REQUIREMENT-AWARE STRATEGIES FOR SCHEDULING MULTIPLE DIVISIBLE LOADS IN CLUSTER ENVIRONMENTS

HU MENGLAN

NATIONAL UNIVERSITY OF SINGAPORE

2012

REQUIREMENT-AWARE STRATEGIES FOR SCHEDULING MULTIPLE DIVISIBLE LOADS IN CLUSTER ENVIRONMENTS

HU MENGLAN

(*B. Eng, HUST*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2012

Acknowledgments

I would like to thank my advisor, Bharadwaj Veeravalli, for supporting me over the years, and for giving me so much freedom to explore and discover new areas of task scheduling in distributed systems. His dedication, encouragement, support, and hard work constantly inspired me to better myself and aim higher.

To my parents: Thank you for letting me pursue my dream for so long so far away from home. I also thank my grandparents for giving me unconditional love and support.

I want to thank my friends and my colleagues in CNDS lab for their kind assistance and suggestions on research and other issues. These include Wang Yang, Zeng Lingfang, Luo Tie, Lau Kah Soon, and other members in CNDS lab. The interesting discussions during lunch and supper time are so enjoyable. Finally, I would like to thank all my friends who made my life in Singapore enjoyable and sociable. Thank you for all the good times.

Table of Contents

Summary

Divisible load applications occur in many fields of science and engineering. Such applications can be easily parallelized in a master-worker fashion, but pose several scheduling challenges. This thesis investigates the problem of scheduling multiple divisible loads in cluster systems with a particular emphasis in capturing two important real-life constraints, various processing requirements of different loads, and different load types.

We first study the problem of scheduling multiple divisible loads with different processing requirements. Since the divisible loads to be performed may widely vary in terms of their required hardware and software, we capture the loads' various processing requirements in our load distribution strategies, a unique feature that is applicable for running proprietary applications only on certain eligible processing nodes. Thus in our formulation each task may only be processed by some certain nodes due to their different processing requirements.

We also study the scheduling of hybrid tasks comprising both divisible and indivisible loads. Indivisible loads are characterized by the property that they need to be processed on their entirety on a single processor while divisible loads can be distributed across several processing nodes by exploiting the underlying data parallelism. Since clusters are designed to handle any types of loads, handling

hybrid tasks comprising both divisible and indivisible loads is common in practice.

We thoroughly investigate the above problems for both real-time and non-real-time tasks. We contribute several efficient scheduling algorithms that are aware of different processing requirements and load types of the tasks. Also, we perform extensive performance evaluations to demonstrate the effectiveness and competitiveness of our algorithms on various scenarios.

List of Figures

Chapter 1

Introduction

1.1 Background of Divisible Load Theory

Distributed computing is an area in computer science that utilizes distributed systems to solve computational problems. A distributed system consists of multiple autonomous computers that communicate through a computer network. These networked computers interact with each other in order to achieve a common goal. In distributed computing, a computational problem is divided into many small tasks, which are then solved in parallel by the networked computers. Due to the great computing power of the distributed systems, they are widely used to solve large-scale scientific and engineering problems, which require massive amounts of computations. Over the last decade, various distributed systems such as high performance computer clusters and grids have been deployed to process enormous applications (or tasks) in various scientific and engineering domains such as mathematics, computational physics, computational chemistry, climate change, weather prediction, seismic analysis, bioinformatics, drug discovery, economic forecasting and financial modeling.

Scheduling tasks in distributed systems is a critical issue which demands a clever design of polices to partition, assign, communicate, and process the load on networked computers. Inefficient scheduling decisions may result in severe overheads and poor performance when running applications. The task scheduling problem includes many forms, depending on the properties of the tasks to be scheduled, the computing platforms, and the goal of scheduling. One type of scheduling problems is that no dependencies exist among the tasks, and the number of tasks and task sizes can be arbitrarily chosen. In the real world, this is the case for various large-scale applications in scientific and engineering domains, such as image processing, database searching, matrix computing, and protein/DNA sequencing. Such applications are structured as large numbers of independent, identical, and low granularity computations and are thus amenable to straightforward parallel computing. These applications have been called divisible loads because a scheduler may arbitrarily divide the loads among networked computers, in terms of both the numbers and sizes of the tasks.

The above scheduling problem can be characterized using the divisible load model, which has been studied extensively in the last two decades, resulting in a cohesive theory called Divisible Load Theory (DLT) [1]. DLT offers a tractable and powerful tool to scheduling that allows linear and continuous modeling of partitionable computations and communications for parallel processing. A vast number of literatures on DLT demonstrates it as an elegant methodology for handling large-scale applications in distributed systems [2, 3, 4, 5, 6]. In practice, this divisible load model is an approximation of an application consisting of massive identical, independent, and low-granularity computations, and has been applied to a wide spectrum of real-life applications including database searching [7], matrix computing [8], and biological sequencing [9].

1.2 Motivations

Prior DLT literatures usually explore the scheduling of a single divisible load. However, in practice, distributed systems are not likely to be utilized exclusively by a single application. Thus it is highly desirable to investigate the problem of scheduling multiple divisible loads. But this problem receives only limited attention due to its complexity. This problem was first considered in [1], which assumed that tasks were executed in the first-in-first-out order on a set of heterogeneous processors, all processors were used by each task, the computations of a task finished on all processors simultaneously, and startup time was negligible. Based on the same set of assumptions, a multi-installment load distribution strategy was proposed in [10]. This strategy is advantageous as it can minimize schedule length, but for some cases it cannot avoid idle times in utilizing processors. Marchal et al. [11] formulated a steady-state multi-task divisible load scheduling problem based on a novel and realistic network model for wide-area networks. They found this problem as NP-complete and hence proposed several heuristics for seeking near-optimal solutions. Drozdowski et al. [12] discussed the computational complexity of scheduling multiple divisible loads on a star network. They showed that the problem is computationally hard (strictly speaking NP-hard) and identified several special cases solvable in polynomial time. Drozdowski and Lawenda [13] analyzed the problem of scheduling multiple divisible loads on homogeneous star systems and showed that this problem is again computationally hard. They also presented polynomial time solutions for special cases.

In the above works the diversity of different divisible loads were not taken into consideration. They assume that all tasks are of the same type and have the same processing requirements. However, in real world, the tasks can be of different types and thus have different processing requirements on computing resources. Accordingly, we identify two open problems in scheduling multiple divisible loads.

Firstly, the above papers assume that all tasks have the same processing requirements so that all processing nodes can compute all tasks. However, for real life applications, this assumption does not always hold true. In practice, due to variations in different tasks and processing nodes not all nodes may be capable of computing all tasks. There exist some proprietary applications that are licensed and are available only at specific vantage locations. Also, compute nodes' software and hardware may become bottlenecks and restrict their capabilities of processing various loads. For example, an application requiring 2 Gigabytes RAM should be assigned to compute nodes that have at least that much RAM. Therefore it is highly desirable to investigate the problem of scheduling multiple divisible loads with various processing requirements and design load distribution strategies that inherently absorb those requirements. We refer to this context simply as requirement-aware scheduling problem.

Secondly, the prior works assume that all tasks are divisible loads. However, in practice divisible and indivisible tasks can coexist in the system for processing. Since clusters are in general catered to process any types of loads, handling a set of tasks comprising both divisible and indivisible loads is very common in real-life. But to the best of our knowledge, none of the earlier works addressed the problem of scheduling hybrid divisible and indivisible tasks on distributed system. To this end, in the thesis we also investigate the problem of scheduling a set of tasks comprising both divisible and indivisible tasks on clusters.

By and large, the first problem addresses which processors can be used to compute a given task while the second problem concerns how many processors are to be used. Therefore both issues deal with tasks' various processing requirements on computing resources from two different perspectives.

1.3 Objectives and Contributions of the Thesis

The above discussion clearly sets the motivation for the problems addressed in this thesis. Thus the general objective of this thesis is to investigate the problem of scheduling multiple divisible loads on cluster systems and design efficient scheduling algorithms that are aware of different processing requirements and load types of the tasks. While achieving this goal, this thesis also addresses two realistic scenarios.

The first scenario is scheduling divisible loads under arbitrary processor release times. Most prior DLT works assume that all processors are simultaneously available for a task. However, in real-life application execution some processors may have been allocated to previously-admitted tasks. Such processors will not be available until their previously-admitted tasks are completed. Thus it is worth studying the case that processors have different release times (or ready times). A few prior works have studied the problem of scheduling divisible loads with different processor release times. Prior work [14] presented strategies for scheduling divisible loads on bus networks with arbitrary processor ready times. Closed-form solutions were derived for identical release times and a heuristic was presented for arbitrary release times. Another work [15] studied scheduling divisible loads on linear chain networks with arbitrary processor ready times. In [15] both single and multi-installment strategies were presented for the cases that a minimum processing time can be achieved. Two heuristics were proposed for the cases that a minimum processing time cannot be realized. Moreover, Bharadwaj and Barlas [16] addressed different processor release times together with finite buffer constraints. In these works predetermined and known ready times were assumed. However, in real-life scenarios, these assumptions are not always satisfied. In many cases processors' release times are indeterminable or unknown until they become available. Therefore, it is beneficial to consider the case of dynamic processor ready times, where ready times are unknown until processors are released. Motivated by this need, in Chapter 4 we consider the requirement-aware scheduling problem with arbitrary processor release times. We study both static and dynamic processor release times and present efficient algorithms to tackle these challenges.

The second scenario is scheduling real-time tasks, which is also a critical issue in the field of distributed computing. Several prior algorithms have been proposed for scheduling real-time divisible loads in cluster environments. In [17] Lin et al. proposed a real-time divisible load scheduling algorithm for cluster environments. Their algorithm employs the DLT rule to compute the minimum number of processors required to meet an application's deadline. In [18] Lin et al. considered different processor available times and revisited the works in [17] to develop a scheduling algorithm which exploits inserted idle times to provide real-time guarantees. In [19] Chuprat and Baruah extended the works in [18] and provided exact solutions to improve the approximate solutions in [18]. In [20] Mamat et al. considered advance reservations in designing real-time divisible load scheduling algorithms. These papers adopted a "single-round" scheme in which one can divide the load into many pieces equal to the number of processors and dispatch the pieces in a single round of allocation. Although the single-round scheme is simple to implement, yet it leads to poor overlap of communication and computation [5]. Therefore, in [21], the authors attempted to extend [5] to design a “multi-round” scheme. However, they only considered the simplest case of one task and did not provide algorithms to schedule multiple real-time tasks. To this end, in the thesis we also study the scheduling problems for real-time tasks. Specifically, in Chapter 5 we investigate the problem of scheduling real-time divisible loads with various processing requirements. We propose to design efficient “multi-round” scheduling algorithms which can efficiently overlap communication with computation by dispatching loads in multiple rounds. In addition, in Chapter 7 we study the problem of scheduling hybrid real-time tasks comprising divisible and indivisible tasks.

