++;
22.     assign $t$ to $G_i$; /*also mark $t$ as grouped*/
23. return group graph;
```

Figure 5.6 Pseudo code of the grouping phase in the EETDS algorithm

```
1.  t = first waiting task of original task sequence;
2.  i = 1;
3.  assign t to $G_i$;
4.  while (not all tasks are grouped) do
5.    t' = FP(t);
6.    if (t' has already been added to one cluster) then
7.      if (LAST(t) - LACT(t') < cc(t, t')) then /* duplicate t', we can shorten the schedule length */
8.        moreenergy = $EN_{t'} – EL_{t't}$; /*energy increase*/
9.        if (moreenergy $\le$ threshold T) then /* increased energy less than our threshold*/
10.         add t' to $G_i$; /*duplicate t', mark t' grouped */
11.         if t has another predecessor $z \neq t'$ has not yet been allocated to any node then
12.           t' = z;
13.         else
14.           if t' is entry task then
15.             t' = the next task that has not yet been assigned to a node;
16.             i++;
17.       else
18.         for another predecessor z of t, $z \neq t'$,
19.         if ($ECT(t')+cc_{t't} = ECT(z) + cc_{zt}$) and z hasn't been allocated) then
20.           t' = z; /* do not duplicate*/
21.     else
22.       for another predecessor z of x, $z \neq t'$,
23.       if  (ECT(t')+ cc(t, t') = ECT(z) + cc(t,z)) and z hasn't been allocated) then
24.         t' = z; /* do not duplicate*/
25.   else add t' to $G_i$; /*duplicate t', mark t' grouped */
26.   t = t';
27.   if t is entry task then
28.     t = the next task that has not yet been allocated to a computational node;
29.     i++;
30.    add t' to $G_i$; /*duplicate t', mark t' grouped */
31. return schedule list;
```

Figure 5.7 Pseudo code of the grouping phase in the HEADUS algorithm

The major difference between EETDS and HEADUS is that HEADUS makes tradeoff between energy savings and schedule lengths, in which task duplications are strictly forbidden if the duplications do not exhibit energy conservation (see Steps 9-10). In other words, duplications are infeasible if they result in a significant increase in energy consumption (e.g., the increase exceeds a threshold). In doing so, the HEADUS algorithm ensures that schedule lengths are optimized using task duplication without greatly affecting energy conservation.

5.4.4 Energy-Efficient Group Allocation Phase

After the grouping stage, the DAG has been partitioned into a number of groups, which will be allocated to heterogeneous computing nodes by the next step in an energy-efficient way. The main objective for this phase is to generate an allocation list with minimized energy dissipation. Recall that there might be exclusion relations among some tasks and nodes, e.g. task t_9 couldn't be allocated to p_1 as shown in Figure 5.1. Therefore, we have to verify whether or not a node and a group are exclusive to each other. In other words, we have to assure that all tasks in the group are exclusion compatible with the node to be allocated on. If any task is exclusive to a current node, our algorithm performs the same verification process on another computing node until an exclusion compatible node is identified. In real world clusters, most computing nodes are compatible with various parallel tasks. Otherwise the clusters cannot provide widely used services for end users. To make our algorithm practical, we implement the compatible verification process in our algorithm to handle exclusion issues.

Once a group and a computing node pass the compatible verification process, the group will be temporarily allocated to the node. Next, the algorithms calculate energy consumption caused by the group running on the node. The estimation of the energy consumption can be carried out using the energy consumption model described in Section 3.3. The value of this energy consumption is saved in an energy cost array. The algorithms apply the same procedure to the next type of compatible node. This procedure is repeatedly performed until all candidate compatible nodes with respect to the group have been considered. Finally, the algorithms update the allocation list with a compatible node that leads to the minimized energy dissipation. After the group allocation phase is accomplished, the allocation list provides an allocation solution with minimized energy consumption of the heterogeneous cluster. Figure 5.8 shows the way of implementing the energy-efficient group allocation phase.

```
Energy_Efficient_Allocation () {
    set allocation list is empty;
    for each cluster c in the cluster graph G {
        n = Energy_Efficient_Calculation (c, N);
            mark c is finally allocated to n, update allocation list;
    }
    return allocation list;
}
Energy_Efficient_Calculation (c, N) {
    i = 1;
    while (n is not the last node in N) {
            Legal_n = Exclusion_Verify (c, n);
            Add Legal_n to the Legal_Node_List;
            n = the next node following Legal_n in N ;
    }
    for each node n in Legal_Node_List {
            if (n has not been allocated with any cluster) {
                mark c to be temporarily allocated to n;
           temp_energy_cost[i] = Energy_Consumption(c,n);
          //here Energy_Consumption()will calcutlate energy cost assumming c is allocted to n;
          i++;
            }
    }
    return the node with minimized value in array temp_energy_cost[]
}
Exclusion_Verify (c, n) {
    for each task t in cluster c {
            if (t is exclusive with n) {
                n = the next node following n in N ;
          Exclusion_Verify (c, n);
            }
    }
    return n;
 }
```

Figure 5.8 Pseudo code of group allocation to minimize energy consumption

5.4.5 A Case Study

In this section, we use a synthetic parallel application as an example to illustrate how the EETDS and HEADUS algorithms work. The task graph of the parallel application is delineated in Figure 5.9. The running trace of each step is given as follows:

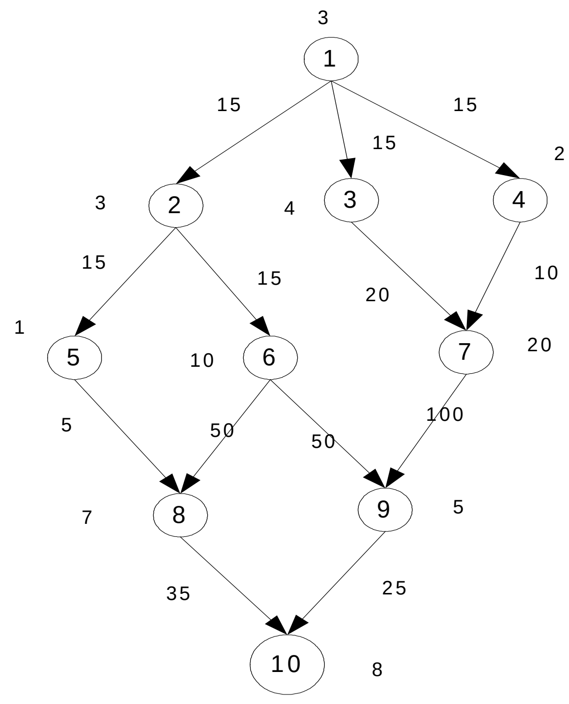


Figure 5.9 A synthetic parallel application

Phase 1. Grouping

<u>Step 1</u>. Generate a task sequence by computing levels: The levels of the tasks can be calculated using Eq. (16). For instance, the level of task v_{10} is 8, since v_{10} is the exit task without any successor. The level of v_8 is 8 + 7 = 15 because v_8 has only one successor task. The level of task v_2 is max$\{L(v_5) + 3, L(v_6) + 3\}$ = 28, since v_2 has two successors -

v_5 and v_6. All the tasks are placed in a queue in the non-increasing order of levels. Thus, we have a list of tasks as {10, 9, 8, 5, 6, 2, 7, 4, 3, 1}

Step 2**.** Calculate the important parameters:

Step 2.1 Compute *EST* and *ECT*: The *EST* and *ECT* values of each task can be computed by applying Eqs. (17) and (18). For example, task v_1 is the entry task and, therefore, $EST(v_1) = 0$. In accordance with Eq. (18), we have $ECT(v_1) = 0 + t_1 = 3$. Since v_2, v_3, and v_4 are unable to start until v_1 finishes and, thus, we have EST(v_2) = EST(v_3) = EST(v_4) = ECT(v_1) = 3. Similarly, EST of v_7 is computed as below

$$\begin{aligned}\mathrm{EST}(\mathrm{v}_7) &= \min\{\max(\mathrm{ECT}(\mathrm{v}_4), \mathrm{ECT}(\mathrm{v}_3) + c_{37}), \max(\mathrm{ECT}(\mathrm{v}_3), \mathrm{ECT}(\mathrm{v}_4) + c_{47})\} \\ &= \min\{\max(5, 7+20), \max(7, 5+10)\} = 15.\end{aligned}$$

Correspondingly, the ECT of v_7 is ECT(v_7) = EST(v_7) + t_7 = 15 + 20 = 35.

Step 2.2 Compute favorite predecessors: The favorite predecessor of a task is determined by using Eq. (19). For example, the favorite predecessor of task v_2, v_3, and v_4 is v_1, simply because these three tasks have only one predecessor. The favorite predecessor of v_8 is v_6 because $ECT(v_6) + c_{68} = 16 + 50 = 66 > ECT(v_5) + c_{58} = 7 + 5 = 12$.

Step 2.3 Compute LAST and LACT: The LACT and ECT values of the exit task v_{10} equal to 79 and, thus, we have $LAST(v_{10}) = LACT(v_{10}) - t_{10} = 79 – 8 = 71$. In case of LACT($v_6$), we have to consider two successors, namely, v_8 (in critical path) and v_9 (not in critical path). We obtain

$\mathrm{LACT}(\mathrm{v}_6) = \min\{\min(\mathrm{LAST}(\mathrm{v}_9) - \mathrm{c}_{69}, \min(\mathrm{LAST}(\mathrm{v}_8)))\} = \min\{(66 - 50), 29)\} = 16$ and $LAST(v_6) = LACT(v_6) - t_6 = 16 – 10 = 6$. Table 5.1 summarizes the values of all the parameters:

Table 5.1 Results of the important parameters

Task	level	EST	ECT	LAST	LACT	FP
1	40	0	3	31	34	--
2	28	3	6	3	6	1
3	37	3	7	42	46	1
4	35	3	5	34	36	1
5	16	6	7	23	24	2
6	25	6	16	6	16	2
7	33	15	35	46	66	3
8	15	16	23	29	36	6
9	13	66	71	66	71	7
10	8	71	79	71	79	9

Step 3**.** Generate a duplication task sequence:

The EETDS algorithm generates the first group of tasks by starting from the first task in the task list obtained in step 1, which is task 10. The first task group containing tasks v_1, v_3, v_7, v_9, and v_{10} forms GROUP 1. Next, the second iteration starts because the algorithm hits task v_1, which is the entry task. At this point, next ungrouped task is task v_8. Tasks v_8, v_6 and v_2 are associated to GROUP 2, and the next task to be considered is task v_1. Since v_1 has been clustered to GROUP 1, the algorithm has to decide whether there is an incentive to duplicate v_1 on GROUP 2. The condition in step 7 (see Figure 5.7) is satisfied, because we have $LAST(v_2) - LACT(v_1) = 3 – 34 < cc_{12} = 15$. Therefore, duplicating v_1 on GROUP 2 gives rise to a shortened schedule length. Thus, GROUP 2 consists of v_8, v_6 , v_2 and v_1. Again, the algorithm hits the entry task and the third iteration is invoked. At this point, task v_5 is added to GROUP 3, followed by task v_2, and v_1, which are not duplicated on GROUP 3 because $LAST(v_5)$ - $LACT(v_2)$ = 23 – 6 = 17 > cc_{12} = 15, which means the schedule length will be increased. Similarly, task v_4 and v_1 are added to GROUP 4 in the last iteration. Finally, the following task groups are created:

Group 1: Task 10, Task 9, Task 7, Task 3, Task 1

Group 2: Task 8, Task 6, Task 2, Task 1

Group 3: Task 5

Group 4: Task 4, Task 1

Accordingly, the final task group generated by HEADUS is like follows (energy threshold T, EN_{active} and EL_{active} are set to 1J, 6J and 1J, respectively):

Group 1: Task 10, Task 9, Task 7, Task 3, Task 1

Group 2: Task 8, Task 6, Task 2

Group 3: Task 5

Group 4: Task 4

Last but not the least, the EETDS and HEADUS algorithms compute the communication cost between each pair of task groups and set the corresponding edges to form a group graph.

Table 5.2 Energy consumption values

	A	B	C	D
C1	3050J	3700J	***2008J***	3000J
C2	1000J	***900J***	1560J	1200J
C3	180J	194J	136J	***75J***
C4	***207J***	226J	251J	243J

Phase 2. Energy-efficient Allocating

In this phase, the EETDS algorithm performs the procedure described in Figure 5.6. Here we just assume that the heterogeneous grid system consists of four types of computing nodes denoted by A, B, C, and D. Energy consumption values of the nodes are listed in Table 5.2:

The final list of allocations determined by the EETDS algorithm is given as follows:

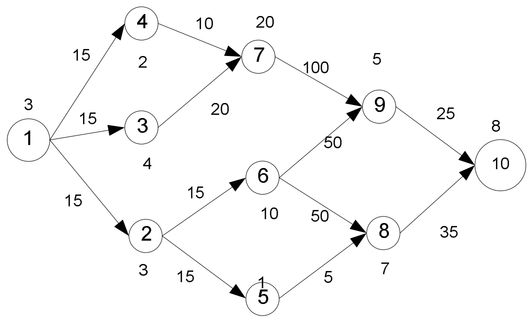
Group 1 is allocated to node C

Group 2 is allocated to node B

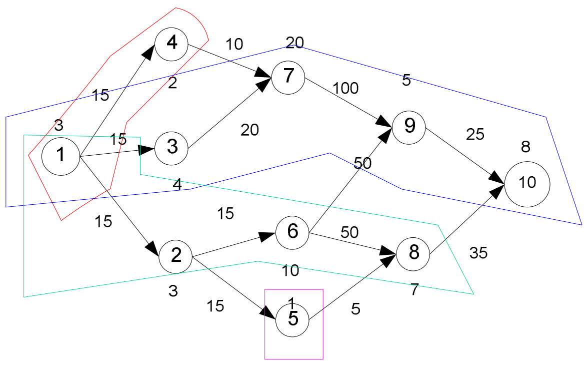
Group 3 is allocated to node D

Group 4 is allocated to node A

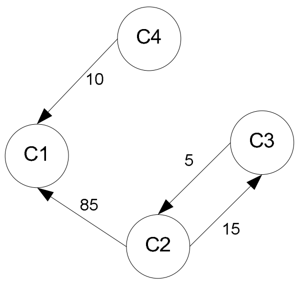
.