The scope of the thesis is to design efficient algorithms for scheduling multiple divisible loads with different processing requirements and load types. We also perform rigorous performance evaluation studies to quantify the performance of our strategies on a variety of scenarios. However, implementation is beyond the scope of the thesis.

This thesis thoroughly studies the requirement-aware scheduling problems and contributes several algorithms to solve these problems. Chapter 4 investigates scheduling multiple divisible loads with arbitrary processor release times and various processing requirements. Both static and dynamic processor release times are taken into account and two scheduling strategies are presented, respectively. In addition, a requirement-aware load selection policy is also provided to handle the contention among different loads with various processing requirements. Chapter 5 considers the problem of scheduling real-time divisible loads with various processing requirements. In this chapter we present requirement-aware real-time scheduling algorithms, which consist of a real-time scheduling policy and two load distribution strategies. Chapter 6 studies the scheduling problem for processing a set of tasks comprising both divisible and indivisible tasks. A scheduling algorithm is provided to achieve high resource utilization. Chapter 7 addresses the scheduling problem for processing a set of tasks comprising both divisible and indivisible real-time tasks. We propose a dynamic real-time scheduling algorithm, which is shown to efficiently exploit the parallelism in divisible loads without undermining the schedulability of indivisible loads and thereby optimize the overall performance.

Nowadays Cloud/Cluster/Grid systems have been widely deployed to handle various applications. In such environments multiple applications share and contend for the resources. But such studies are currently missing in the DLT domain and hence become the primary focus of this thesis. Therefore the problems addressed in this thesis are very useful and directly applicable. To the best of our knowledge, the contributions in this study are the first of its kind to schedule multiple divisible loads with various processing requirements and load types.

The organization of this thesis is as follows. Chapter 2 presents a detailed review of related works in the fields of scheduling divisible loads, indivisible loads and real-time tasks, respectively. Chapter 3 introduces mathematical models and assumptions used throughout the thesis. In Chapters 4 and 5 we investigate the problem of scheduling multiple divisible loads with different processing requirements for non-real-time and real-time tasks, respectively. Chapter 4 also considers the challenge of arbitrary processor release times. In Chapters 6 and 7 we study the problem of scheduling a set of hybrid tasks comprising divisible and indivisible loads for non-real-time and real-time tasks, respectively. Finally, in Chapter 8, we conclude this thesis and put forward some future recommendations in the context of this problem.

Chapter 2

Related Works

2.1 Divisible Load Scheduling

The research of scheduling divisible loads in distributed systems originated in 1988, with two initial works [22] and [23]. A later work [24] first presented a formal mathematical framework for scheduling divisible loads and denominated the theory as Divisible Load Theory (DLT). DLT offers a tractable and powerful tool to scheduling that allows linear and continuous modeling of partitionable computations and communications for parallel processing. DLT literature until 1996 was summarized by a landmark book [1] including above mentioned theoretical framework and formulations. The book popularizes DLT and paves the way for various extended DLT works in the last decade. In addition, two recent survey articles [25, 26] highlighted the advantages of using DLT. Moreover, a special issue of the Cluster Computing journal was devoted to divisible load scheduling [27]. Below we review related DLT literature in detail.

Network topology is a critical issue to consider when designing scheduling algorithms. Previous DLT works have studied many network topologies, such as bus, linear chain, tree, mesh, and arbitrary graph. Bus is one common network topology of local area networks and has received much attention in DLT studies. In bus networks, processors are connected through a bus and the communication speed between any two processors is identical. Also, any two processors can directly communicate with each other. The authors in [28] provided a closed form solution of optimal load distribution in bus networks. A later work [29] demonstrated the optimality principle for bus networks. Linear daisy chain is another basic network topology, in which processors are sequentially linked one by one. Any intermediate processor in the chain receives its load fraction from its predecessor and relays the rest of the load to next processor. Robertazzi in [30] presented a concept of "equivalent processor" and used it to determine load distribution in linear chain networks. Mani and Ghose in [31] provided a closed-form solution of optimal load distribution in linear networks. A more complicated topology is a tree network. The initial work which studied tree networks was [32], in which both "with front-end" and "without front-end" cases were studied. However, this paper only provided recursive equations but not rigorous closed-form solution. The closed-form solution for tree networks was derived in [33]. Beaumont et al. in [3] investigated both one-round and multi-round algorithms for both star and tree network topologies, in which both the linear and affine cost models were considered. A few open problems were also discussed in this work. In [34] the cost of result collection was taken into consideration and approximate algorithms were proposed for arbitrary processor trees. Another complicated topology which has received much attention in DLT literature is the mesh topology. The first work for the mesh topology was reported in [35], in which a circuit-switched routing algorithm was presented for a two-dimensional mesh network. This paper also provided a scattering approach and analyzed the performance bound with communication delays. A later work [36] studied a two-dimensional toroidal mesh and proposed a Peters-Syska scattering algorithm which was shown to outperform the method in [35]. Another work [37] derived a closed-form solution for load distribution in three-dimensional mesh networks. Furthermore, Li [38] presented load distribution algorithms for linear arrays that can be generalized to k-dimensional meshes. In this work a pipelined communication technique was used and a closed-form solution was derived. A more recent work [39] proposed improved methods using a multi-round technique for k-dimensional meshes and these methods has exhibited superior performance than the algorithms provided in [38]. Finally, the arbitrary graph topology has also been addressed. J. Yao et al. [40] proposed a RAOLD-OS strategy for scheduling divisible loads in an arbitrary graph. This algorithm works by first constructing a minimum spanning tree and then scheduling loads on this tree. In another work [41] Darin England et al. designed a robust spanning tree for arbitrary graphs to make a balance between robustness and performance.

Another important issue regarding the communication model is whether multiple simultaneous transmissions are allowed on one processor. Most papers in DLT literature adopted a sequential communication mode, in which multiple simultaneous transmissions are not allowed on one processor. This assumption is simple and useful in LAN since the communication throughput is restricted by the bandwidth of the LAN, not the number of concurrent links. On the other hand, current technologies allow simultaneous transmissions on one processor and the parallel communication mode can be beneficial for computing platforms over wide area networks (WAN) to achieve higher throughput than the sequential communication mode due to bandwidth-sharing properties. Hence, many works

[36, 38, 11] have also considered the parallel communication mode. Marchal et al. [11] proposed a multi-port network model for wide-area networks and proposed heuristics for seeking sub-optimal solutions for a steady-state multi-task divisible load scheduling problem.

Start-up overheads have also received much attention in DLT research. In practice, start-up overheads exist for both communication and computation. For communication, the overheads may due to connection initialization, transmission delays, queuing delays, and so on. For computation, the overheads may include processor initialization, unpacked delays, layered protocol delays, and so on. Although in many cases such overheads are negligible and a linear cost model can be applied to model the communication and computation, yet some works [34, 36, 37, 42, 43, 44] have taken these overheads into account and adopted an affine cost model in which the overheads were treated as constants. The authors in [34] investigated the cost of result collection and presented approximate algorithms for arbitrary trees with start-up overheads. In [36] and [37], circuit-switching-based scheduling strategies were proposed for 2D and 3D mesh networks under the affine cost model. Another work [42] studied start-up overheads for different network topologies including bus, linear chain, tree and hypercube, and proposed recursive equations for different cases. However, this work only addressed communication overheads. A more general work [43] studied overheads in both communication and computation. Closed-form solutions were provided and the effect of the start-up costs was discussed in this paper. A later work [45] has demonstrated that scheduling divisible loads on star networks under the affine cost model is NP-Complete. Moreover, the combined effect of start-up cost and the finite buffer constraint was studied in [44].

The above papers frequently adopted a ”single-round” scheme in which one can divide the load into many pieces equal to the number of processors and dispatch the pieces in a single round of allocation. Although the single-round scheme is simple to implement, yet it leads to poor overlap of communication and computation. To this end, some prior works also proposed ”multi-round” algorithms which can efficiently overlap communication with computation by dispatching loads in multiple rounds. The first multi-round algorithms were proposed in [46, 47], for linear chain and tree networks respectively. The authors in [47] presented a multi-installment algorithm, which begins with small load chunks and gradually increases chunk sizes throughout the load distribution process. In addition, the authors provided a closed-form solution for homogenous systems, given a fixed number of rounds. Later works [48, 49] also proposed multi-round algorithms and all of these works adopted a linear cost model in which start-up overheads were not taken into consideration. Under the same model, [50] proposed a multi-round algorithm which was shown to be asymptotic optimal. However, one cannot use this algorithm to obtain the optimal number of rounds because the asymptotic optimality is achieved when the number of rounds approaches infinity, which is impractical. To derive more practical solutions, [3, 51, 5] adopted an affine cost model to take into account start-up overheads in communication and computation. Beaumont et al. in [3] investigated both one-round and multi-round algorithms for both star and tree network topologies, in which both the linear and affine cost models were considered. A few open problems were also discussed in this work. Yang et al. in [5] proposed a multi-round algorithm called UMR which can derive a near optimal number of rounds under the affine model. To adapt to dynamic environments, [52, 53] extended UMR to handle time-varying computation and communication speeds. Authors in [52] proposed to decrease chunk sizes at the end of load distribution to tolerate performance prediction errors in dynamic environments, while in [53] the authors focused on designing efficient performance prediction schemes. Furthermore, multi-round algorithms are also used to solve multi-task scheduling problems. Marchal et al. in [11] designed a novel network model for wide-area networks and formulated a steady-state multi-task divisible load scheduling problem. They proposed multi-round heuristics for seeking sub-optimal solutions for this problem.