(a) The originial task description



(b) The partitioned task graph



(c) The cluster graph

Cluster 1 is allocated to node C
Cluster 2 is allocated to node B
Cluster 3 is allocated to node D
Cluster 4 is allocated to node A

(d) Final allocation list

Figure 5.10 Allocation results showing how the EETDS algorithm works

5.5 Time Complexity Analysis

The time complexity of the EETDS scheduling algorithm is O($|V|^2$).

Proof. The algorithm consists of two major phases: the grouping and energy-aware allocation phases. Let us first analyze the time complexity of each phase.

Let us start from the first step in the grouping phase. In this step, the algorithm traverses all tasks of a DAG to compute the levels of the tasks. The time complexity to calculate the levels is O(|*E*|), where |*E*| is the number of messages. This is because in the worst case all the messages in the DAG have to be examined. Furthermore, it takes

$O(|V|\log|V|)$ time to sort the tasks in an increasing order of the levels, where $|V| = n$ is the number of tasks. Therefore, the time complexity of step 1 is $O(|E| + |V|\log|V|)$.

The second step is performed to obtain all the important parameters like *EST, ECT, FP, LACT,* and *LAST*. Phase 2 calculates these parameters by applying the depth first search with the time complexity of $O(|V| + |E|)$.

In the last step of the grouping phase the tasks are associated into several groups, which can help in reducing schedule length. First, each task is checked and added to one or more groups in the iteration based on the duplication strategy. In the worst case, all the tasks in the critical path must be duplicated, meaning that the time complexity is $O(h|V|)$ time, where h is the height of the DAG. Since h is less than or equal to $|V|$, the time complexity of the third step is $O(|V|^2)$.

Consequently, the total time complexity of these three steps is $O(2|E| + |V|(\lg|V|+1) + |V^2| = O(|E|+|V|^2)$. For a dense DAG, the number of messages are proportional to $O(|V|^2)$. Hence, the time complexities of the grouping phase is $O(|V|^2)$.

In the second phase, the algorithm executes the compatibility verification process and calculates the energy consumption caused by each group on each compatible computing node. Suppose the grouping phase generates a group set $G= \{g_1, g_2, g_3, \dots g_q\}$ with q different groups. We have a heterogeneous node set $P = \{p_1, p_2,..., p_m\}$ with m different type of processors, the algorithm attempts to select two elements randomly from the sets G and P in order to estimate energy cost. According to the permutation and combination theory, the time complexity is $C_q^1 \times C_m^1$. Obviously, the number q of groups is always less than the number of tasks and the number of m is a constant (i.e. the number of heterogeneous nodes in a real cluster). Since the calculation of power consumption for each combination can be completed in linear time, the time complexity of the group allocation phase is O($c|V|$), where c is a constant relies on m and other related calculation time. Similarly, the verification process can be done within O($c|V|$). Therefore, the overall time complexity of the EETDS algorithm is O($|V|^2$), where V is the number of tasks in a parallel task set.

5.6 Simulation Results

In this section, we evaluate the effectiveness of the proposed EETDS and HEADUS scheduling algorithms.

5.6.1 Simulation Metrics and Parameters

We conducted extensive experiments using Gaussian Elimination and Fast Fourier Transform applications. In addition, we compare EETDS and HEADUS with two existing scheduling algorithms: the Non-Duplication Scheduling algorithm (NDS) and the Task Duplication Scheduling algorithm (TDS). We also compare our algorithms with a baseline algorithm: Energy-Efficient Non-Duplication Scheduling (EENDS). The NDS, TDS and EENDS algorithms are briefly described below.

(1) *NDS*: This a non-duplication-based algorithm (also know as the static priority-based Modified Critical Path (MCP) algorithm [52]) with time complexity of O(n^2(log$n$ + m)), where n and m are the numbers of tasks and nodes, respectively. NDS, which does not duplicate any task, makes scheduling decisions using the critical-path method.

(2) *TDS*: The TDS algorithm allocates all tasks that are in a critical path to one computing node. If tasks have already been dispatched to other nodes, TDS only

duplicates the tasks that can potentially shorten scheduling length. TDS aims to generate a schedule of a parallel application with the shortest schedule length.

(3) EENDS: To the best of our knowledge, EENDS is a baseline algorithm that could not be found in the literature. In order to comprehensively understand the impacts of grouping phase, we combine the second phase of our algorithm with the NDS grouping to form a new EENDS scheduling algorithm.

Table 5.3 Characteristics of experimental system parameters

Parameters	Value (Fixed) - (Varied)			
Different trees to be	Gaussian elimination, Fast Fourier Transform			
Execution time of	{5, 4, 1, 1, 1, 1, 10, 2, 3, 3, 3, 7, 8, 6, 6, 20, 30, 30 }-(random)			
Execution time of Fast Fourier Transform	{15, 10, 10, 8, 8, 1, 1, 20, 20, 40, 40, 5, 5, 3, 3 }-(random)			
Computing node type	AMD Athlon 64 X2 4600+ with 85W TDP (Type 1) AMD Athlon 64 X2 4600+ with 65W TDP (Type 2) AMD Athlon 64 X2 3800+ with 35W TDP (Type 3) Intel Core 2 Duo E6300 processor (Type 4)			
CCR set	Between 0.1 and 10			
Computing node heterogeneity	Environment1: # of Type 1: 4 # of Type 2: 4 # of Type 3: 4 # of Type 4: 4	Environment2: # of Type 1: 6 # of Type 2: 2 # of Type 3: 2 # of Type 4: 6	Environment3: # of Type 1: 5 # of Type 2: 3 # of Type 3: 3 # of Type 4: 5	Environment4: # of Type 1: 7 # of Type 2: 1 # of Type 3: 1 # of Type 4: 7
Network energy	20W, 33.6W, 60W			

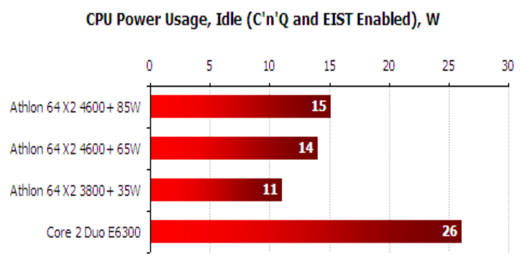
The basic yet important method we used in our experiments is called OTOP (Once Tuning One Parameter). Specifically, in each experimental study we only vary one parameter while keeping the other parameters unchanged. By tuning one parameter at a time, we are allowed to clearly observe its impacts on clusters and easily analyze system sensitivities to this specific parameter. Important system parameters tuned in our experimental studies include Communication-to-Computation Ratio (or CCR for short), network heterogeneity, and computing heterogeneity.

Note that the CCR value of a parallel application on a heterogeneous cluster is defined as the ratio between the average communication cost of |*E*| messages and the average computation cost of *n* parallel tasks in the application on the given cluster with *m* heterogeneous computing nodes. Formally, the CCR of an application *(T*, *E)* is expressed by Eq. (22) as below.

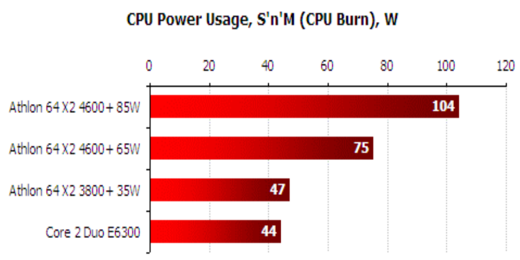
$$CCR(T,E)=\frac{\frac{1}{|E|}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\frac{1}{m(m-1)}\sum_{u=1}^{m}\sum_{v=1,v\neq u}^{m}\frac{s_{ij}}{b_{uv}}\right)}{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{m}\sum_{j=1}^{m}c_i^j\right)}=\frac{\frac{1}{|E|\cdot(m-1)}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\sum_{u=1}^{m}\sum_{v=1,v\neq u}^{m}\frac{s_{ij}}{b_{uv}}\right)}{\frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{m}c_i^j\right)} \qquad (20)$$

Generally speaking, communication-intensive applications have high CCRs, whereas CCRs of computation-intensive applications are low.

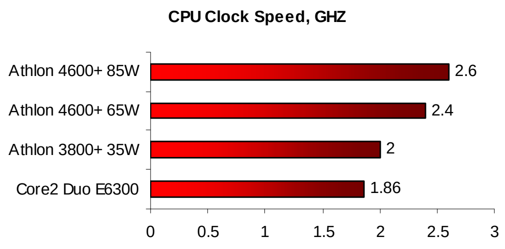
Table 5.3 summarizes the configuration parameters of simulated heterogeneous clusters used in our experiments. On the right hand side of each row in Table 5.3, parameters in the first part are fixed, whereas parameters in the second part are varied or randomly generated using uniform distributions. In order to illustrate the heterogeneity of computing nodes, we choose to test four heterogeneous cluster computing environments, in which the numbers of four types of computing nodes are different in processors. The last row in Table 5.3 represents the network heterogeneity by setting various energy consumption rates. Figure 5.11 shows the energy consumption parameters, CPU speed parameters of different types of processors used in computing nodes. All these data comes from the latest test report of Xbit Lab (**http://www.xbitlabs.com**). Figure 4.6 depicts the task graphs of parallel applications used in our simulation.



(a) CPU Power Consumption Rate (Idle)



(a) CPU Power Consumption Rate (Busy)



(c) CPU Clock Speed (Unit: GHZ)

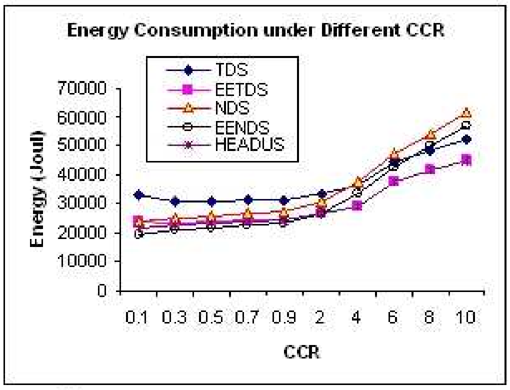
Test platform for CPU energy consumption was built using ASUS M2N32-SLI Deluxe mainboard based on Nvidia nForce 590 SLI chipset, 1GB DDR2-800 SDRAM, PowerColor X1900 XTX 512MB graphics card and Western Digital Raptor WR740GD HDD. The processor was cooled down by Zalman CNPS9500 AM2 air cooler. Intel Core 2 Duo E6300 processor was built using ASUS P5W DH Deluxe mainboard on Intel 975X chipset. The system was also equipped with the same graphics card, memory and hard disk drive. The CPU was cooled down with a similar air-cooling solution from Zalman – CNPS9500 LED.

(d) Test Environment Parameters

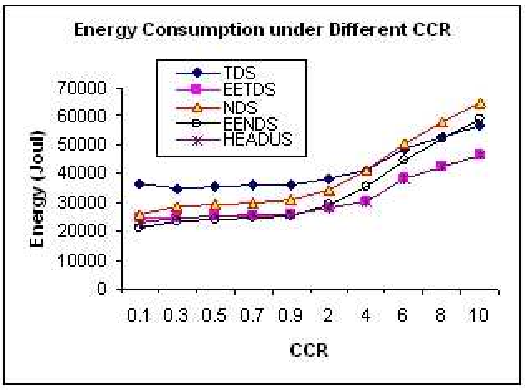
Figure 5.11 Parameters used in simulation (data from test report of Xbit Lab)

5.6.2 Experimental Results for Gaussian Elimination

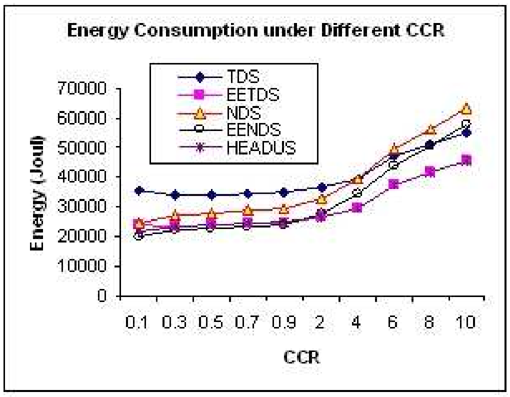
In this subsection, we evaluate five scheduling algorithms using the Gaussian Elimination application on a heterogeneous grid. Figure 5.12 shows the impacts of CCR on energy dissipation of the cluster running the Gaussian Elimination application. Five observations are evident from this group of experiments. First, the energy consumption of Gaussian Elimination under all the five scheduling schemes is very sensitive to CCR. Second, EETDS and HEADUS provide noticeable energy savings compared with the TDS and NDS algorithms. Third, NDS outperforms TDS with respect to energy conservation when the CCR values are small. However, TDS is superior to NDS when CCR becomes large (e.g., CCR is greater than or equal to 4). Fourth, EETDS and HEADUS work well in all these four heterogeneous cluster computing environments. These results demonstrate that EETDS and HEADUS have overall better performance compared with the other three and HEADUS is the best energy-efficient scheduling algorithm among the five examined schemes. Last, the energy savings exhibited by EETDS and HEADUS become more pronounced with the increasing values of CCR. These results indicate that with respect to energy conservation our algorithms are more appropriate for communication-intensive applications as opposed to computation-intensive applications.



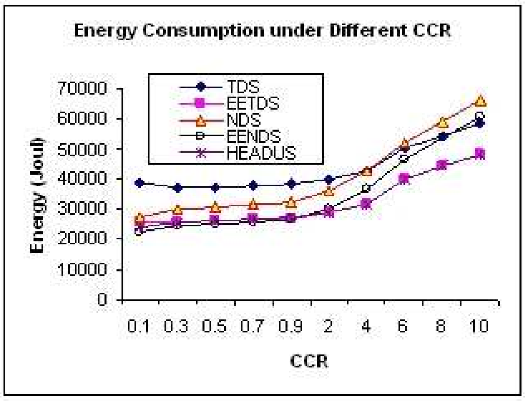
(a) CCR sensitivity under environment1



(b) CCR sensitivity under environment2



(c) CCR sensitivity under environment3

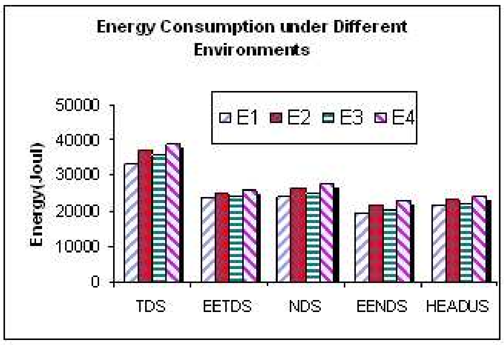


(d) CCR sensitivity under environment4

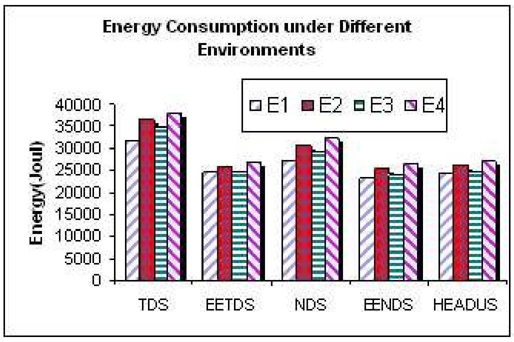
Figure 5.12 CCR sensitivity for Gaussian when Net_Energy=33.6

Since our algorithms are designed for heterogeneous grids, we tested energy dissipation in the four different environments, which are shown in Table 5.3. Figure 5.13 illustrates the impacts of the computing heterogeneity on grid computing platforms. First, we observe that EETDS and HEADUS can conserve more energy in E1 and E3 (see Figs. 5.13(a) and (b) compared with E2 and E4 (see Figs. 5.13(c) and (d), from which we can draw the conclusion that less energy is consumed by clusters with more energy-efficient computing nodes. Second, the computing heterogeneity has significant impacts on the energy efficiency of EETDS. For example, when CCR equals to 0.1 in the four clusters, the EETDS algorithm reduces energy consumption (compared with TDS) by 38.5%, 49.1%, 48.7%, and 51.7%, respectively. These experimental results indicate that EETDS and HEADUS can conserve even more energy for heterogeneous clusters that are comprised of energy-consuming computing nodes. Third, Figure 5.13 shows a similar performance trend that EETDS and HEADUS significantly conserve energy in overall because TDS consumes huge energy when CCR is small and NDS consumes more energy when CCR is large due to the high energy dissipation in the network interconnections.

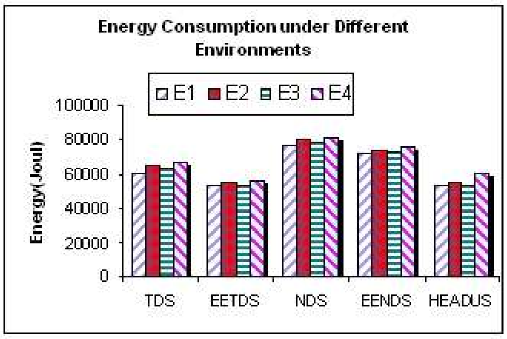
Next, let us quantitatively show the impacts of network heterogeneity on the performance of these five scheduling algorithms. In this group of experiments, we vary network energy consumption rates. Three network energy consumption rates are chosen: 20W, 33.6W, and 60W. It is worth noting that these three energy consumption rates represent real-world network interconnections (e.g. Merinet) widely used in clusters.



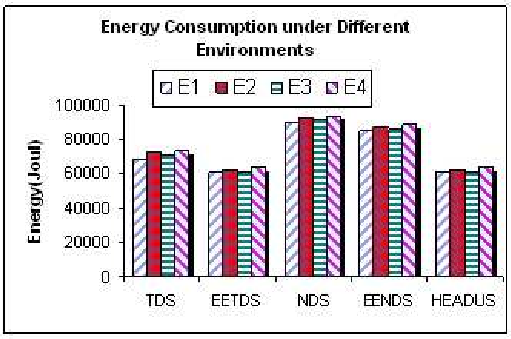
(a) Energy consumption when Net_Energy=60 and CCR=0.1



(b) Energy consumption when Net_Energy=60 and CCR=0.5



(c) Energy consumption when Net_Energy=60 and CCR=8

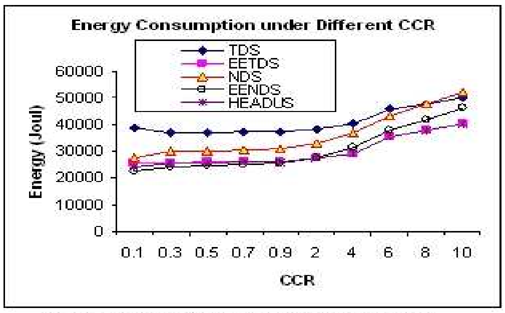


(d) Energy consumption when Net_Energy=60 and CCR=10

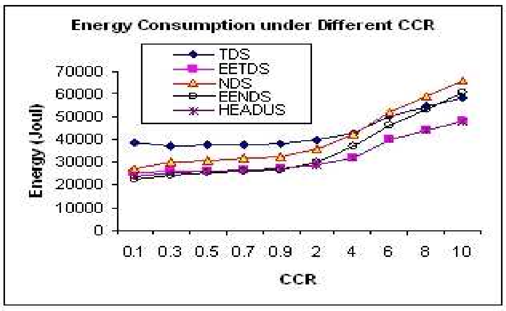
Figure 5.13 Computational nodes heterogeneity experiments

After comparing Figs. 5.14(a), (b), and (c), we can quantify the impacts of network heterogeneity on energy dissipation exhibited by the five scheduling algorithms. For instance, given computing environment 1, EETDS can improve energy efficiency over TDS by 27.9%, 27.9%, 27.8% when network energy consumption rate is 20W, 33.6W, and 60W, respectively (CCR is set to 0.1). However, when CCR is large (e.g., 10), these improvements in energy efficiency are scaled down to 15.6%, 13.3% and 10.2%, respectively. In this set of experiments we confirm that the network energy consumption contributes a whole lot to the grids' total energy consumption when CCR is large. Last, we conclude that NDS is not suitable for communication-intensive parallel applications because NDS has schedule lengths significantly increased when communication overheads are high.

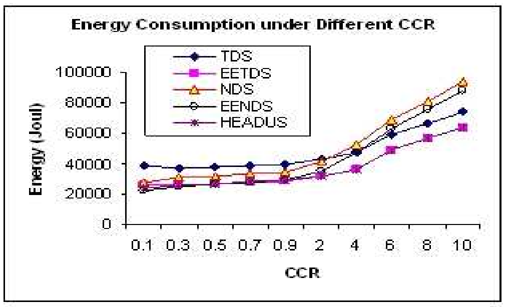
Finally, we illustrate the energy threshold sensitivity of HEADUS algorithm in Figure 5.14(d). We did this simulation by setting threshold as 100J, 500J and 1kJ under different CCRs in environment 4 when Net_Energy consumption rate is set to 60W. Our conclusion is that energy threshold does affect the performance of HEADUS. More specifically, HEADUS is very sensitive to threshold, especially when the energy consumption of related CPU and links is comparable with the energy threshold.



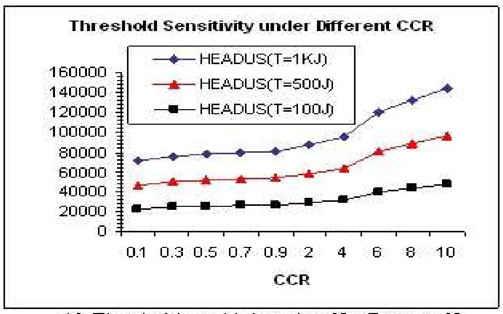
(a) Energy consumption when Net_Energy=20



(b) Energy consumption when Net_Energy=33.6



(c) Energy consumption when Net_Energy=60

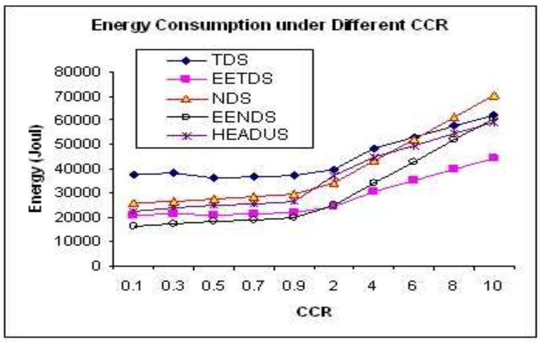


(d) Threshold sensitivity when Net_Energy=60

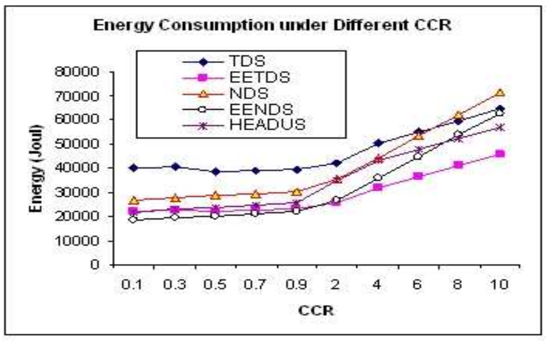
Figure 5.14 Network heterogeneity and threshold sensitivity experiments

5.6.3 Experimental Results for Fast Fourier Transform

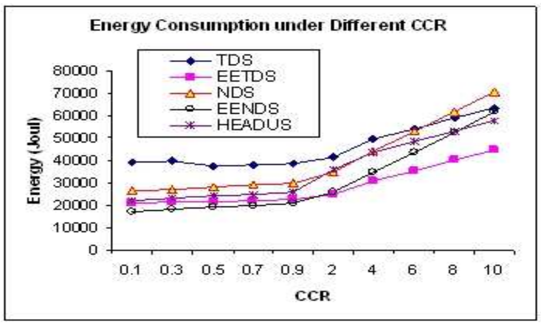
The goal of this group of experiments is to compare the performance of the proposed EETDS and HEADUS algorithms with the NDS, TDS and EENDS algorithms with respect to energy conservation under the FFT application. First, we are focused on relationships between CCR and energy consumption of the FFT application. Figure 5.15 plots total energy consumption of the four heterogeneous clusters running FFT. CCR is gradually varied from 0.1 to 10. Figure 5.15 shows that the total energy consumption caused by the FFT application becomes more sensitive to CCR when CCR is greater than 2. Compared with the TDS algorithm, EETDS conserves approximately 46% and 31% energy when CCR is small and large in environment 4. Accordingly, HEADUS conserves roughly about 47% and 17% respectively. Also, EETDS and HEADUS outperform NDS with 17.5% & 19.5% for small CCRs and 34.7% & 20.5% for big CCRs. Therefore, we can see that HEADUS is more appropriate for computation intensive application and EETDS works better in highly communication intensive applications. When CCR is greater than 8, even EENDS consumes more energy because the first grouping phase in EENDS generates groups that have high communication overheads.



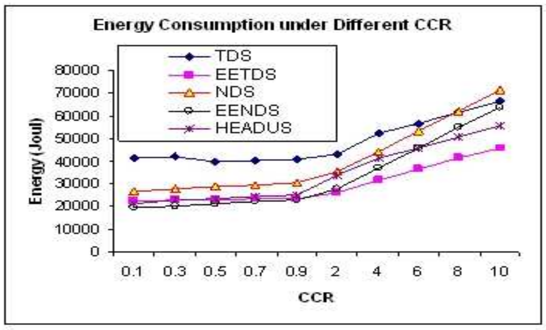
(a) CCR sensitivity under environment1



(b) CCR sensitivity under environment2



(c) CCR sensitivity under environment3

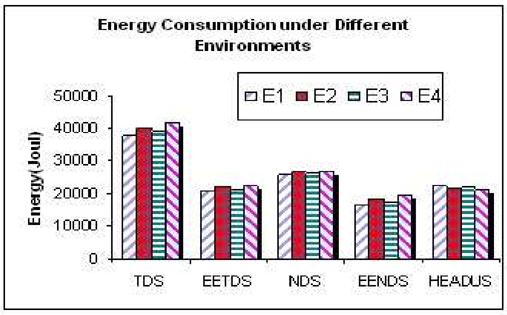


(d) CCR sensitivity under environment4

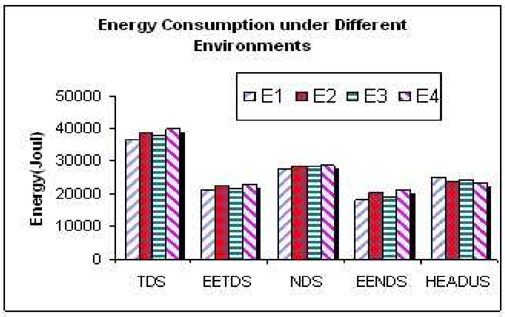
Figure 5.15 CCR sensitivity for FFT when Net_Energy=20W

Moreover, Figure 5.15 shows that when CCR is relatively small, energy consumption under the TDS algorithm is noticeably higher than those under the other four algorithms. This is mainly because energy dissipation in the network interconnections is diminished with a small CCR. Not surprisingly, EETDS improves energy efficiency over NDS up to 50% when CCR is large (e.g., CCR = 10).

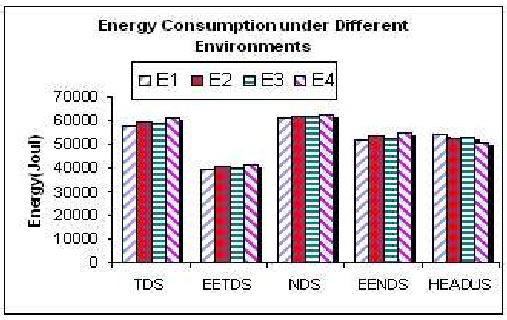
Now we evaluate the impacts of computing heterogeneity using the FFT application. Experimental results in terms of energy efficiency are depicted in Figure 5.16. For all four cluster computing environments, EETDS and HEADUS significantly improves energy efficiency over the three alternative scheduling algorithms (see Figure 5.16). These results coupled with the results plotted in Figure 5.15 confirm that regardless of the heterogeneities and CCR values, our algorithm are consistently the most energy-efficient scheduling algorithm among all the five examined schemes.