Since DLT provides a practical framework for mapping independent tasks onto heterogeneous platforms, it has been widely applied to a large spectrum of real-life applications including data measurement in wireless sensor networks [54], image processing [55], large-scale matrix computations, [8], discrete wavelet transform computations [56], database searching [7], load balancing on grid platforms [57, 58], multimedia applications [59, 60, 61], biological sequences aligning [9], and parallel video processing [62]. All the above applications demand processing massive computational loads that can be partitioned into small and independent fractions and thus they are amenable to the DLT paradigm.

2.2 Real-Time Scheduling

The scheduling of real-time tasks on clusters is a critical issue for achieving high performance. Many real-time scheduling algorithms have been proposed in literature. Real-time scheduling algorithms generally fall into two categories: static (offline) [63, 64] and dynamic (online) [65, 66, 67]. In static algorithms, the allocation of tasks to processors and the time that the tasks begin execution are determined beforehand. Static algorithms [63] are used to schedule periodic real-time tasks and the benefit is that once a schedule is obtained, deadlines will never be violated. However, static algorithms cannot be used to schedule aperiodic real-time tasks whose arrivals are unexpected. Scheduling such tasks requires dynamic scheduling algorithms which dynamically admit and schedule new tasks without compromising the guarantees for previously admitted tasks. In multiprocessor systems, most real-time scheduling is NP-hard [65, 68, 69]. The authors in [68] have shown that there are no optimal algorithms for scheduling aperiodic real-time tasks whose characteristics are not known a priori in multiprocessor systems. It was shown in [65] that a heuristic concerning a combined function of the deadline and earliest start time of a task outperforms classical heuristics like EDF (Earliest Deadline First) and LLF (Least Laxity First).

In the past decades, most prior works on multiprocessor real-time scheduling assumed that a task is indivisible (independent) and executed on only one processor [65, 66]. This may cause deadlines to be violated due to poor resource utilization. For example, a task will miss the deadline when the required computation time is greater than the deadline. These motivate the research on parallel scheduling of real-time tasks. Several recent algorithms have been proposed to deal with the parallel real-time task scheduling problem [70, 67, 71, 72, 73, 74]. The authors in [70] designed real-time algorithms for scheduling tasks on a partitionable hypercube multiprocessor. The paper [67] presented a dynamic multiprocessor real-time scheduling algorithm which allocates a task to multiple processors when the allocation of one processor cannot guarantee the deadline. The authors in [71] proposed a dynamic scalable task scheduling algorithm, in which the number of processors allocated to each task is kept as small as possible while the deadline is satisfied and a task with a large workload derivative is favored to start earlier. Another recent work [72] addressed fairness issues for scheduling grid applications with deadlines. Further, Netto and Buyya [74] studied the scheduling of parallel bag-of-tasks applications with deadlines.

2.3 Scheduling Coarse-Grain Independent Tasks

In the recent past, the problem of task scheduling in distributed systems has been extensively studied. A number of scheduling algorithms have been proposed in the literature. Since this scheduling problem is NP-complete [75], the majority of proposed solutions are heuristic algorithms. These algorithms can be classified either as knowledge-based or knowledge-free. Knowledge-based algorithms assume that perfect performance prediction information concerning resources and tasks is known at the time of scheduling. Well-known knowledge-based heuristics include Max-Min, Min-Min, Sufferage [76, 77], XSufferage [78], and Storage Affinity (SA) [79]. Max-Min fist schedules the task whose earliest completion time over all processors is the largest among all unscheduled tasks and then allocates the task to the processor on which the earliest completion time can be achieved. The only difference between Max-Min and Min-Min is that Min-Min first schedules the task with the shortest earliest completion time. Sufferage differs from the two algorithms as it first schedules the task with the largest sufferage value among all unscheduled tasks. The sufferage value of a task is defined as the difference between its earliest completion time and its second earliest completion time.

The aforementioned works relied on accurate performance prediction information on underlying resources. But in real-life distributed systems such information may be difficult to obtain. This motivates the design of knowledge-free approaches, such as RR [80] and WQR [81]. These heuristics require no performance prediction information on resources. RR makes replicas of running tasks in a round-robin fashion after conducting list scheduling for all of the unscheduled tasks. WQR schedules tasks in an arbitrary order and dispatches them on the resources as soon as they become available. When all tasks were scheduled, WQR starts to create replicas, but it limits the number of times that a task may be replicated.

Chapter 3

Models

3.1 The Task Model

In this thesis, divisible load applications are of particular interests. These divisible loads consist of massive identical, independent, and low-granularity data units which can be distributed to networked computers for parallel processing. Also, the divisible loads may be of different types. For instance one application may deal with files and another with matrices. Following Marchal et al.'s model [11], we have some definitions below. For each load L_j $(j = 1, 2, \cdots)$, let S_j be the size (in bytes) of L_j and W_j be the workload, i.e., the amount of computations (e.g., floating point operations) required to compute L_j. We partition each load into a large number of identical, independent, and low-granularity load units and let N_j be the total number of units of load L_j. Also, we let S_j^u be the size (in bytes) of a load unit of L_j and let W_j^u be the workload of a unit of L_j, i.e., the amount of computations (e.g., floating point operations) required to compute a unit of L_j. Thus for a divisible load L_j, $S_j = S_j^u * N_j$ and $W_j = W_j^u * N_j$. The parameters W_j^u and S_j^u represent the task granularity of load L_j. We can obtain W_j^u by testing one data unit of the load L_j on a single processor and thus acquire the total amount of computations of L_j as N_j is known [9]. When describing the divisible loads, we use terms "task" and "load" interchangeably.

3.2 The System Model

Our target computing platform is a cluster system connected through a local area network (LAN). The input data are divisible loads originally residing in a single machine, P_0, which is the master node for scheduling the loads. From P_0's perspective, the logical topology of the platform is actually a star topology. This star topology is the generic architecture for implementing master-worker computations and widely used by prior works including [3, 4, 5, 6]. Let M be the number of processing nodes in the system and the nodes are denoted as $(P_1, P_2, \cdots, P_M)$. P_0 does not participate in processing but dispatches loads to the M nodes for parallel processing. Each processor P_i $(i = 1, 2, ..., M)$ has a compute speed C_i. The communication speed (bandwidth) of the LAN is B. Following prior works [17, 67, 71], we assume that B and C_is are fixed as the cluster is a dedicated system for the master node P_0 to execute the loads.

In this thesis, start-up time overheads are considered to be negligible and we adopt a linear cost model, which is widely used in many prior DLT studies including [2, 6, 11]. In this linear model, it takes $\frac{XS_j^u}{B}$ time units to send X units of load L_j from P_0 to each processor P_i. Similarly, each processor P_i has a compute power C_i. It takes $\frac{XW_j^u}{C_i}$ time units to compute X units of L_j on P_i.

We assume that P_0 sends loads in a sequential mode: Loads are not sent to the processors simultaneously. Although current technologies allow simultaneous load transmission to multiple processors, this parallel communication mode cannot significantly improve the performance of our strategies in LAN environments primarily for two reasons. Firstly, the overall communication throughput of P_0 is restricted by the bandwidth of the LAN, not the number of receivers. Secondly, as our strategies are shown to guarantee efficient overlapping of communications with computations, there is very little room, if not none, for any parallel mode to improve performance. Therefore, we adopt the sequential mode as the parallel mode can cause unnecessary complexities for scheduling. Notice that in some other scenarios the parallel mode may significantly outperform the sequential mode and we will discuss this in Chapter 8.

Finally, we assume that each processor has adequate storage to store and compute any amount of data. When compared to the time taken for computation and communication, the time for reporting the result back to P_0 is negligible. The time taken for making scheduling decisions is also negligible.

We now present a list of general notations that will be used throughout the thesis. Other relevant and specific notations used in later chapters will be introduced in respective chapters.

- B: bandwidth of the underlying network.
- C_i: compute speed of P_i.
- J: number of divisible load applications to be scheduled.
- L_j: j-th load ($j = 1, 2, \cdots, J$).
- M: total number of processors in the system.
- N_j: the total number of units of L_j.

- P_0: master node who distributes tasks to processors.
- P_i: i-th processor ($i = 1, 2, \cdots, M$).
- S_j: the size (in bytes) of L_j.
- S_j^u: the size (in bytes) of a unit of L_j.
- W_j: the workload of L_j, which is the amount of computations (e.g., floating point operations) required to compute L_j.
- W_j^u: the workload of a unit of L_j, which is defined as the amount of computations required to compute a unit of L_j.

Chapter 4

Scheduling Multiple Divisible Loads with Arbitrary Processor Release Times

This chapter investigates the problem of scheduling multiple divisible loads in networked computer systems with a particular emphasis in capturing two important real-life constraints, the arbitrary processor release times (or ready times) and heterogeneous processing requirements of different loads. We study two distinct cases of interest, static case, where processors' release times are predetermined and known, and dynamic case, where release times are unknown until processors are released. To address the two cases, we propose two novel **scheduling strategies**, referred to as *Static Scheduling Strategy* (SSS) and *Dynamic Scheduling Strategy* (DSS), respectively. In addition, we capture tasks' processing requirements in our strategies, a unique feature that is applicable for handling loads on networks that run proprietary applications only on certain nodes. Thus each task can only be processed by some certain nodes in our formulation. To handle the contention of multiple applications that have various processing requirements but share the same processing nodes, we propose an efficient **load selection policy**, referred to as *Most Remaining Load First* (MRF). We integrate MRF into our SSS and DSS to address the problem of scheduling multiple divisible loads with arbitrary processor release times and heterogeneous requirements. We evaluate the strategies using extensive simulation experiments.

4.1 Problem Formulation

The cluster system in this chapter consists of M processors $(P_1, P_2, \cdots, P_M)$ which have different release times (or ready times). The ready time of processor P_i $(i = 1, 2, ..., M)$ is T_i^R. We consider both static and dynamic cases of arbitrary processor release times. In the static case processor release times are predetermined and known before the execution starts. In contrast, in the dynamic case a processor's release time is unknown until the processor is released. Once a processor is released, it can be fully utilized by the master node P_0 for processing loads.

The scheduling problem addressed in this chapter is to distribute J divisible load applications $(L_1, L_2, ..., L_j)$ onto M processors so that the total processing time of all loads can be minimized. As we consider different types of applications and heterogeneous processing nodes, these divisible loads have different computing requirements and each load can only be processed by some certain processors. To guarantee that all loads can be successfully completed, each load can be processed by at least 1 processor.

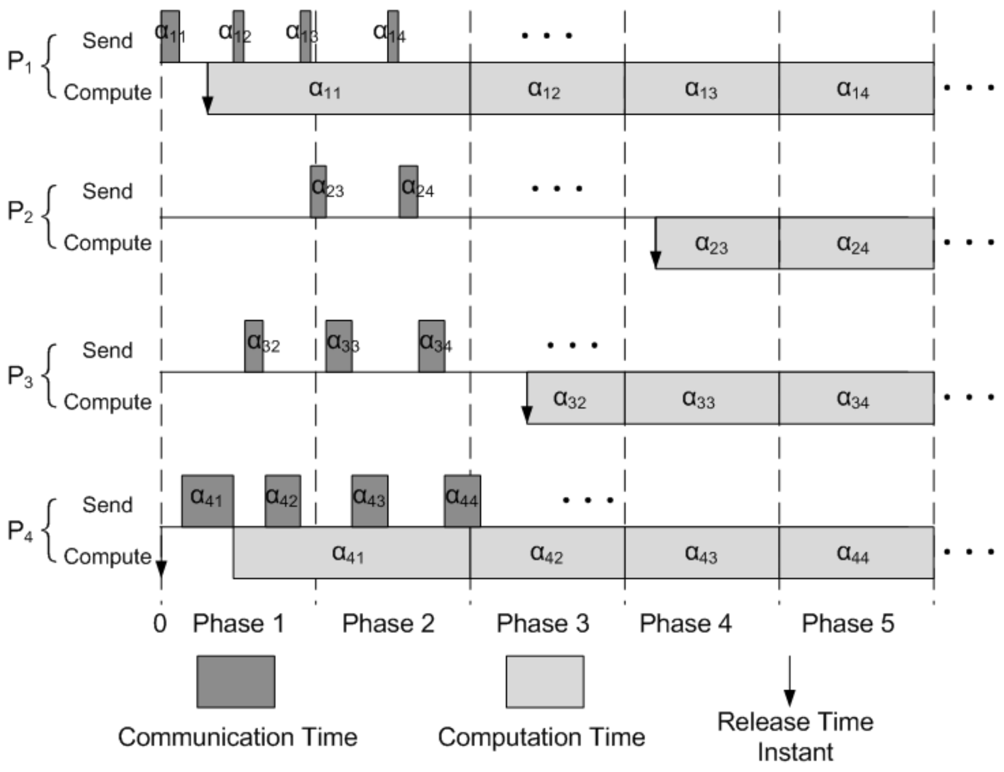


Figure 4.1: Timing diagram of SSS.

4.2 Static Scheduling Strategy

In this section, we consider scheduling multiple divisible loads in the case of static processor release times and present *Static Scheduling Strategy* (SSS). Fig. 4.1 shows the timing diagram of SSS for an example of 4 processors and one load. The identifier α_{ik} indicates the load chunk allocated to processor P_i in round k. As shown in Fig.4.1, SSS is a phase-based multi-round approach. SSS works in an incremental fashion, consuming several rounds for scheduling and computing loads. We use the term "load chunk" to denote a number of load units distributed in a round to a processor. In each round, SSS works by iterating the steps of selecting a load, selecting a processor, and then allocating a load chunk from the load to the processor. The loads distributed in each round is computed in a corresponding phase. The length of each phase is T_p, which is fixed and predetermined. Below we describe the detailed design of SSS and we consider an arbitrary round k ($k = 1, 2, 3, \cdots$) as an example to illustrate the working flow of SSS.

As multiple heterogeneous loads are involved, P_0 needs to first select a load before it can allocate a load chunk to some processor. The key issue of load selection is to address the contention of various loads which share the same compute resources but have various processing requirements. To design an efficient load selection policy, we need to achieve high processor utilization in scheduling. If a processor is only able to compute certain loads but all those loads have been completed, it will remain idle and this naturally wastes its compute power. Also, we observe that such wastes severely happen at the end of the entire execution as many loads gradually get completed and hence many processors will remain idle. Specifically, if the processing of a load completes much later than others, many processors that cannot compute it have to remain idle for long time, resulting in serious compute power waste. Therefore, to minimize this waste and achieve high performance, we should provide fairness to different loads and prevent some loads from being completed much later than other loads. According to this principle, we propose a fair load selection policy, referred to as *Most Remaining Load First* (MRF). This policy selects the load with the most remaining amounts (in terms of required computations). We denote L_j^R as the total amount of computations required for processing all un-allocated units of L_j. We can write L_j^R as:

$$L_j^R = N_j^R W_j^u \tag{4.1}$$

where N_j^R is the number of un-allocated units of L_j. Thereby, among all available loads, MRF selects the one with the largest L_j^R. Since MRF first selects the loads with the most remaining amounts, it can effectively prevent some loads from being completed much later than other loads. Notice that at the beginning of each round all remaining loads are set as available. If P_0 cannot find any processor for a selected load in round k, then this load will be set as unavailable.

Suppose load L_j is selected for scheduling. Now P_0 needs to select a processor and allocate it a load chunk from L_j. To make efficient scheduling decisions, we should first consider the workings of the processors to understand the rationale behind the scheduling. In SSS, the processors compute chunks once they have any unprocessed chunks. Also, they process chunks in a first-come-first-serve way. Hence, following a common strategy, in each round k P_0 attempts to dispatch appropriate amounts of loads to processors so that the processors can finish their chunks of round k simultaneously when phase $(k+1)$ ends (as shown in Fig. 4.1). The strategy that guarantees processors to finish simultaneously is widely used in many DLT works [1] since this strategy achieves load balancing and optimizes resource utilization.

For processor selection in round k P_0 only considers the processors satisfying 3 criteria: Firstly, such processors are capable of processing L_j. Secondly, such processors are released before phase $(k+1)$ ends because all processors are scheduled to finish the chunks of round k when phase $(k+1)$ ends to conform to the above rule. Thirdly, such processors have idle time before phase $(k+1)$ ends. If a released processor is allocated an adequate amount of load to continually compute until phase $(k+1)$ ends, the processor is categorized as having no idle time before phase $(k+1)$ ends; otherwise, it is classified as having idle time before phase $(k+1)$ ends.

The processors satisfying the 3 criteria are considered for L_j and P_0 picks one among them to allocate a chunk. Since all processors have the same communication speeds and their compute speeds are independent to loads, thus P_0 selects a processor among all available ones. If no processor is available for L_j, L_j will be set as unavailable for round k. This means L_j cannot be scheduled in this round but it will be considered in next rounds. Then P_0 executes MRF again to select another load. If no loads are available, P_0 will end round k. It may be noted that two cases can cause that no loads are available. The first is that all loads have been finished allocation. The second is that all released processors that can compute the loads have been allocated enough load chunks so that they have no idle time. In this case, P_0 will immediately start round $(k + 1)$.

There is another rule for ending round k as explained below. The transmission of round k must be finished before phase $(k+1)$ starts so that processors can start immediately to compute the chunks of round k in phase $(k + 1)$ and P_0 can also start immediately to transmit the chunks of round $(k+1)$ in phase $(k+1)$. Thus, when phase k ends P_0 will immediately terminate the communication of round k and start round $(k + 1)$. Therefore, if there are too many processors so that P_0 cannot utilize all of them, then P_0 will abandon some processors, guaranteing that the communication of round k can be finished before phase $(k+1)$. This rule also guarantees the loads with the most remaining amounts can be timely scheduled in the next round.

Suppose P_0 has selected load L_j and processor P_i for scheduling, now a load chunk can be allocated from L_j to P_i. In round k P_0 dispatches appropriate amounts of load to P_i so that the chunk can be guaranteed to reach P_i before phase $(k + 1)$ starts and P_i can finish the chunk when phase $(k + 1)$ ends (as shown in Fig. 4.1). To allocate a chunk, P_0 first calculates P_i's available compute time before phase $(k + 1)$ ends. Then it can allocate a chunk according to the available time. We denote this time as T_{ik}^{comp}. To obtain T_{ik}^{comp} we should consider 4 constraints. Firstly, P_i can process loads only after it becomes ready. Secondly, P_i can process the new chunk only after it finishes previously allocated chunks. Thirdly, even if P_i is ready and idle, necessary communication time should be subtracted from the total available time. Finally, the communication of P_i should be finished before phase $(k+1)$ starts. We can obtain T_{ik}^{comp} as follows:

$$\max T_{ik}^{comp} \tag{4.2}$$

subject to:

$$\begin{cases} T_{ik}^{comp} \leq T_p(k+1) - T_i^R \\ T_{ik}^{comp} \leq T_p(k+1) - T_i^F \\ T_{ik}^{comp} \leq \dfrac{T_p(k+1) - T^{comm}}{1 + \frac{C_i S_j^u}{BW_j^u}} \\ T_{ik}^{comp} \leq \dfrac{BW_j^u (kT_p - T^{comm})}{C_i S_j^u} \end{cases} \tag{4.3}$$

where T_i^F is P_i's finish time of its previously allocated chunks, which is obtained according to prior load allocation. T^{comm} is the time instant when P_0 is allocating loads to P_i and thus $(T_p(k+1) - T^{comm})$ is the total available time when P_0 is allocating loads to P_i. The above linear program can be simplified as follows:

$$T_{ik}^{comp} = \min(E_1, E_2, E_3, E_4) \tag{4.4}$$

where:

$$\begin{cases} E_1 = T_p(k+1) - T_i^R \\ E_2 = T_p(k+1) - T_i^F \\ E_3 = \dfrac{T_p(k+1) - T^{comm}}{1 + \frac{C_i S_j^u}{BW_j^u}} \\ E_4 = \dfrac{BW_j^u (kT_p - T^{comm})}{C_i S_j^u} \end{cases} \tag{4.5}$$

As E_1, E_2, E_3, and E_4 in Equation (4.5) can be directly calculated, we can easily determine T_{ik}^{comp} and obtain the chunk size α_{ijk} (in terms of number of units) allocated to P_i from L_j in round k. In reality, this number should be an integer. Thus we can write α_{ijk} as follows:

$$\alpha_{ijk} = \lfloor \frac{C_i T_{ik}^{comp}}{W_j^u} \rfloor \tag{4.6}$$

If N_j^R is less than the chunk size α_{ijk}, then α_{ijk} is accordingly reduced to that number and all un-allocated units of L_j are allocated to P_i. In this case, P_i will still be available after allocated the chunk since it still has some idle time before phase $(k+1)$ ends. After dispatching the chunk, P_0 iterates the above steps of load selection, processor selection, and load allocation until no loads are available for this round, or phase k terminates. Then P_0 terminates round k and starts round $(k+1)$, repeating scheduling loads until all the loads are allocated.