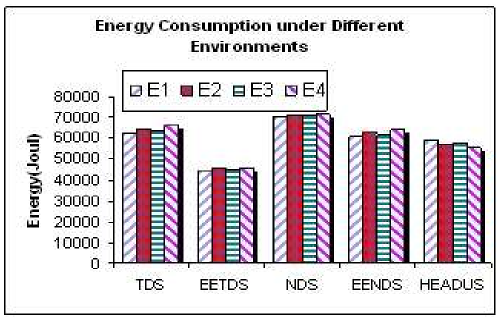
(a) Energy consumption when Net_Energy=20 and CCR=0.1



(b) Energy consumption when Net_Energy=20 and CCR=0.5



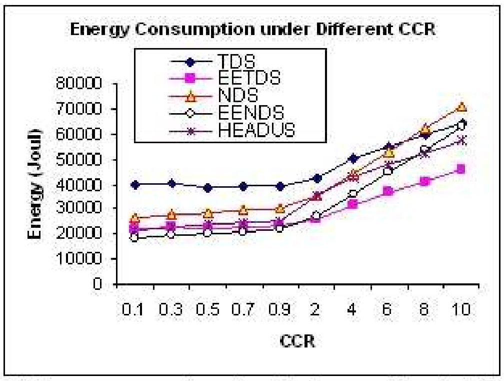
(c) Energy consumption when Net_Energy=20 and CCR=8



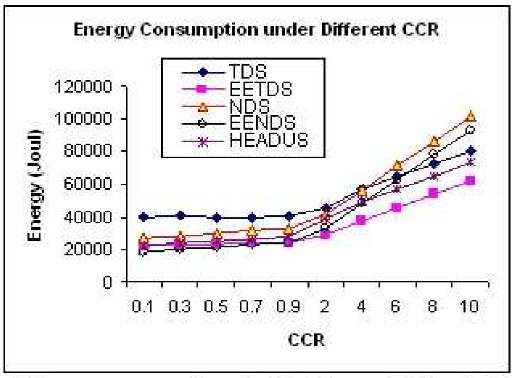
(d) Energy consumption when Net_Energy=20 and CCR=10

Figure 5.16 Computational nodes heterogeneity experiments for FFT

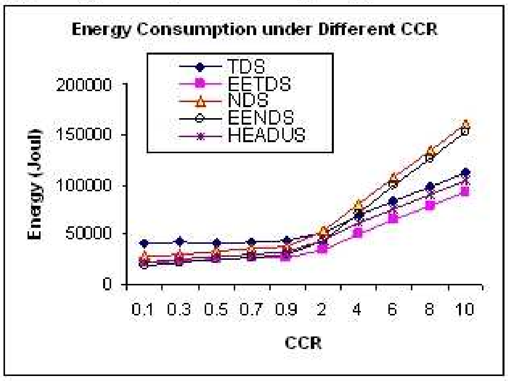
Figure 5.17 shows the impacts of network heterogeneity on the energy consumption experienced by the four scheduling algorithms. Comparing Figs. 5.17(a), (b), and (c), we observe that the impacts of network heterogeneity are highly dependent on CCR. Energy consumption cased by network interconnections account for the major portion of the energy dissipation in the clusters under the condition that CCR is large.



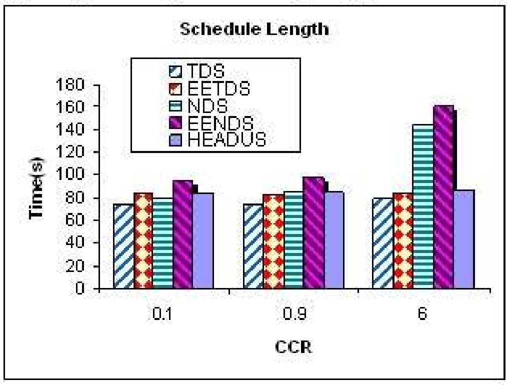
(a) Energy consumption when Net_Energy=20 under E2



(b) Energy consumption when Net_Energy=33.6 under E2



(c) Energy consumption when Net_Energy=60 under E2



(d) Schedule Length Evaluation for Gaussian Elimination

Figure 5.17 Network heterogeneity for FFT and schedule length for Gaussian

5.6.4 Experimental Results of Schedule Length

Schedule length is one the of most important performance metrics. Our algorithms are conducive to conserve energy without significantly degrading performance. In this set of experimental results, we will evaluate the impact to schedule length. Figure

5.17(d) summarizes empirical results based on the Gaussian Elimination application. Figure 5.17(d) reveals that both EETDS and HEADUS have only a marginal performance degradation compared with TDS. That is partially because the four types of processors used in the computing nodes consume more energy if they run at full speed. Therefore, EETDS and HEADUS are forced to sacrifice performance to some extent by allocating parallel tasks to energy-efficient computing nodes. Although EETDS and HEADUS increase schedule length by an average of 9% and 10% compared with TDS, EETDS and HEADUS do conserve energy by an average of 32% and 34%. Nevertheless, the performance degradation problem can be remedied by the advancement of hardware technology (e.g., high CPU capacity and high CPU energy efficiency).

5.7 Summary

In this chapter, we addressed the issue of allocating and scheduling tasks of parallel applications running on heterogeneous grids in a way to conserve energy without adversely affecting performance. Specifically, we proposed two novel scheduling algorithms called EETDS and HEADUS, which aim to make the best tradeoffs between energy savings and performance for tasks of parallel applications running on heterogeneous clusters. EETDS and HEADUS are designed and implemented based on the previous algorithms used in chapter 4 for homogeneous clusters. Both the EETDS and HEADUS algorithms consist of two major phases. In the first phase, a grouping method is employed to minimize schedule lengths of parallel applications. The goal of phase two is to leverage energy-consumption parameters to achieve high energy efficiency.

The experimental results show that compared with TDS, NDS and EENDS, EETDS and HEADUS can significantly reduce energy consumption in heterogeneous grids with only a marginal degradation in performance.

Chapter 6

Energy-Efficient Storage Systems

In the previous two chapters, we have addressed the energy conservation issue for high-performance cluster and grid systems through energy-efficient scheduling. These scheduling algorithms primarily consider the energy consumed by CPU and interconnection. The significantly energy consumed by storage systems has not been discussed.

In this chapter, we address the energy conservation issue for large-scale storage systems by proposing buffer disk based architecture and designing energy-aware resource management strategy.

The rest of this chapter is organized as follows. In section 6.1, we present the motivation of this study. Section 6.2 illustrates the buffer-disk based parallel disks architecture. In section 6.3, we demonstrate the heat-based load balancing strategy. Mathematical models for calculating the power of parallel storage systems are explained in section 6.4.The experimental environment and simulation results are presented in section 6.5. Finally, section 6.6 will summarize the primary contribution of this chapter and future research directions.

6.1 Motivation

Storage systems are considered as one of the major energy consumer in most high performance computing platforms. That is mainly because most high-performance computing servers have to storage and process massive data. Historically, tape libraries are preferred over disk arrays for massive storage environments, in large part due to the capacity and cost differential between tapes and disks. Over the last decade the original tape systems have been gradually replaced by parallel disk systems because of the continuous expansion of disk capacity and continuous drop of disk price. However, large-scale parallel storage systems consume significant amount of energy. A recent industry report shows that storage devices account for almost 27% of the total energy in a data center [40]. Therefore, new technologies focused on the design of energy-efficient parallel storage systems are highly desirable.

In this chapter, we present a buffer disk (BUD for short) based architecture to build energy efficient parallel storage systems. The basic idea of BUD is simple and straightforward. To most people, it is common sense that leaving a light bulb on at daytime is a waste of energy. The same thing happens if we keep the disks on when it does nothing. It makes no sense that we still feed those idle disks power, without producing any useful work. The primary design goal of BUD is to conserve energy by serving most of the requests in a small number of buffer disks and turning as many idle disks as possible to a low power mode. Nevertheless, one potential problem of the BUD architecture is that a limited number of buffer disks may easily become the bottleneck. Worst case access patterns can direct all requests to a single buffer disk, resulting in arbitrarily large delays for very small arrival rates. Therefore, we also designed the heat based load balancing strategy for BUD in order to improve the performance.

6.2 Buffer-Disk Architecture

The buffer disk architecture (see Figure 6.1) consists of four major components: a RAM buffer, *m* buffer disks, *n* data disks, and a buffer-disk controller. The buffer disks temporarily cache the requests for the data disks. Data disks remain in low power mode unless a read request misses in the buffer disk or the write log for a specific data disk grows too large. The buffer-disk controller is the "brain" of the whole system, which contains the energy-related reliability algorithms, data partitioning algorithms, data movement/placement strategies, and prefetching strategies. Our ultimate objective in this research is to conserve more energy without adversely affecting the performance of the disk system. More specifically, the controller strives to move the frequently accessed data from data disks into buffer disks. This allows as many data disks as possible to switch into low-power modes. The rationale behind this strategy relies on the fact that only a small percentage of the data is frequently accessed in a wide variety of data-intensive applications [59]. To achieve this goal, we proposed the heat-based algorithm to dynamically balance the workload. This algorithm aids in avoiding the potential "traffic jam" caused by over loaded buffer disks. Here we want to note that all our solutions and experimental results illustrated in the following sections are primarily based on read requests.

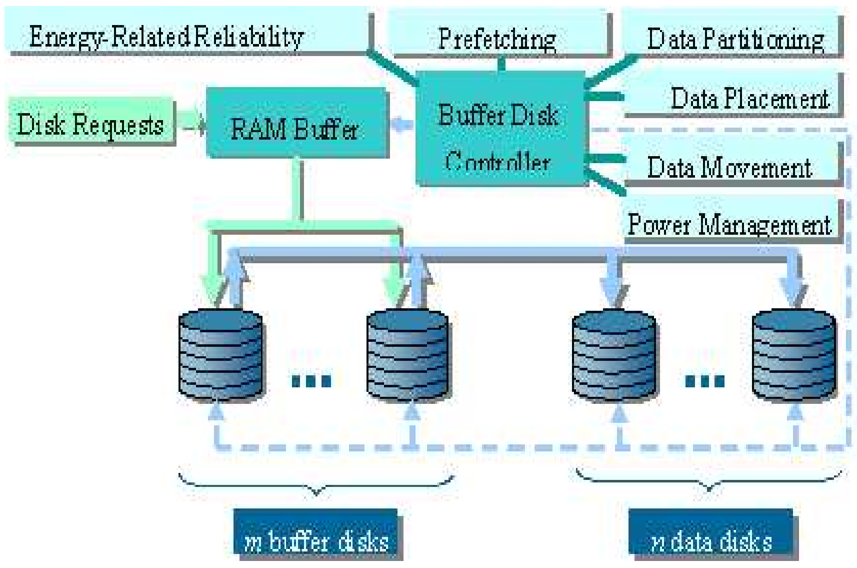


Figure 6.1 The buffer disk architecture

6.3 Heat-Based Load Balancing

To conserve energy, most data disks will run in the low power state and all the traffic will be directed to a limited number of buffer disks. This can potentially make the buffer disks overloaded and they may become the system bottleneck and degrade the system performance. Load balancing is one of the best solutions for the inherent shortcoming of the BUD architecture. Basically, there are three types of load balancing strategies called non-random load balancing, random load balancing, and redundancy load balancing. Sequential mapping belongs to non-random load balancing because the buffer disks have fixed mapping relationship with specific data disks. The round-robin mapping is a typical random load balancing strategy by allocating data to each buffer disk with equal portions and in order. Redundancy load balancing strategies for storage systems include EERAID [60], eRAID [61], and RIMAC [62]. In this section, we will propose a heat-based load balancing strategy, which also belongs to random load balancing strategy. The primary objective of our strategy is to keep all buffer disks as equally loaded as possible and to minimize the overall response time of all requests.

6.3.1 A Concrete Example

Before we start discussing our proposed heat-based load balancing algorithm, we will demonstrate a concrete example. In it some buffer disks are over loaded, thus degrading the performance of the whole system.

Suppose we have 15 requests cached in the RAM buffer and they are going to be dispatched to different buffer disks by the controller. Requests have different colors, which represent that they will access different data blocks. For example, request 1(white) will access data block 1 (white) existing in data disk 1 and request 6 (green) will access the data block 6 (green) existing in data disk 6. Figure 6.2 illustrates the dispatching results of the sequential mapping algorithm, which is a typical non-random load balancing strategy. In the sequential mapping strategy, a buffer disk will only cache the data coming from specific data disks in a sequential way. For instance, the data in data disk 1 and data disk 2 will only be copied to buffer disk 1 and similarly, buffer disk 3 will only cache the data coming from data disk 5 and data disk 6. Figure 6.2 shows that the three buffer disks are not well load balanced because buffer disk 1, 2, and 3 serve 9 requests, 3 requests and 3 requests respectively. Obviously, buffer disk 1 has become the bottleneck whereas the other two buffer disks are only slightly loaded. Round robin mapping is a typical random load balancing strategy. Figure 6.3 illustrates the scheduling results of the round robin mapping, in which data blocks are cached to the buffer disks in a round robin way. We can see that buffer disk 1, 2, and 3 are allocated 7 requests, 5 requests and 3 requests respectively. Although we get better results as compared to sequential mapping, three buffer disks are still not well balanced. It is highly possible that buffer disk 1 could cause performance degradation when more requests are processed.