Now we discuss on the mechanism of SSS and highlight its advantages. From Fig. 4.1 we can observe that, in phase $(k+1)$, while computing the chunks of round k, processors can simultaneously receive the chunks of incoming rounds. In this way, communications and computations are pipelined and overlapped to save time. Prior DLT works usually adopt a single-round approach, in which processors need to wait for P_0 (thus, wasting time) to transmit loads to other processors before they can start their own communication and computation [4, 6]. But in SSS such waste is significantly reduced by efficient overlapping. In addition, in each round P_0 dispatches loads according to processors' available time so that all processors can simultaneously finish their loads of that round. This naturally achieves load balancing in each round. Further, SSS offers efficient resource utilization since processors can continuously work throughout consecutive phases, provided processors can process enough numbers of loads.

It may be noted that in the last round, processors may finish computing at different times. But this influences the overall performance little because the unbalancing is limited in one phase and for all other rounds efficient load balancing is guaranteed. However, if the number of rounds is too small, for example, less than 5, unbalanced amounts of load in the last round may become considerable. Therefore, the length of each phase T_p is not arbitrarily chosen. If T_p is too large, unbalanced load in the final round will become significant and seriously weakens the performance. In addition, the time taken in waiting for communication in round 1 will also become considerable. On the other hand, T_p cannot be too small because the data units cannot be infinitesimally small. Therefore, we should avoid determining T_p as either very small or very large. When implementing SSS in real-world applications, we should check the task granularity and the total size of the data before T_p can be determined. In Section 4.4 we will evaluate the impact of T_p on our strategies.

4.3 Dynamic Scheduling Strategy

Now we tackle a more realistic situation wherein processors that will eventually participate are unknown until they become available. To handle unexpected processor releases, we revisit SSS to carry out an alternate design, namely *Dynamic Scheduling Strategy* (DSS). DSS is again a phase-based multi-round approach. Fig. 4.2 is a timing diagram of DSS showing an example of 4 processors and 1 load. The identifier α_{ik} means the chunk allocated to processor P_i in round k. Same as SSS, in each round, DSS iterates the steps of selecting a load by MRF, selecting a processor, and then allocating a load chunk from the load to the processor accord-

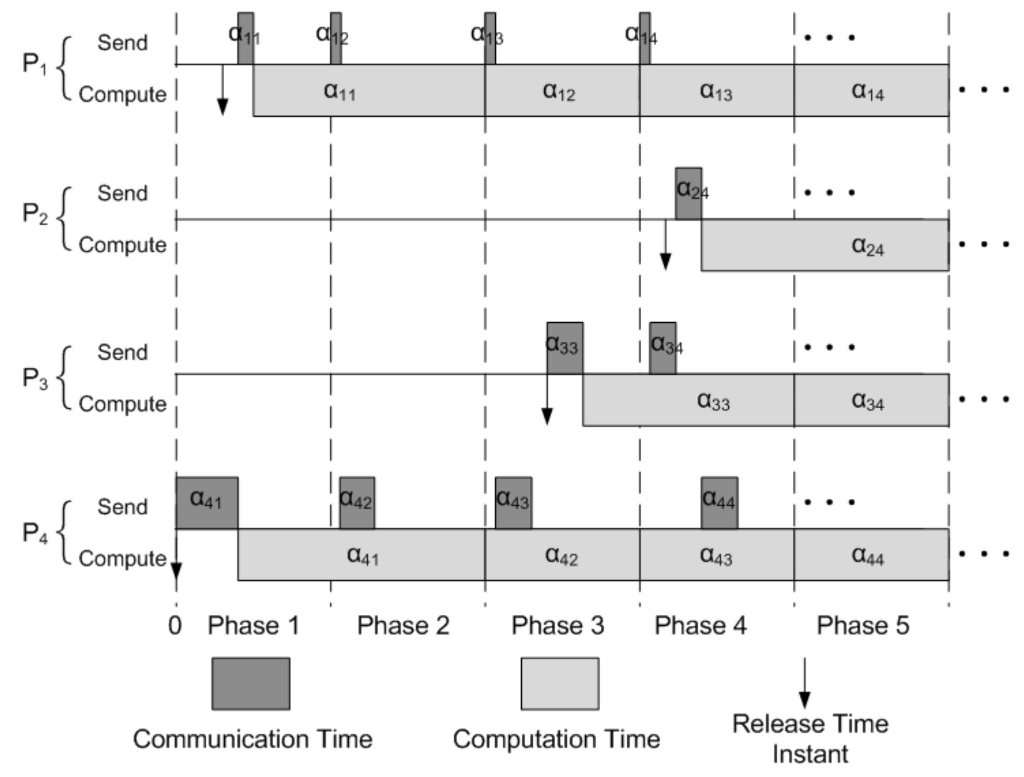


Figure 4.2: Timing diagram of DSS.

ing to Equations (4.4) and (4.6). These steps repeat until all loads are completed. Since the steps have been thoroughly described in Section 4.2, we omit to detail these again, but below we intend to describe the differences between SSS and DSS.

In SSS load distribution can be determined before the entire execution, but in DSS the distribution should be dynamically generated in each round since a processors' participation is unknown until it is released. In other words, we should avoid making scheduling decisions too early since such decisions may result in poor utilization of new processors. Therefore DSS works in a periodic fashion: For each round k, P_0 starts dispatching at the beginning of phase k and distributes the chunks of this round in phase k. Thus the chunks sent in phase k can timely reach the processors before the computation of round k starts in phase $(k + 1)$.

Same as SSS, in DSS the transmission of all chunks of round k must be finished before phase $(k+1)$ starts so that processors can timely start to compute these chunks in phase $(k+1)$ and P_0 can also timely start to transmit the chunks of round $(k+1)$ in phase $(k+1)$. When phase k ends P_0 terminates round k and immediately starts the next round. Therefore the communication time of round k is only limited in phase k (as shown in Fig. 4.2).

Notice that P_0 may finish the communication of round k before phase k ends. But P_0 will not start round $(k+1)$ until phase $(k+1)$ starts. This is because a processor's ready time is unknown until it is released; if load distribution is determined too early and loads are sent too early, when new processors are released the loads may have been allocated to early-released processors and the new processors may not be involved in computation. Although it is possible to redistribute loads to the new processors and the early-released processors so that the new processors can also be utilized, this process will waste communication resources, and become cumbersome to handle scheduling and load balancing, especially in the presence of heterogeneous loads. Therefore, in DSS P_0 always starts the communication of round k at the beginning of phase k and works in a periodic way as we mentioned before. After sending loads to all available processors, P_0 will remain idle, waiting until phase k ends, and then start a next round.

DSS handles newly released processors in a different way to SSS. In fact, in SSS new processors are not explicitly addressed, but in DSS extra efforts are needed to handle dynamic processor releases. In DSS, P_0 keeps monitoring new releases. If a new processor P_i becomes ready in phase k, P_0 will realize this immediately. If P_0 is busy when P_i becomes ready, P_0 will handle it after the current transmission; otherwise, P_0 will immediately address it. To handle the new processor, P_0 checks all the loads that can be processed by P_i. If some loads that can be executed by P_i are set as unavailable but they have un-allocated units, P_0 will set such loads as available. Thus these loads can be considered again in this round and P_i can also be utilized immediately. Then P_0 continues the steps of scheduling loads.

Now we conclude the mechanisms and benefits of DSS. In order to handle new processors dynamically, DSS works in a periodic fashion so that scheduling decisions will not be made too early. Also, DSS elegantly handles unexpected release times by dynamically scheduling loads to new processors once they become ready. Moreover, same as SSS, DSS provides efficient load balancing and resource utilization as we analyzed in Section 4.2. DSS also implements efficient pipelining and overlapping of communication with computation (as shown in Fig. 4.2) to optimize processing time. The complexity of both SSS and DSS for scheduling J tasks with MRF is $O(J^2)$.

4.4 Performance Evaluation

In this section, we evaluate the performance of SSS and DSS strategies by rigorous simulation experiments. We typically follow the style of simulation study used in most earlier studies in DLT literature [2, 4, 6].

To evaluate SSS and DSS, we compare their performance with an ideal case (the performance bound) as there are no strategies available in the literature to compare in this current problem context directly. For the ideal case, we assume that arbitrary processor releases are perfectly handled so that each processor is immediately involved in computation once it is released. Also, we assume that the ideal case is not influenced by the granularity of the loads, which means the number of load units can be rational numbers when simulating the ideal case. In addition, in the ideal case communications can be “optimally” overlapped with computation such that communication delays are zero. However, if the network bandwidth is too small, communications may not absolutely hidden even if an “optimal” scheduling policy is applied. Hence we recognize a performance bound for the communication of the ideal case. That is, in the ideal case the total processing time of X units of load L_j cannot be less than the minimum communication time for transmitting such a load chunk, which is $\frac{XS_j^u}{B}$ seconds. If a calculated processing time of the ideal case is less than its minimum communication time, we let the processing time be the minimum communication time. Therefore the ideal case presents the performance bound among all possible solutions since the resource utilization of either computation or communication is 100% in this case. In the following figures the results of the ideal case are denoted as “Ideal”.

The normal experiment sets are denoted as “SSS-MRF” and “DSS-MRF”. Also, we simulate an ideal load allocation case (ILA), in which each processor can compute all loads. We apply ILA to both SSS and DSS to produce “SSS-ILA” and “DSS-ILA”, respectively. Thus, by comparing SSS/DSS-MRF with SSS/DSS-ILA, we can study the effect of heterogeneous loads and evaluate MRF. By comparing SSS/DSS-ILA with the ideal case, we can evaluate the effectiveness of SSS/DSS in handling homogeneous loads.