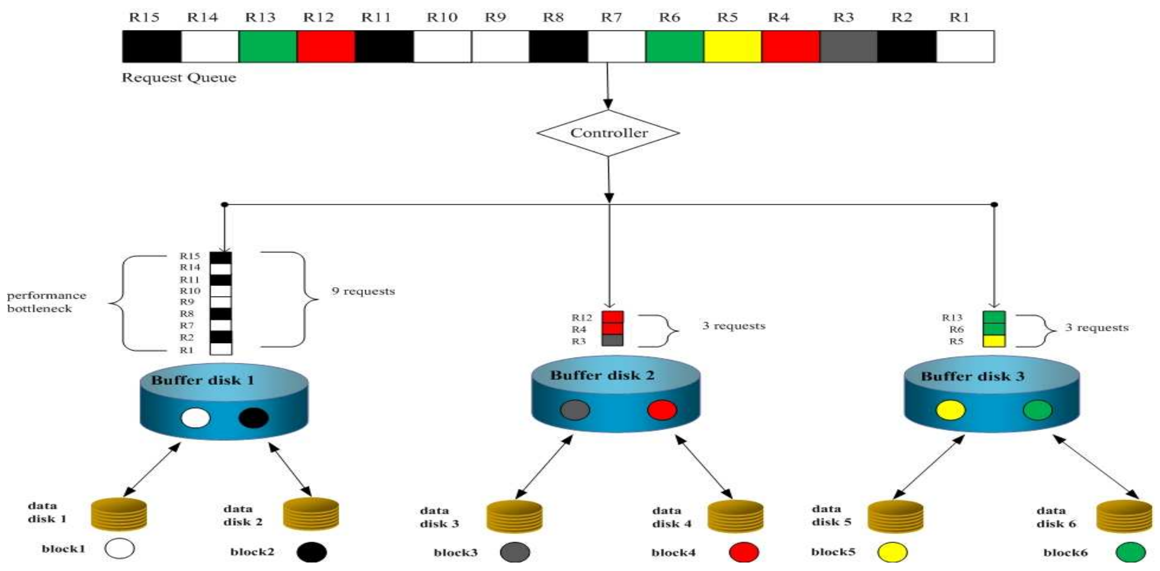


Figure 6.2 Allocation results of sequential mapping strategy

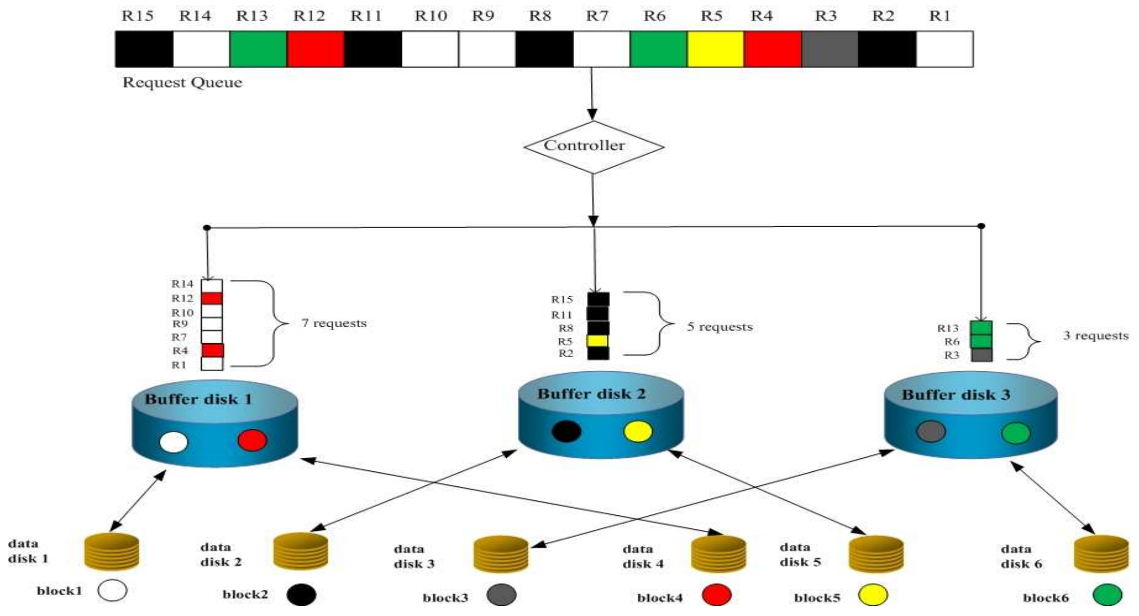


Figure 6.3 Allocation results of round-robin mapping strategy

6.3.2 Heat-based Load Balancing Algorithm

In contrast with sequential and round robin mapping algorithms, we proposed a heat-based mapping strategy to achieve load balancing. The basic idea of heat-based mapping is that blocks in data disks will be mapped to buffer disks based on their heat. Our goal is to make the accumulated heat of data blocks allocated to each buffer disk the same or close to this ideal situation. In other words, the temperature, or the workload of each buffer disk should be the same. The temperature of a buffer disk is the total heat of all blocks existing in the buffer disk. For example, if we suppose all blocks have the same data size, the heat of blocks 1-6 is 5, 4, 1, 2, 1, and 2 respectively. Therefore, block 1 is cached to buffer disk 1, block 2 and 3 are copied to buffer disk 2 and block 4, 5 and 6 are mapped to buffer disk 3. With this mapping the temperature of each buffer disk is 5. Figure 6.4 depicts the dispatching results of the heat-based load balancing strategy.

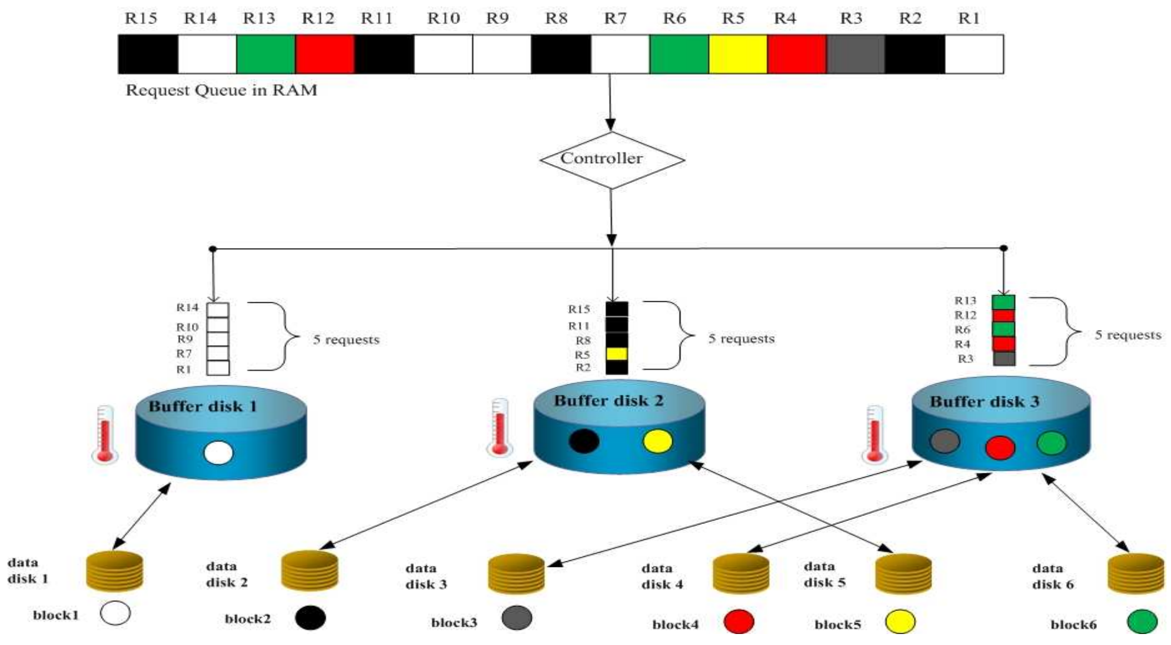


Figure 6.4 Allocation results of heat-based mapping strategy

To clearly describe our heat-based load balancing algorithm, we define the key parameters as follows.

Access Frequency: times a data block is accessed within a specific time unit.

Heat weight: the ratio of requested data size and standard data size (1MB)

Heat: the multiplication of access frequency and heat weight

Temperature: the accumulated heat of all data blocks existing in a buffer disk

The heat could be used to measure the popularity of a data block and the temperature clearly shows how busy a buffer disk is. To calculate the heat more accurately, we need to consider the impact of block size. A large block size should have higher heat compared to a small block with the same access frequency. This is due to the fact that the system will spend more time to complete the response operation for the larger block. In other words, the larger blocks should have higher heat weight.

Since our algorithm is executed online, dynamic tracking of the heat of blocks is crucial. We implemented two strategies to dynamically track the heat. In the first strategy, the controller will snapshot the first *k* requests of the request queue and run the heat calculation function. Once the *k* requests are captured, they will be removed from the main request queue in the memory. We call these *k* requests a snapshot request window and this window will be the input of the heat-based load balancing algorithm. However, the snapshot window strategy is only suitable for bursty request patterns but not for sparse request patterns. When a sparse request pattern is encountered, it may take too long to collect a snapshot window of *k* requests. The response time suffers if we do not serve the requests until all *k* requests ready. Therefore, we designed a second strategy called the observation time window strategy. In this strategy, the controller will serve the requests that arrive within a specific observation time, T seconds, no matter how many requests arrive. That means, the maximum waiting overhead for each request is T.

```
1.  Input:  the request window ;            /*  request window  will be updated periodically */
2.  for each unique target block in the queue       /* each request has a target block to be accessed */
3.      AF = Access_Frequency_Calculation() ;              /* calculate the block access frequency*/
4.      HW = accessed block size/ standard block size;                          /*calculate the heat weight*/
5.      heat = AF * HW;                                                          /*calculate the heat */
6.  sort the data blocks based on heat and save them in Linklist_Block; /* first block has the highest heat */
7.  sort the buffer disk based on current temperature to a Linklist_Buffer ;/* first disk has lowest temperature*/
8.  pointer p_buffer = the first buffer disk in the Linklist_Buffer;
9.  pointer p_block = the first block in the Linklist_Block;
10. pointer t_buffer ;  /* t_buffer points to the buffer disk which have the copy of target block*/
11. for each block in the Linklist_Block
12.     if (p_block.found = = false)                     /* the target block cannot be found in buffer disks*/
13.         if (p_buffer. free = = true)          /* the candidate buffer disk has enough space*/
14.             wake up the corresponding data disk  and cache the data;
            /* The data blocks within the batch prefetching window will be copied to the buffer disk p_buffer;

15.             dispatch all requests accessing p_block to p_buffer;
16.             recalculate and update the information of block heat and buffer disk temperature ;
17.         else /* the first candidate buffer disk has no space*/
18.             if (p_buffer.next != empty)
                    p_buffer ++;                /* seek another candidate buffer disk*/

19.                 go to setp 12;
20.             else /* all buffer disks are already full*/
21.                 reset p_buffer to the first buffer disk in the Linklist_Buffer;
22.                 data_replace_function(p_buffer);  /* replace existing data blocks using LRU algorithm */
23.                 dispatch all requests accessing p_block to p_buffer;
24.                 recalculate and update the information of block heat and buffer disk temperature ;
25.     else /* p_block is found in one buffer disk t_buffer */
26.         dispatch all requests accessing p_block to t_buffer ;
27.         recalculate and update the information of block heat and buffer disk temperature ;
```

Figure 6.5 Heat-based load balancing algorithm

Figure 6.5 outlines the pseudo code of the heat-based load balancing algorithm. We should note that the request window in the first line could represent the snapshot window or the observation time window. This algorithm will periodically collect the requests waiting in the queue, analyze the target block of each read request, and calculate the heat of each unique block. If the target block cannot be found in the buffer disk, the controller will send a data miss command. This will wake up the corresponding data disk and copy the block to the buffer disk that has the lowest temperature. In a special case, the selected buffer disk may not have free space to store a new data block. The controller will seek the next buffer disk with a temperature that is higher than the initial buffer disk selected, but still lower than any other buffer disk. In the worst case, no candidate buffer disk will be found because all buffer disks are full. A data replacement function based on the LRU algorithm will be executed to replace some existing data blocks. If the target block has already been cached in one of the buffer disks, that buffer disk will serve the corresponding request. Once the algorithm has made the decision how to dispatch these requests, the block heat and buffer disk temperature will be recalculated and updated accordingly. Since this is an online algorithm, the decision made at the current time period relies on the heat and temperature information collected at the last time period.

6.4 Energy Consumption Models

In order to compare the energy efficiency of the BUD architecture with disk arrays without buffer disks, we define the energy consumption model in this section. As we all know, the states of a disk (either a buffer disk or a data disk) include active, sleep, idle, or shut down. Some modern disks even have different energy consumption modes for the active state (different rotation speeds), in our study we only consider the active, idle, and sleep states in this study to simplify the problem. The core of our power model used in our simulator is a summation of all power states multiplied by the time each power state was active. In addition, the power state transition overhead is also considered and added to the total energy consumption of BUD. Moreover, we suppose the buffer disk will never enter the sleep state. Therefore, the buffer disks only have two states, active and idle. Similarly, the data disks will either be active when they are copying data to buffer disks or sleeping when no data access is required. In what follows, a series of functions are presented to formally illustrate how we calculate the energy consumption of the BUD architecture. The calculation for traditional parallel disk arrays is trivial and ignored here.

We denote the energy consumption rates of the disks when they are in active, idle and sleep mode by P_{active}, P_{idle}, and P_{sleep}, respectively. Similarly, let T_{active}, T_{idle} and T_{sleep} be the time intervals when the disk is in the active, idle and sleep states, respectively. Hence, the energy dissipation E_{active} of the disk when it is in the active state can be written as $P_{active} \cdot T_{active}$. Similarly, the energy E_{idle} of the disk when it is sitting idle and the energy E_{sleep} of the disk when it is sleeping can be expressed as $P_{idle} \cdot T_{idle}$ and $P_{sleep} \cdot T_{sleep}$ respectively. In addition to that, we denote E_{tr} as the energy consumption overhead when disks transit from one state to another and N_{tr} indicates how many times a disk transits its power state. Now the total energy consumed by each buffer disk can be calculated as

$$\begin{aligned} E_{buffer} &= E_{tr} \times N_{tr} + E_{active} + E_{idle} \\ &= E_{tr} \times N_{tr} + P_{active} \cdot T_{active} + P_{idle} \cdot T_{idle} \end{aligned} \quad (21)$$

In a similar way, the total energy consumed by each data disk can be calculated as

$$\begin{aligned} E_{data} &= E_{tr} \times N_{tr} + E_{active} + E_{sleep} \\ &= E_{tr} \times N_{tr} + P_{active} \cdot T_{active} + P_{sleep} \cdot T_{sleep} \end{aligned} \quad (22)$$

Although we use the same term E_{tr} in both equations, the value of E_{tr} is different because the energy overhead for transitions between the active, idle, and sleep states are different. The energy values for each of the previously mentioned transitions are made explicit in Table 2. The time interval T_{active} when the disk is in the active state is the sum of serving times of disk requests submitted to the disk system.

$$T_{active} = \sum_{i=1}^{n} T_{service}(i), \quad (23)$$

where n is the total number of submitted disk requests, and $T_{service}(i)$ is the serving time of the ith disk request. $T_{service}(i)$ can be modeled as

$$T_{service}(i) = T_{seek}(i) + T_{rot}(i) + T_{trans}(i). \quad (24)$$

where T_{seek} is the amount of time spent seeking to the desired cylinder, T_{rot} is the rotational delay and $\mathrm{T}_{\mathrm{trans}}$ is the amount of time spent actually reading from or writing to the disk.

Suppose there are a total of m buffer disks and n data disks in the BUD parallel storage systems, now we can quantify the total energy with the equation below

$$E_{total} = \sum_{i=1}^{m} E_{buffer}(i) + \sum_{i=1}^{n} E_{data}(i) \quad (25)$$

6.5 Simulation Results

In this section, we present the performance evaluation of the BUD parallel disk system and the heat-based load balancing algorithm proposed above. To simulate the BUD architecture, we implemented our simulator, called BUD_Sim, using Java language. We tried our best to consider and incorporate as many details of real disks as possible. For example, we calculate the seek time as a non-linear function (Table 6.1) of the seek distance using the seek-time-versus-distance curve presented in [63].

Table 6.1 Seek time calculation

Seek distance	Seek time (ms)
< 616 cylinders	$3.45 + 0.597\sqrt{d}$
≥ 616 cylinders	$10.8 + 0.012 d$

In addition, we have implemented a load generator, which can generate synthetic workloads according to specified parameter distributions, or analyze and filter real traces and feed them as the input to BUD_Sim. Using the generator, we could easily control and systematically tune all relevant parameters of a workload based on our evaluation requirements.

Another important decision for implementing BUD_Sim is the type of hard disk drives we should simulate. We believe that the buffer disks should have higher performance (e.g. short seek time, high rotation speed) compared with data disks. Consequently, buffer disks are more expensive and cost higher energy. It is still worthwhile to use higher performance buffer disks because the number of buffer disks is limited compared with the number of data disks. We will have an overall optimal performance/cost rate. In BUD_Sim, the high-performance IBM disk, IBM 36Z15 Ultrastar, serves as the buffer disk and the low performance disk, IBM 73LZX Ultrastar, serves as the data disk. Table 6.2 illustrates the detailed parameters of these two types of disks, which are from IBM manuals and power measurements published in [64]. In Table 6.3, we summarize the important parameters that have been used in our simulation.

Table 6.2 Hardware characteristics of disks

Parameters	IBM 36Z15 Ultrastar (high perf.)	IBM 73LZX Ultrastar (low perf.)
Standard interface	SCSI	SCSI
Number of platters	4	2
Rotations per minute	15000	10000
Average seek time	3.4 ms	4.9 ms
Average rotation time	2 ms	3 ms
Transfer rate	55 MB/sec	53 MB/sec
Power (active)	13.5 W	9.5 W
Power (idle)	10.2 W	6.0 W
Power (sleep)	2.5W	1.4W
Energy (spin down)	13.0 J	10.0 J
Time (spin down)	1.5 s	1.7 s
Energy (spin up)	135.0 J	97.9 J
Time (spin up)	10.9 s	10.1 s

Before the simulation results are discussed, we briefly outline the baseline parallel storage system and load balancing algorithms. They are used for comparison with our proposed BUD architecture and heat-based load balancing algorithm. In section 6.3.1 and 6.3.3, where we compare the energy consumption and response time, the baseline parallel storage system has no buffer disks. All disk drives greedily serve the requests in order to shorten the response time, i.e. disks only have active and idle modes and will never sleep. Therefore, the energy and time overhead caused by spin-up and spin-down could be avoided. In section 6.3.2, the other two baseline algorithms, called sequential mapping and round robin mapping, are compared against the proposed heat-based mapping algorithm. Please refer to section 6.2.1 for more detailed information about these two mapping strategies.

Table 6.3 Important parameters

Parameters	**Range/Value**
# of requests:	{2000,5000,10000,20000}
# of buffer disks	3
# of data disks	30
data block size	{64KB, 1MB, 4MB, 64MB}
average interval (light load trace)	2.5s
average interval (heavy load trace)	0.5s

6.5.1 Evaluation of Energy Consumption

This set of experimental results aims at evaluating the energy efficiency of the buffer disk based parallel storage systems. To fairly compare the results, we generated and executed a large number of requests and simulated both large reads (average data size is 64MB) and small reads (average data size is 64KB). Figure 6.6 and Figure 6.7 plot the total energy consumption of NO-buffer and Heat-BUD running 2000, 5000, 10000, and 20000 large read requests and small read requests, respectively.

There are three important observations here. First, the BUD can significantly conserve energy compared with No-Buffer parallel storage systems. Second, the more requests BUD serves, the more potential power savings is revealed. For example, BUD outperforms No-Buffer in terms of energy conservation by 75.83%, 77.89%, 80.18% and 81.16% for 2000, 5000, 10000, and 20000 large reads respectively. This is expected because more requests lead to more opportunities for BUD to keep the data disks in sleep mode. Third, BUD performs better for small reads (average 84.4% improvement) than large reads (average 78.77% improvement). The rationale behind is that BUD will consume more energy when moving large data blocks to buffer disks.

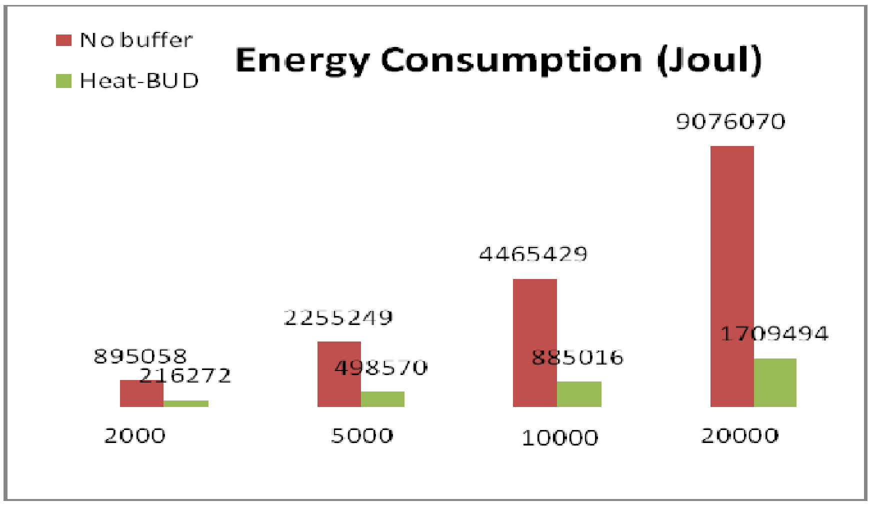


Figure 6.6 Energy consumption for large reads

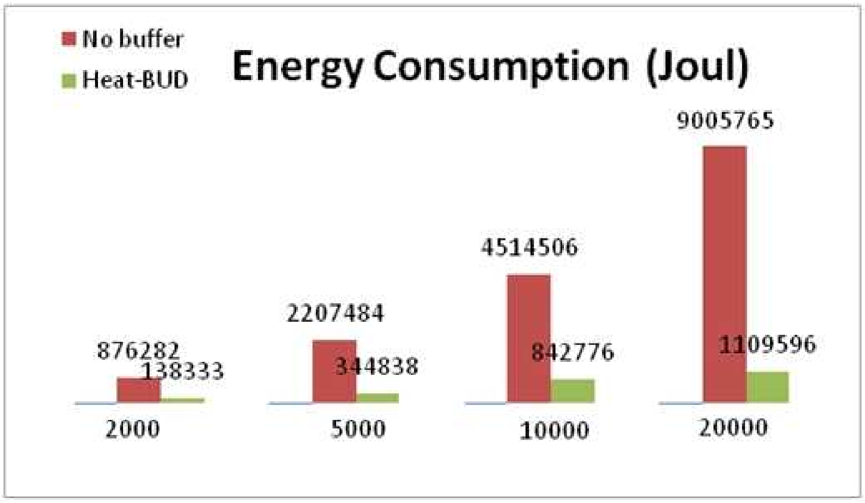


Figure 6.7 Energy consumption for small reads

6.5.2 Evaluation of Load Balancing

In this section, we will evaluate the load balancing ability of the heat-based algorithm. Please note that there are actually two levels of load balancing in real parallel storage systems. The first level is memory caching, i.e. the main memory could cache the popular disks. The second level is buffer disk caching. In order to study the effects of load balancing in the buffer disks, we suppose no data are cached in the memory.

Recall that the temperature of a buffer disk clearly shows how busy it is. Figure 6.8 records the temperature of three buffer disks when we run the simulation for 1000 requests in BUD. From Figure 6.8, we can see that the three temperature curves merge together most of the time. This means that the three buffer disks are almost equally loaded most of the simulation time. In order to identify the information hidden in Figure 6.8, how the dynamic load balancing works, we plot the initial stage, intermediate stage, and final stage of the temperature tracking trace in Figure 6.9, Figure 6.10 and Figure 6.11. At the initial stage, the three buffer disks are not load balanced. Buffer disk 2 is the busiest disk and buffer disk 1 is lightly loaded. Therefore, the heat-based algorithm will keep allocating requests to buffer disk 1. We can see that the temperature of buffer disk 1 keeps growing and it catches buffer disk 3 first. After that, the temperatures of buffer disk 1 and 3 cross-rise for a while and then they catch buffer disk 2. At this point, the system is load balanced for the first time. Figure 6.10 shows that the whole system is perfectly load balanced in the intermediate stage because the temperatures of three buffer disks rise in turns. Interestingly, we find in Figure 6.11 that the three temperature curves are not as closely intertwined in the final stage when compared to the intermediate stage. This could be explained by the fact that the heat-based load balancing might not be that efficient when all data blocks that are requested are already present in the buffer disks. In other words, if a data miss operation does not occur, there is no chance for the heat-based algorithm to execute. Therefore, the temperature of buffer disks will be largely decided by the access pattern of coming requests.

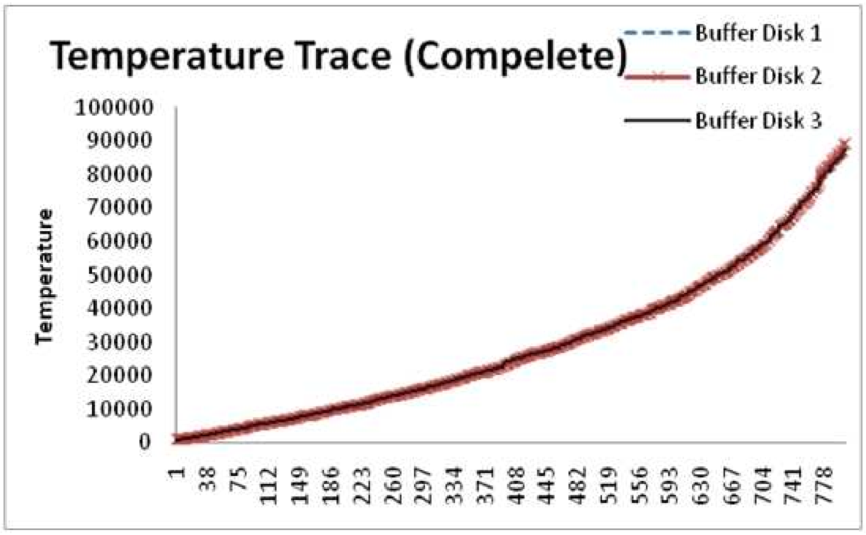


Figure 6.8 Temperature tracking trace

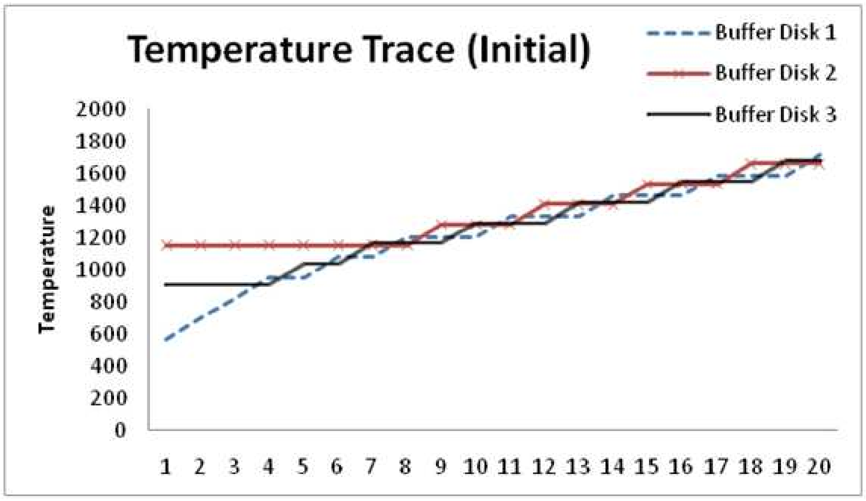


Figure 6.9 Temperatures in initial stage

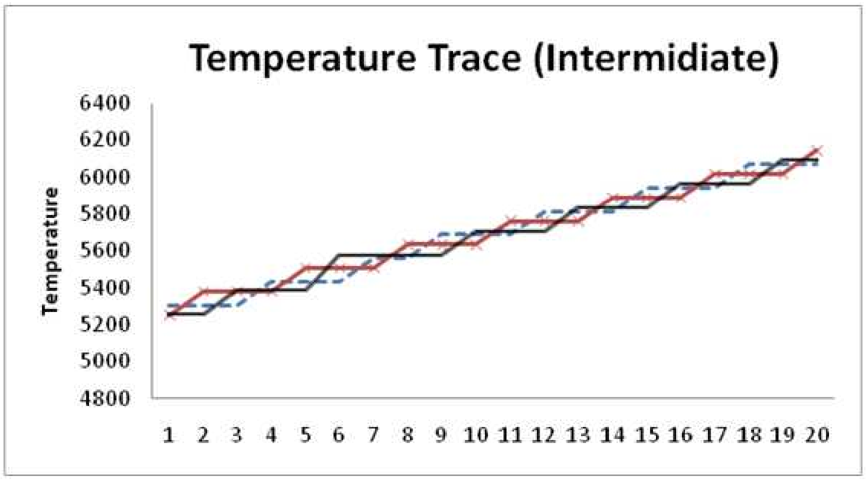


Figure 6.10 Temperatures in intermediate stage

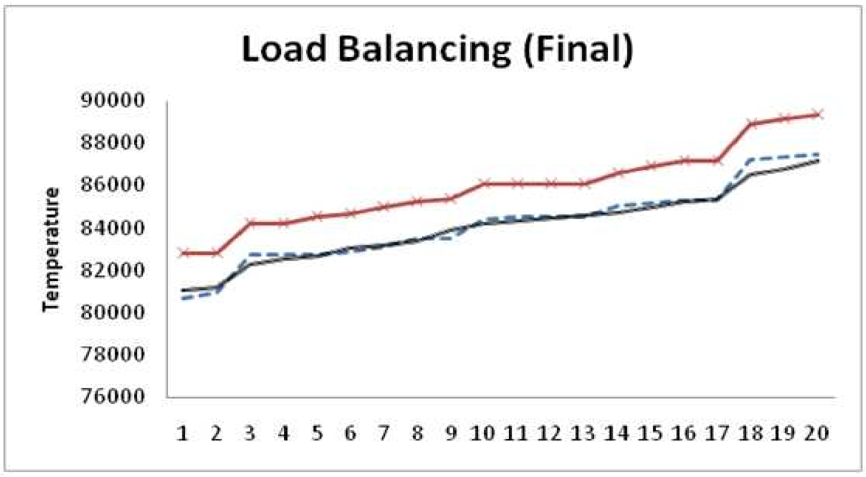


Figure 6.11 Temperatures in final stage

To compare the load balancing efficiency of sequential mapping, round robin mapping, and heat-based mapping, we simulated 5000 requests with average data size of 4MB using these three mapping strategies. The simulation results depicted in Figure 6.12 prove that the proposed heat-based mapping is the most efficient algorithm that achieves load balancing. In addition, the random mapping method (round robin mapping) outperforms non-random mapping strategies (sequential mapping) overall.