The simulated system consists of randomly generated processing nodes. The initial configurations are set as follows: The number of processing nodes M is 20 and the number of loads J is 10. For each processor P_i the computation power C_i is uniformly distributed among $[100, 500]$ and the communication speed B is set as 100. Processors' ready times are uniformly distributed among $[0, 100]$ time

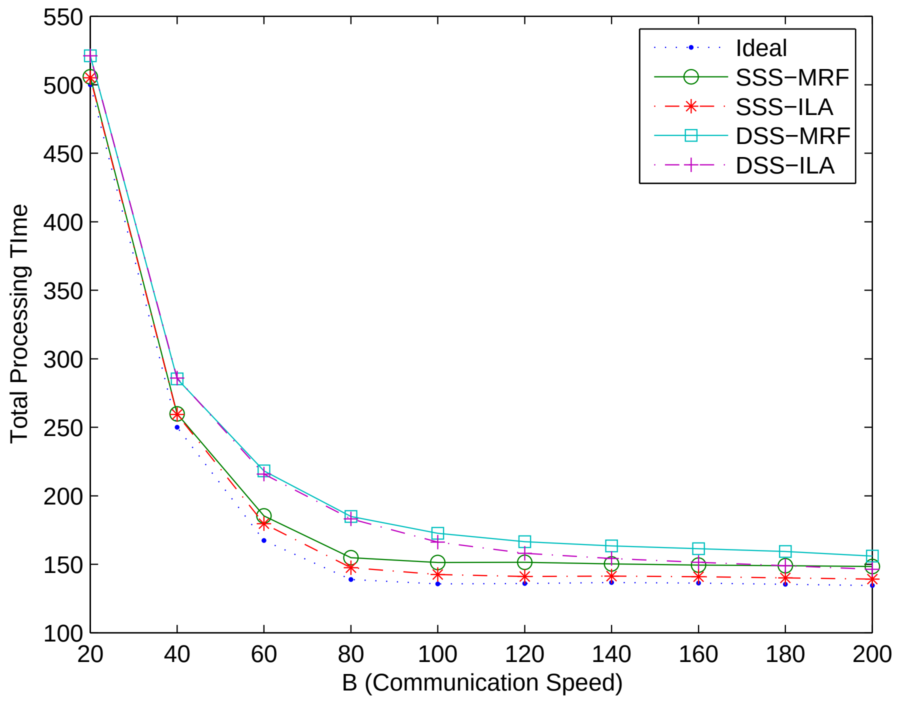


Figure 4.3: Average total processing time versus B.

units. As stated before, for SSS the processors' ready times are known before execution starts and for DSS the processors' ready times are unknown until they are released. In addition, to study the influence of different load types and sizes, we let S_j^u be uniformly distributed among $[0.5, 1.5]$, let W_j^u be uniformly distributed among $[1, 100]$, and let N_j be uniformly distributed among $[500, 1500]$. The length of each phase T_p is 10 time units. To study the influence of multiple loads with heterogeneous requirements, we use parameter p to denote the probability that a load L_j can be processed by any processor. We let p_j be uniformly distributed among [0.2, 0.8].

In the following experiments we vary our interested parameters while fixing other parameters as their initial values to study the effect of the interested parameters. We first vary B from 20 to 200 and the corresponding processing time

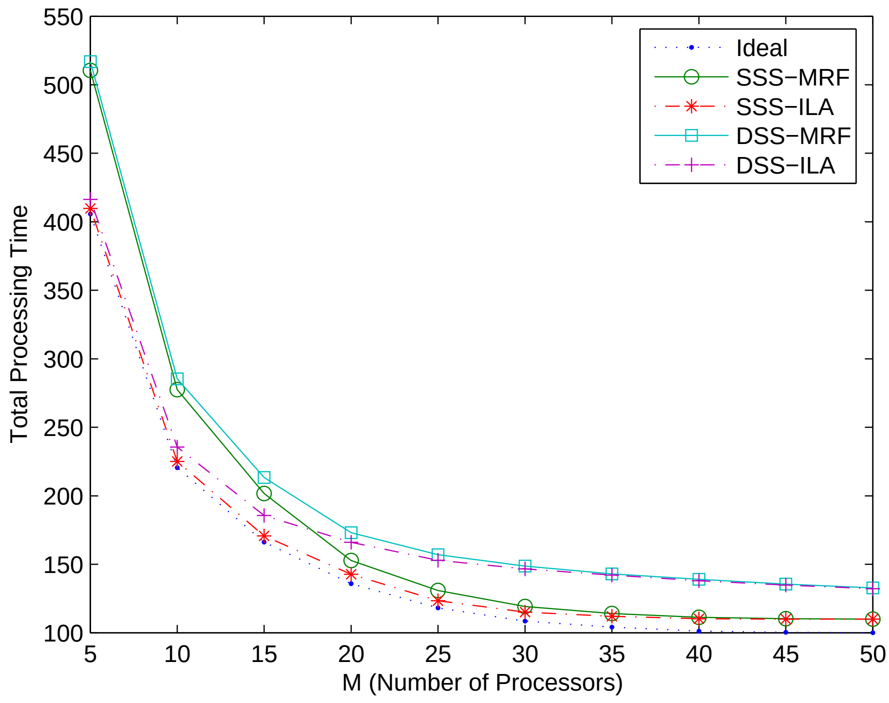


Figure 4.4: Average total processing time versus M.

is shown in Fig. 4.3. In addition, we vary M from 4 to 40 and Fig. 4.4 plots the processing time versus M. Further, we vary J from 2 to 20 and Fig. 4.5 depicts the processing time versus J. Then, to study the effect of p_js, we let p_js be uniform distributed among $[p_a - 0.1, p_a + 0.1]$ and we vary p_a among $[0.2, 0.8]$, and the corresponding processing time is shown in Fig. 4.6. Moreover, to investigate the effect of different load groups with different p values, we consider a load set consisting of two groups - group 1 are "easy" loads for which p_j is uniformly distributed among $[0.5, 0.9]$, and group 2 are "hard" loads for which p_j is uniformly distributed among $[0.1, 0.5]$. We use p_b to denote the proportion of the number of loads in group 1 to the number of the loads in the whole load set. We vary p_b among $[0, 1]$ and Fig. 4.7 plots the processing time versus p_b. Finally, we vary T_p from 0.3125 to 160 and the corresponding processing time is shown in Fig. 4.8.

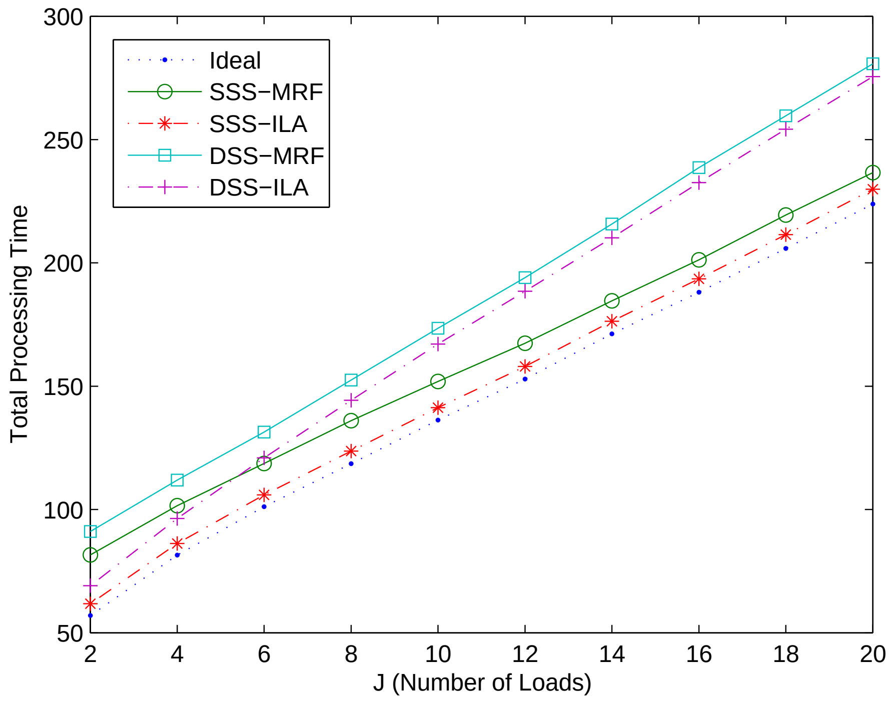


Figure 4.5: Average total processing time versus J.

The results in these figures show that both SSS and DSS strategies are efficient, and their performance is close to to the ideal case in most cases. The only exceptions are located in Fig. 4.6, 4.7, and 4.8, where the performance of both SSS/DSS-MRF is much worse than the ideal case when processors cannot compute enough loads or T_p is not properly chosen. In addition, SSS outperforms DSS in most cases in our evaluations. This is because in SSS P_0 can dispatch loads to new processors before they become ready, but in DSS new processors have to wait until P_0 knows them and then dispatches loads to them. This results in higher resource utilization in SSS than in DSS. Moreover, the available communication time of each round is limited to one phase in DSS , but may be more than one phase in SSS. Accordingly, in each round P_0 can send more loads in SSS than in DSS.

Fig. 4.3 shows that the processing time of all experiment sets stabilizes when

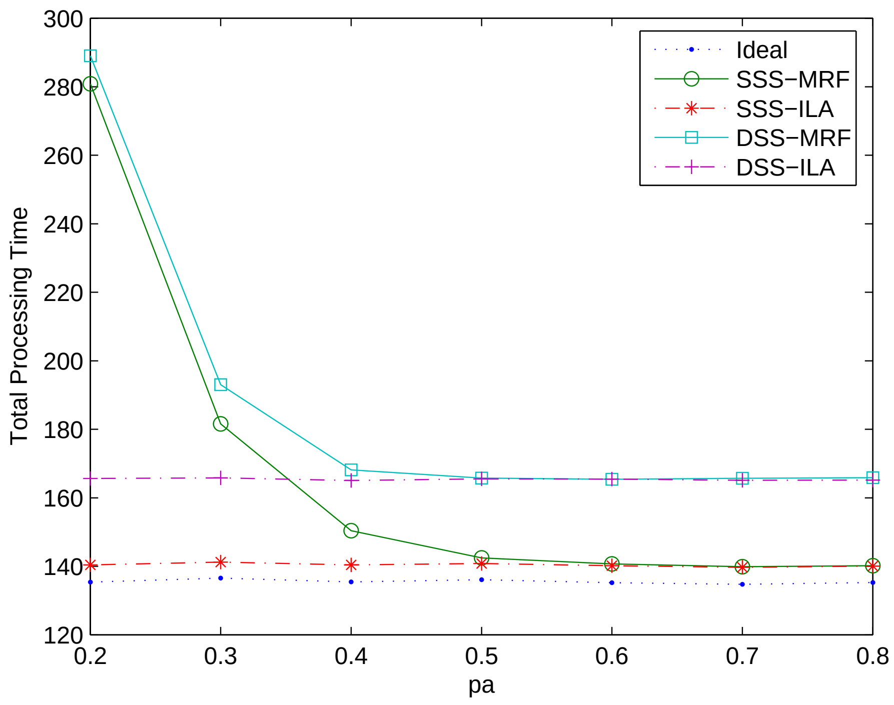


Figure 4.6: Average total processing time versus p_a.