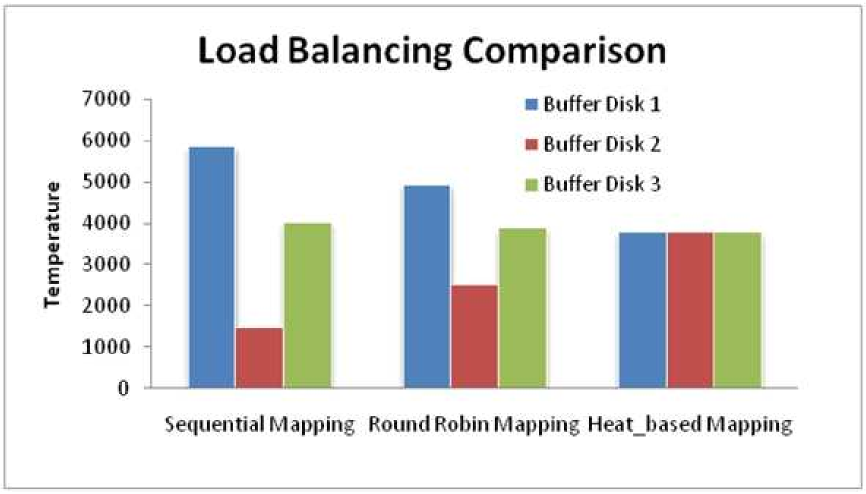


Figure 6.12 Load balancing comparison

6.5.3 Evaluation of Response Time

Response time is one of the most important criteria to evaluate the BUD architecture. This is because the buffer disk architecture leads to response time penalties. This is especially true in the early stages of a workload when few data blocks are cached in buffer disks. However, we believe that the performance penalty in the early stage is worthwhile as long as the system can provide quick response times when the initial caching stage is over.

In order to accurately evaluate the response time, we simulated 25000 requests for large reads (average data size 64MB) and small reads (average data size 64KB), which are illustrated in Figure 6.13-6.16 respectively. For each simulation, we first execute 20000 requests to complete the caching stage. After that, we execute 5000 more requests to see whether or not the system can leverage the response time delay. Since the number of sample requests is too large, it is difficult to analyze the performance trend. Therefore, we plot the trend line in each figure (the black line inside) to better analyze the changing response time trend. The trend line is plotted by calculating the average response time of every 100 tasks and inserting this value into the trend line. For example, if we have 5000 requests, the program will calculate 50 average response times which will be the data points in trend line.

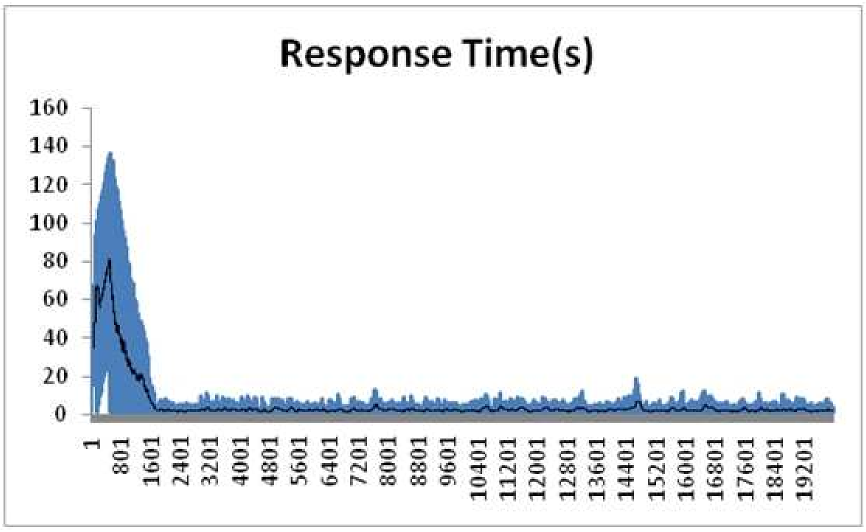


Figure 6.13 Response time trace before training (64MB)

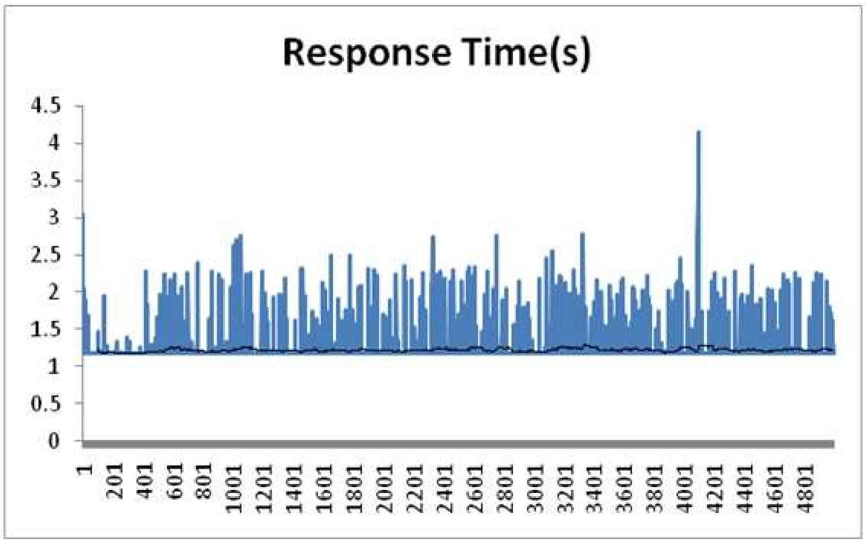


Figure 6.14 Response time trace after training (64MB)

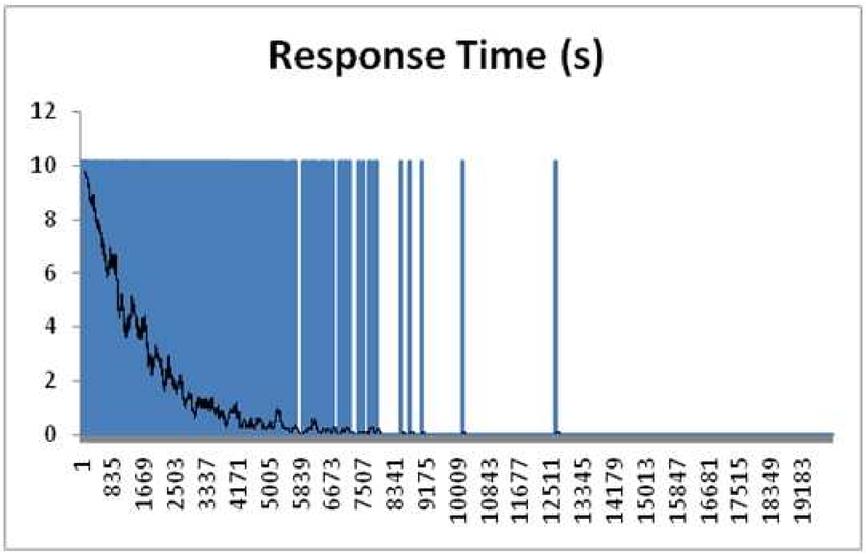


Figure 6.15 Response time trace before training (64KB)

Figure 6.13 and Figure 6.15 verify our prediction of the response time delay in the early caching stage. For example, we can see in Figure 6.13 that the response time delay rises up to 140s. However, we are very delighted to witness the performance improve when more and more hot data blocks are cached in the buffer disks. After the training process, the average response time is very close to the performance of a greedy No-Buffer parallel storage system. For example, the average response time of BUD shown in Table 6.4 is 1.219s for large reads and 0.01s for small reads. These numbers are in the same level of No-Buffer parallel disk systems. We can even predict that the BUD could offer better performance than No-Buffer strategies if higher performance disks serve as the buffer disk in the future.

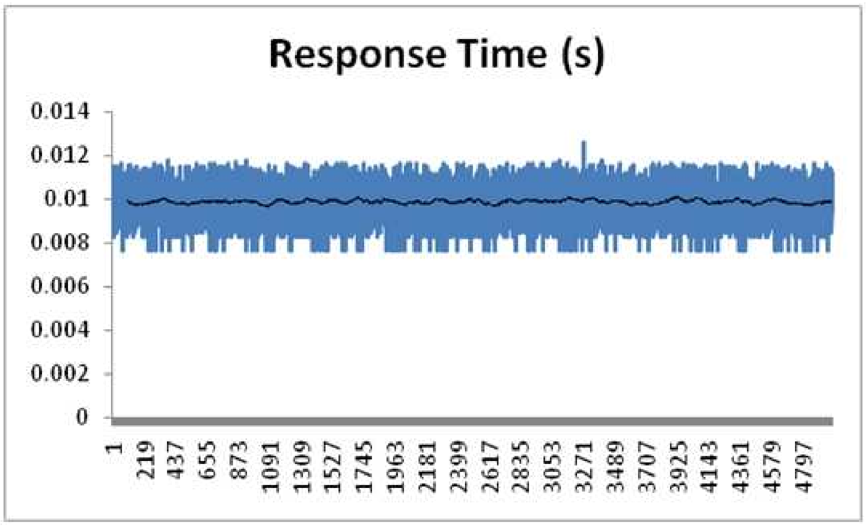


Figure 6.16 Response time trace after training (64KB)

Table 6.4 Average response time comparison

	Average Response Time
training (64MB):	5.614s
after training (64MB):	1.219s
training (64KB):	0.767s
after training (64KB):	0.01s
NO-Buffer(64MB)	1.216s
NO-Buffer(64KB)	0.01s

6.6 Summary

In this chapter, we propose a buffer disk based architecture for parallel storage systems, or BUD for short, which can conserve energy by allowing as many data disks as possible running in low-power mode. A heat-based dynamic data-caching strategy was proposed to improve the performance of BUD architecture by achieving good load balancing in buffer disk layer. We also analyze and compare the impact of three mapping methods, which are sequential mapping, round robin mapping, and heat-based mapping respectively. These mappings are applied to the BUD architecture to gauge load balance, energy consumption, and performance.

The preliminary results have shown substantial gains that BUD can conserve more than 80% of energy when compared with traditional parallel systems that do not employ buffer disks. In addition, the average response time could be as good as the No-Buffer parallel systems. For the future research work we would like to explore the impact of the number of buffer disks and data disks to the system. In addition, we need to incorporate traces from real-world applications to improve the feasibility of our approaches.

Chapter 7

Conclusions and Future Work

In this dissertation, we propose a general architecture for building energy-efficient high-performance computing platforms and discuss the possibility of incorporating energy-efficient techniques in each layer of the proposed architecture. In addition, we have developed a series of energy-efficient algorithms for high-performance computing platforms like clusters, grids and large-scale storage systems. This chapter concludes the dissertation by summarizing the contributions and describing future directions. The chapter is organized as follows: section 7.1 highlights the main contributions of the dissertation. In section 7.2, we concentrate on some future directions, which are extensions of our past and current research on green computing for high-performance computing platforms.

7.1 Main Contributions

Currently, more and more data centers face the energy crisis. This crisis appears to be a mismatch between requirements and capabilities. On the requirements side, to meet application demands and the regulatory requirements, we need to deploy more and more servers. During the years 2000-2010, the number of servers is expected to grow by 6 times and the number of storage disks is expected to grow by 69 times. Accordingly, demands for energy use will significantly increase. How to get enough power to support future data center has become a serious problem. The objective of our research is to find possible and potential energy-efficient techniques to reduce power consumption of high-performance computing platforms built in giant data centers. The main contributions are summarized as follows:

- **Architecture for High-Performance Computing Platforms**

As far as we have known, there is no existing general architecture which is suitable for most high-performance computing platforms. Especially, there is no previous research have discussed the energy conservation issue of high-performance computing platforms in the architecture level. We propose a general architecture for high-performance computing platforms and discuss the possibility of incorporating energy-efficient techniques to each layer of the proposed architecture (See Figure 3.1).

- **Energy-Efficient Scheduling for Clusters**

In the past few years, high-performance clusters have been widely used to solve challenging and rigorous engineering tasks in industry and scientific applications. Due to extremely high energy cost, reducing energy consumption has become a major concern in designing economical and environmentally friendly Clusters for many applications. We propose two energy-efficient duplication-based scheduling algorithms called EAD and PEBD for clusters. They aimed to reduce energy consumption in clusters while minimizing communication overheads associated with parallel tasks. Rather than just consider energy or performance, our algorithms strived to balance the scheduling lengths and energy savings by judiciously replicating predecessors of a task if the duplication can aid in performance with limited energy consumption. We conducted extensive experiments using both synthetic benchmarks and real-world applications to prove the efficiency of these two algorithms.

- **Energy-Efficient Scheduling for Grids**

Grids are complicated heterogeneous super computing platforms which can simultaneously execute thousands of parallel tasks. How to energy-efficiently schedule those parallel tasks in complex heterogeneous grids environment is an open problem. The objective of this study is to develop energy-efficient data grids to provide significant energy savings for data-intensive applications running on grids. We designed a generic energy-aware scheduling framework for grids and proposed two energy-efficient algorithms called EETDS and HEADUS. In addition, we evaluated the performance and energy efficiency of the proposed algorithms by conducting extensive simulations.

- **Energy-Efficient Storage Systems**

With the tremendous development of human society, billions of data in the form of knowledge and information is generated every day. In order to save and process these massive data sets with high-performance, a large number of disks have to be operated in parallel, which introduces a serious problem: huge energy consumption. To build energy-efficient storage systems, we propose a buffer-disk based architecture. In addition, we design and implement corresponding energy-aware load balancing strategy for the buffer-disk architecture.

7.2 Future Work

In the course of designing and evaluating energy-aware resource management techniques for high-performance computing platforms, we have found several interesting issues that are still unresolved. This section overviews some of these open issues that need further investigation. These open issues present opportunities for my future research.

- **Energy-Efficient Scheduling for Embedded Systems**

Embedded/mobile devices are even more sensitive to power consumption due to the limited battery life. I will extend my previous energy-aware research to embedded devices/sensor networks and evaluate previous algorithms in terms of energy efficiency in a more power sensitive environment.

- **Energy-Aware Load Balancing**

The nature of load balancing is to equally spread work between many computers, processes, hard disks or other resources in order to get optimal resource utilization and decrease computing time. In order to do this, the controller or scheduler has to keep as many resources active as possible. This will lead to a potential problem - huge energy consumption. Now we are in a dilemma: increase throughput means more energy consumption while saving energy means system performance degradation. It is expected to propose a power-aware load balancing schema which aims at judiciously spreading work in an energy-efficient way.

- **Optimize Data Movement**

I/O-intensive applications tend to have a huge amount of transferred data. Since the transferred data may be moved from node to node, data movement has a significant impact on the overall performance of load balancing polices. To alleviate such a burden resulting from data movements, it is necessary to propose a predictive model to move data without compromising the performance of applications running on local nodes. The new model should largely depend on data distribution, the amount of data, data access pattern, and network traffic.

- **Dynamic Scheduling Strategies in Grids**

The performance of a large scale heterogeneous grid system is very sensitive to various unforeseen and unplanned events that can happen at short notice, which include but not limited to breakdowns of computers and random arrivals of new jobs. These real-time events not only interrupt system operations, but also have negative impacts on job schedules made on the fly. Therefore, it is highly desirable to develop adaptive dynamic scheduling strategies which can handle those unpredicted events. Multi-agent techniques are promising approaches to building complex, robust, and cost-effective schedulers for the next-generation grid systems, because multi-agents are autonomous, distributed and dynamic in nature. The agent-based dynamic scheduling strategy could be a possible solution to generate robust schedules in a complicated and dynamic distributed computing environment like grids.

- **Service Level Agreement Research in High-Performance Clusters**

It is desirable to develop high-performance clusters to provide secure and reliable services for various types of customer requests submitted to the systems. Various cluster computing use cases have different requirements such as execution deadline, higher security, higher reliability, low cost etc. Therefore, it is highly imperative to develop widely accepted regulations at the high level.

References

[1] B. Moore. Taking the data centre power and cooling challenge. Energy User News, August 27th, 2002.

[2] J. Chase and Ron Doyle, "Energy Management for Server Clusters", *Proc. the 8th Workshop Hot Topics in Operating Systems (HotOS-VIII)*, pp. 165, May 2001.

[3] R. Bryce. Power struggle. Interactive Week, December 2000. http://www.zdnet.com/intweek/, found under stories/news/0,4164,2666038,00.html.

[4] http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf.

[5] S. Srinivasan and N.K. Jha, "Safety and Reliability Driven Task Allocation in Distributed Systems", *IEEE Trans. Parallel and Distributed Systems,* Vol. 10, No. 3, pp. 238-251, March 1999.

[6] L. Benini and G. De Micheli, "Dynamic Power Management: Design Techniques and CAD Tools", *Kluwer Academic Publisher*, 1998.

[7] J. Rabaey and M. Pedram (Editors), "Lower Power Design Methodologies", *Kluwer Academic Publisher*, Norwell, MA, 1998.

[8] A. Raghunathan, N. K. Jha, and S. Dey, "High-Level Power Analysis and Optimization, Kluwer Academic Publisher", Norwell, MA, 1998.

[9] E.N. M. Elnozahy, M. Kistler, and R. Rajamony, "Energy-Efficient Server

Clusters," *Proc. Int'l Workshop Power-Aware Computer Systems*, Feb. 2002.

[10] L. Benini, A. Bogliolo, G. D. Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management," *IEEE Trans. VLSI Sys.*, vol. 8, no. 3, pp.299-316, June 2000.

[11] F. Douglis, P. Krishnan, B. Bershad, "Adaptive Disk Spin-down Policies for Mobile Computers", *USENIX Symp. Mobile and Location-Independent Computing*, pp. 121-137, 1995.

[12] M. Srivastava, A. Chandrakasan. R. Brodersen, "Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation", *IEEE Trans. VLSI Systems*, Vol. 4, No. 1, pp. 42-55, March 1996.

[13] X. Qin and H. Jiang, "A Dynamic and Reliability-driven Scheduling Algorithm for Parallel Real-time Jobs on Heterogeneous Clusters", *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 885-900, Aug. 2005.

[14] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy", *Proc. IEEE Annual Foundations of Computer Science*, pp. 374-382, 1995.

[15] I. Hong, M. Potkonjak, and M. Srivastava, "On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor", *Proc. Computer Aided Design,* pp. 653-656, 1998.

[16] T. Xie, X. Qin, and M. Nijim, "Solving Energy-Latency Dilemma: Task Allocation for Parallel Applications in Heterogeneous Embedded Systems", *Proc. 35th Int'l Conf. Parallel Processing*, Columbus, Ohio, Aug. 2006.

[17] W. Dally, P. Carvey, and L. Dennison, "The Avici Terabit Switch/Rounter", *Proc.*

Hot Interconnects 6, pp. 41-50, Aug. 1998.

[18] .N. M. Elnozahy, M. Kistler, and R. Rajamony, “Energy-Efficient Server Clusters”, *Proc. Int’l Workshop Power-Aware Computer Systems*, Feb. 2002.

[19] Mellanox Technologies Inc., “Mellanox Performance, Price, Power, Volumn Metric (PPPV)”, http://www.mellanox.co/products/shared/PPPV.pdf, 2004.

[20] T. Hagras and J. Janecek, “A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems”, *Proc. IEEE Heterogeneous Computing Workshop*, 2006.

[21] I. Page, T. Jacob, and E.Chen, “Fast Algorithms for Distributed Resource Allocation”, *IEEE Trans. Parallel and Distributed Sys.*, vol. 4, no. 2, pp. 188-197, Feb. 1993.

[22] G.C. Sih and E.A. Lee, “Declustering: A New Multiprocessor Scheduling Technique”, *IEEE Trans. Parallel and Distributed Sys.*, vol. 4, no.6, pp. 625-637, June 1993.

[23] G.C. Sih and E.A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures”, *IEEE Trans. Parallel and Distributed Sys.*, vol. 4, no. 2, pp. 175-186, Feb. 1993.

[24] C.M. Woodside and G.G. Monforton, “Fast Allocation of Processes in Distributed and Parallel systems”, *IEEE Trans. Parallel and Distributed Sys.*, vol. 4, no. 2, pp. 164-174, Feb. 1993.

[25] N.S. Bowen, C.N. Nikolaou, and A. Ghafoor, “On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems”, *IEEE Trans. Computers*, vol. 41, no. 3, Mar. 1992.

[26] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", *IEEE Trans. Computer*s, pp. 50-60, June 1982.

[27] V.M. Lo, "Heuristic Algorithms for Task Assignments in Distributed System", *IEEE Trans. Computers*, vol. 37, no. 11, pp. 1,384-1,397, Nov. 1988.

[28] S. Yajnik, S. Srinivasan, and N.K. Jha, "TBFT: A Task-Based Fault Tolerance Scheme for Distributed Systems", *Proc. Int'l Conf. Parallel and Distributed Computer Sys.*, Oct. 1994.

[29] Y. Shin and K. Choi, "Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems", *Proc. Design Automation Conf.*, 1999.

[30] S. Bansal, P. Kumar, and K. Singh, "An Improved Duplication Strategy for Schedulng Precedence Constrained Graphs in Multiprocessor Systems", *IEEE Trans. Parallel and Distributed Systems*, Vol. 14, No. 6, pp. 533-544, June 2003.

[31] S.S. Pande, D.P. Agrawal and J. Mauney, "A Scalable Scheduling Method for Functional Parallelism on Distributed Memory Multiprocessors", *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 4, pp. 388-399, April 1995.

[32] S. Darbha and D. P. Agrawal, "A Task Duplication Based Scalable Scheduling Algorithm for Distributed Memory Systems", *J. Parallel and Distributed Computing*, vol. 46, no. 1, pp. 15-27, Oct. 1997.

[33] S. Ranaweera, and D.P. Agrawal, "A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems", *Proc. Parallel and Distributed Processing Symp.*, pp.445-450, May 2000.

[34] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems", *Proc. IEEE*

Heterogeneous Computing Workshop, 2006.

[35] H.J. Siegel *et al.*, "Mapping subtasks with multiple versions on an ad-hoc grid", *Proc. IEEE Heterogeneous Computing Workshop*, 2006.

[36] Y. Kishimoto and S. Ichikawa, "Optimizing the configuration of a heterogeneous cluster with multiprocessing and execution-time estimation", *Proc. IEEE Heterogeneous Computing Workshop*, 2006.

[37] J. Cuenca, D. Gimenez and J.-P. Martinez, "Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems", *Proc. IEEE Heterogeneous Computing Workshop*, 2006.

[38] StorageTek Corp. 9310 tape silo information, http://www.storagetek.com/products/tape/9310/2001.

[39] D. Colarelli and D. Grunwald, "Massive Arrays of Idle Disks for Storage Archives", *Proc. of the 15th High Performance Networking and Computing Conf.*, November 2002.

[40] "Power, heat, and sledgehammer", White paper, Maximum Institution Inc., http://www.max-t.com/ownloads/whitepapers/SledgehammerPowerHeat20411.pdf.

[41] Fred Moore, "More Power Needed", Energy User News, November 2002.

[42] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the Power-Hunger Disk", *Proc. Winter USENIX Conf.*, pp.292-306, 1994.

[43] Q. Zhu, F. M. David, C. F. Devaaraj, Z. Li, Y. Zhou, and P. Cao, "Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management," *Proc. High-Performance Computer Framework*, 2004.

[44] S.W. Son and M. Kandemir, "Energy-aware data prefetching for multi-speed

disks", *Proc. ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2006.

[45] S.W. Son, M. Kandemir, and A. Choudhary, "Software-directed disk power management for scientific applications", *Proc. Int'l Symp. Parallel and Distributed Processing*, April, 2005.

[46] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Fanke, "DRPM: Dynamic Speed Control for Power Management in Server Class Disks", *Proc. Int'l Symp. of Computer Framework*, pp. 169-179, June 2003.

[47] E. Pinheiro and R. Bianchini, "Energy Conservation Techniques for Disk Array-Based Servers", *Proc. of the 18th International Conference on Supercomputing (ICS'04)*, June 2004.

[48] P. Scheuermann, G. Weikum, P. Zabback, "Data partitioning and load balancing in parallel disk systems", *The International Journal on Very Large Data Bases*, vol. 7, issue 1, pp. 48-66, Feb.1998

[49] S. Darbha, D.P. Agrawal, "Optimal Scheduling Algorithm for Distributed-Memory Machines", *IEEE Trans. Parallel and Distributed Systems*, Vol. 9, No. 1, pp.87-95, Jan. 1998.

[50] R.L. Graham, L.E. Lawler, J.K. Lenstra, and A.H. Kan, "Optimizing and Approximation in Deterministic Sequencing and Scheduling: A Survey", *Annals of Discrete Math*, pp.287-326, 1979.

[51] H.El. Rewini, T.G. Lewis, and H.H. Ali, "Task Scheduling in Parallel and Distributed Systems", New Jersy: Prentice Hall, 1994.

[52] M.Y. Wu and D.D. Gajski, "Hypertool: A Performance Aid for Message-Passing

Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 330-343, July 1990.

[53] Standard Task Graph Set web site. http://www.kasahara.elec.waseda.ac.jp/schedule/ making_e.html#application.

[54] H. Kasahara and S. Narita, "Parallel Processing of Robot-Arm Control Computation on a Multiprocessor System", *IEEE J. Robotics and Automation*, Vol.RA-1, No.2, pp. 104-113 (1985).

[55] H. Kasahara, H. Honda and S. Narita, "Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR", *Proc. IEEE ACM Supercomputing '90* (1990).

[56] A. Yoshida, K. Koshizuka and H. Kasahara, "Data-Localization for Fortran Macrodataflow Computation Using Partial Static Task Assignment", *Proc. 10th ACM Int'l Conf. on Supercomputing*, pp. 61-68 (1996).

[57] Wikipedia about Grid computing, http://en.wikipedia.org/wiki/Grid_computing.

[58] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A Security Architecture for Computational Grids", *Proc. ACM Computer and Communication Security*, San Francisco, CA, USA, 1998.

[59] T.T. Kwan, R.E. McGrath, and D.A Reed, "NCSA's World Wide Web Server: Design and Performance", *IEEE Computer*, vol. 28, no. 11, pp. 68 – 74, Nov. 1995.

[60] D. Li and J. Wang, "EERAID: Energy-Efficient Redundant and Inexpensive Disk Array", *Proc. of the 11th ACM SIGOPS European Workshop*, Sept 2004.

[61] D. Li and J. Wang, "Conserving Energy in RAID Systems with Conventional

Disks", *Proc. of the International Workshop on Storage Network Architecture and Parallel I/Os*, Sept 2005.

[62] X. Yao and J. Wang, "RIMAC: A Redundancy-based Hierarchical I/O Architecture for Energy-Efficient Storage Systems", *Proc. of the 1st ACM EuroSys Conference*, Apr 2006.

[63] C. Ruemmler, J. Wilkes, "An introduction to Disk Drive Modeling", *IEEE Trans. on Computers*, Vol. 27, no. 3, pp. 17 – 28, 1994.

[64] E.V. Carrera, E. Pinheiro, and R. Bianchini, "Conserving Disk Energy in Network Servers", *Proc. of the 17th International Conference on Supercomputing (ICS'03),* June 2003.