B is greater than 80. This is due to the fact that when the network bandwidth is large enough communication can be efficiently overlapped with computation and hence varying B impacts little on the performance. In the contrary, when B is smaller than 80, the processing time sharply increases as B decreases since when B is small, communication requires too much time and thus cannot be effectively hidden. Moreover, when B is large the performance gap between SSS and DSS becomes small. A possible reason is that since the advantages of SSS to DSS lie in communications, the advantages will become small when communication speeds are enough high. Similarly, when B is small all experiment sets deliver close performance, indicating that for slow networks communication delays dominate the performance and other factors become negligible.

Fig. 4.4 shows that as M increases the processing time of all sets decreases as

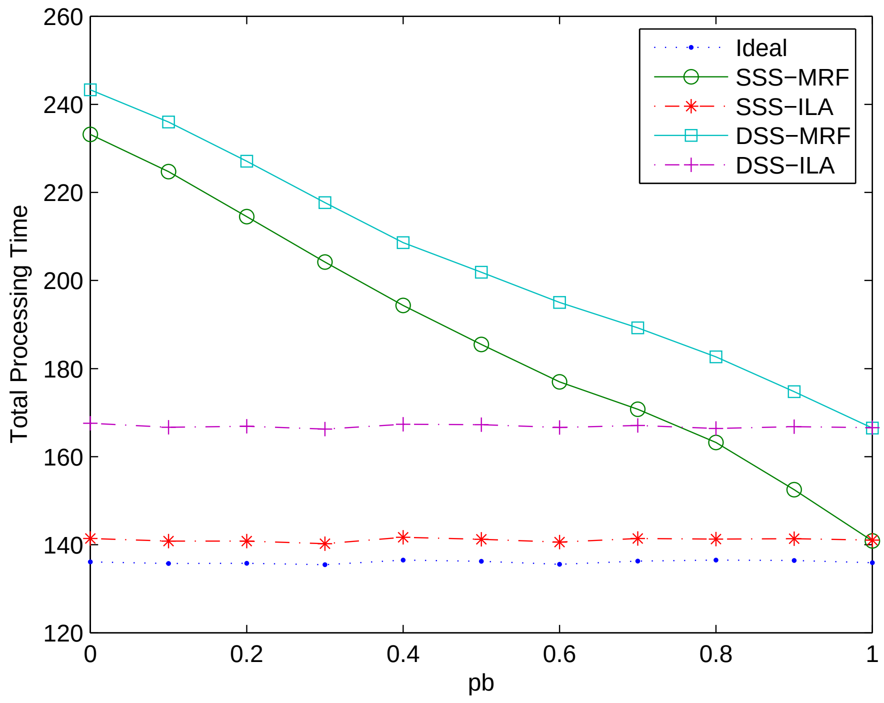


Figure 4.7: Average total processing time versus p_b.

more nodes computing in parallel saves more time. But the performance of SSS and DSS gradually stabilizes when M is greater than 30. A plausible explanation is that when there are abundant processors, network bandwidth becomes a bottleneck and restricts resource utilization. Further, the performance gap between SSS/DSS-ILA and SSS/DSS-MRF is significant when M is small, but gradually decreases as M grows. This indicates that if there are only a few processing nodes, MRF cannot always find proper nodes for scheduling, but if there are abundant nodes probably MRF can find proper nodes.

From Fig. 4.5 we observe that the performance gap between SSS/DSS-MRF and the ideal case gradually decreases as J grows. One contributing factor is that the required computation time is approximately proportional to the total load size but the communication delays which cannot be overlapped are merely related to T_p,

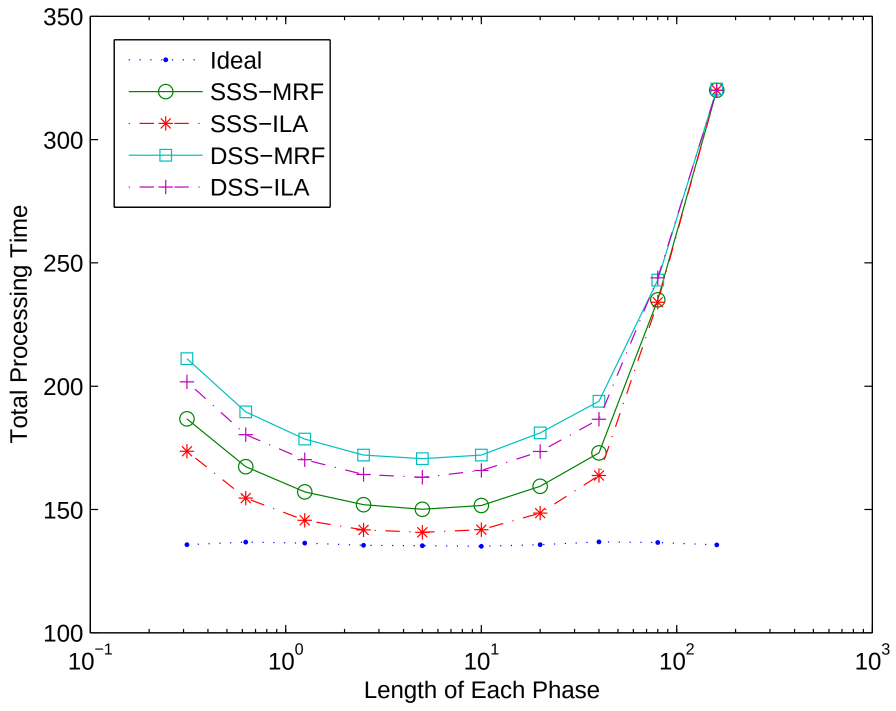


Figure 4.8: Average total processing time versus T_p.

which is fixed. Thus the effect of communication delays become more significant when the processing time shortens as J decreases. Another contributing factor is that when there are a few loads, the choices for MRF are very limited. Thus, MRF may not be as efficient as ILA. This also explains the fact that the performance gap between ILA and MRF decreases as J grows.

From Fig. 4.6 we observe that when p_a is less than 0.4 the processing time of SSS/DSS-MRF sharply increases as p decreases. A plausible explanation is that when p_a is less than 0.4, a processor is unable to compute enough number of loads and thus cannot be fully utilized. Thereby, increasing p_a enables the processors to compute more loads and thus improves the performance. On the other hand, when p_a is greater than 0.4 their performance stabilizes as p_a grows. This is because when p_a is greater than 0.4, a processor becomes efficient to process enough number of loads and thus be fully utilized in computation. Accordingly, increasing p_a impacts little on the performance. Fig. 4.7 shows another case wherein processors cannot compute enough loads for group 2. Hence increasing p_b enables the processors to compute more loads and thus improves the performance.

Fig. 4.8 shows that when T_p is greater than 40, the processing time of SSS/DSS sharply increases as T_p increases. This is due to the fact that the time taken in waiting for communication in round 1 becomes considerable when there are only a few rounds. Also, when T_p is too large, load unbalancing in the final round becomes considerable and seriously weakens the performance. In the contrary, when T_p is less than 1.25, the processing time of SSS/DSS significantly increases as T_p decreases. This is because load units cannot be infinitely small. If T_p is too small probably some processors cannot receive a unit as they cannot finish 1 unit in 1 phase. Further, the performance of SSS/DSS stabilizes when T_p is among $[1.25, 40]$. This shows that although T_p cannot be too small or too large, there is still a wide range in which varying T_p only slightly influence the performance of SSS/DSS. This range is related to the granularity and the sizes of the loads. As SSS and DSS are designed for processing divisible loads consisting of large numbers of low-granularity computations, choosing satisfactory values of T_p may be not challenging. When implementing SSS/DSS for real-world applications, simulations or experiments can help determine satisfactory values of T_p once the granularity and size of the data are given.

Chapter 5

Scheduling Real-Time Divisible Loads

In the last chapter, we have addressed the requirement-aware scheduling problem for processing multiple divisible loads with arbitrary processor release times. To further enhance quality-of-service (QoS) and provide performance guarantees in distributed computing environments, in this chapter we investigate the requirement-aware problem in real-time cluster systems. We propose scheduling algorithms referred to as *Requirements-Aware Real-Time Scheduling*(RARTS) algorithms, which consist of a novel scheduling policy, referred to as *Minimum Slack Capacity First* (MSCF), and two multi-round load distribution strategies, referred to as *All Eligible Processors* (AEP) and *Least Capability First* (LCF). We perform rigorous performance evaluation studies to quantify the performance of our strategies on a variety of scenarios.

5.1 Problem Formulation

Consider a real-time aperiodic divisible load model in which for each load L_j, A_j is its arrival time and D_j is its deadline. As we consider diverse tasks and heterogeneous processors, the divisible loads have different computing requirements and hence each load can only be processed by certain processors. We define processor P_i's capability on computing L_j as C_{ij}. If processor P_i can process L_j, $C_{ij} = 1$; otherwise $C_{ij} = 0$. To guarantee that all loads can be successfully completed, each load can be processed by at least 1 processor.

The scheduler has complete knowledge about currently active list of tasks, but not about future tasks which have not arrived. When a new task arrives, the scheduler runs an admissibility test to dynamically determine whether it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. That is, the scheduler generates a task list containing the new task and all previously admitted tasks. It then executes RARTS algorithms to schedule all loads in the list. Upon completion of the test, if all tasks can be completed within their respective deadlines, it can generate a new schedule to replace the existing schedule and the new task is accepted. Otherwise, it rejects the new task and maintains the existing schedule.

5.2 Design and Analysis of RARTS Algorithms

This section presents the detailed design and analysis of RARTS algorithms. We describe the MSCF scheduling policy in Section 5.2.1 followed by two load distribution strategies, AEP and LCF, in Section 5.2.2 and 5.2.3, respectively. Notice that a complete load management process requires both a scheduling policy (e.g., MSCF) and a load distribution strategy (e.g., AEP or LCF).

5.2.1 The MSCF Scheduling Policy

As multiple real-time tasks are involved, P_0 first needs to adopt a scheduling policy to determine the order in which the tasks are to be considered for allocation. The most widely used urgency-based scheduling policy is the *Earliest Deadline First* (EDF) method, which orders tasks by their absolute deadlines. Another well-known scheduling policy is the *Least Laxity First* (LLF) approach, which first selects the task that has the smallest slack time, defined as the time difference between the deadline and the finish time of a task if it is started now.

For a set of real-time divisible loads with heterogeneous processing requirements, EDF and LLF may not properly reflect their real urgency. We can immediately realize this via the following example. Consider processing 2 loads on 5 processors. The computation speed of each processor is 1 data unit per time unit. Load L_1 has 15 units of data and its deadline is 4. Load L_2 has 5 units of data and its deadline is 7. Also, all processors can process L_1 but only processor P_1 can process L_2. It takes 3 time units to process T_1 on all processors and 5 time units to process L_2 on P_1. Both EDF and LLF will schedule L_1 first on all processors according to DLT rule [1], causing L_2 to violate its deadline. However, if we schedule L_2 first on P_1, L_1 can also be successfully scheduled on the other 4 processors. This example shows that to schedule real-time divisible loads with heterogeneous processing requirements, we shall consider processors' capabilities in determining tasks' deadlines.

To this end, we present the *Minimum Slack Capacity First* (MSCF) policy to handle real-time divisible loads with various requirements. For a given load L_j, we only consider the processors which are capable of processing L_j and become available before L_j's deadline. Such processors are denoted as "eligible processors" for L_j. Since a divisible load can be partitioned into small pieces for parallel processing, the total available computation capacity for L_j before its deadline is denoted as C_j^a, which is the amount of computations that can be performed using all L_j's eligible processors before L_j's deadline. We can obtain C_j^a as follows:

$$C_j^a = \sum_{i=1}^{M} C_{ij} T_{ij}^{avail} C_i \tag{5.1}$$

where the available time T_{ij}^{avail} of P_i which can be utilized for computing load L_j is given as:

$$T_{ij}^{avail} = \max(D_j - T_i^F, 0) \tag{5.2}$$

where T_i^F is P_i's finish time of its previously allocated load units, which can be calculated according to prior load allocation.

To determine L_j's urgency, we shall consider both the available computation capacity before its deadline and the remaining amount of the load. We denote L_j^r as the total amount of computations required for processing all remaining units of L_j. We can write L_j^r as:

$$L_j^r = N_j^r W_j^u \tag{5.3}$$

where N_j^r is the number of remaining units of L_j. In MSCF we use "slack capacity" δ_j to determine L_j's urgency. We denote δ_j as:

$$\delta_j = C_j^a - L_j^r \tag{5.4}$$

Thus δ_j is defined as the amount of loads that can be processed before L_j on all L_j's eligible processors without violating L_j's deadline. This metric can reflect L_j's urgency by considering both available capacity before the deadline and the

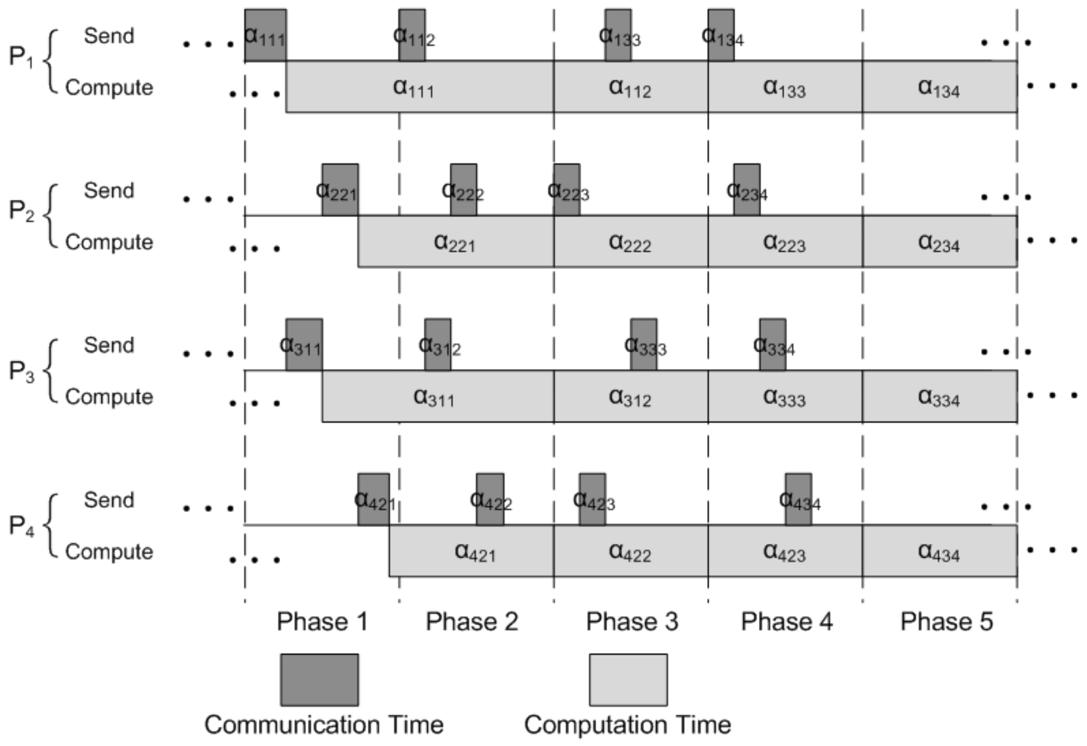


Figure 5.1: Timing diagram of AEP.

remaining load size. Hence, in MSCF the task with the smallest δ_j is first selected. If δ_j is negative, L_j cannot be successfully scheduled.

5.2.2 The AEP Load Distribution Strategy

The scheduling policies (EDF, LLF, and MSCF) only determine the scheduling order of tasks, but they do not determine how to distribute the selected task to processors. To address this dilemma in processor assignment, we propose two load distribution strategies, AEP and LCF. Both the strategies employ a "phase-based" multi-round approach and in this subsection we first introduce AEP. Fig. 5.1 shows the timing diagram of AEP for an example of 4 processors and 3 tasks in 4 rounds. The identifier α_{ijk} indicates the load chunk (in terms of number of load units) distributed to processor P_i from load L_j in round k. AEP works in an incremental and periodical fashion, executing multiple rounds to distribute and process loads. This process lasts for multiple phases and each phase is a fixed period of which the length is T_p. To achieve high performance, the basic idea behind AEP lies in implementing efficient pipelining. That is, the load chunks distributed in round k ($k = 1, 2, 3, \cdots$) are sent in phase k and computed in phase $(k + 1)$. In phase $(k + 1)$, while computing the chunks of round k, processors can simultaneously receive the chunks of round $(k + 1)$, as shown in Fig. 5.1. In this way, communications and computations are pipelined and overlapped. Also, the transmission of round k must be completed before phase $(k + 1)$ starts so that processors can start immediately to compute the chunks of round k in phase $(k+1)$ and P_0 can also start immediately to transmit the chunks of round $(k+1)$ in phase $(k + 1)$.

Suppose load L_j is selected for load distribution. Now we describe how P_0 distributes L_j to eligible processors in each round. The allocation process starts from round 1 and here we consider an arbitrary round k as an example. In round k, since P_0 attempts to dispatch L_j to all eligible processors, we shall first determine the eligible processors for L_j, which need to satisfy the following two criteria. Firstly, such processors are capable of processing L_j. Secondly, such processors have idle time in phase $(k + 1)$. If a processor is distributed an adequate amount of load to continually compute until the end of phase $(k + 1)$, the processor is categorized as having no idle time in phase $(k + 1)$; otherwise, it is classified as having idle time in phase $(k + 1)$.

As P_0 attempts to dispatch L_j to all eligible processors, it (sequentially) selects each eligible processor P_i to distribute it a load chunk. To distribute a chunk, P_0

first computes P_i's available compute time before phase $(k+1)$ ends. Then it can allocate a chunk according to the available time. We denote this time as T_{ik}^{comp}. To obtain T_{ik}^{comp} we shall consider the following three constraints. Firstly, P_i can process the new chunk between the time when it finishes previously allocated chunks and the deadline of L_j or the termination time of phase $(k+1)$. Secondly, even if P_i is idle, necessary communication time should be subtracted from the total available time. Thirdly, the communication of P_i should be finished before phase $(k+1)$ starts. Thus we can obtain T_{ik}^{comp} as follows:

$$T_{ik}^{comp} = \min(E_1, E_2, E_3) \tag{5.5}$$

where:

$$\begin{cases} E_1 = T_{end} - T_i^F \\ E_2 = \dfrac{T_{end} - T_k^{comm}}{1 + \frac{C_i S_j^u}{BW_j^u}} \\ E_3 = \dfrac{BW_j^u (T_0 + kT_p - T_k^{comm})}{C_i S_j^u} \end{cases} \tag{5.6}$$

where $T_{end} = \min(D_j, T_0 + T_p(k+1))$. This means if the deadline D_j occurs in phase $(k+1)$, P_i can process loads until D_j is met; otherwise P_i can work till the end of phase $(k+1)$. Here T_0 is the time instant when the new schedule is generated and the start time of phase k is $T_0 + T_p(k-1)$. Thus the compute time of round 1 can be greater than 1 phase when there is no prior load allocation ($T_i^F = 0$), as shown in Fig. 5.1. T_k^{comm} is the time instant when P_0 can start to send the load chunk to P_i and thus $(T_{end} - T_k^{comm})$ is the total available time when P_0 is allocating loads to P_i. Notice that all load chunks of different tasks of round k are sent in phase k according to the order in which they are scheduled (as shown in Fig. 5.1). Therefore, at the beginning of each round k, T_k^{comm} is initialized as $T_0 + T_p(k-1)$ and once a chunk of round k is allocated, the scheduler will update T_k^{comm} by adding the corresponding communication time of the chunk. As E_1, E_2, and E_3 in Equation (5.6) can be directly calculated, T_{ik}^{comp} can also be easily determined. Knowing T_{ik}^{comp}, we can calculate the chunk size α_{ijk} (in terms of number of units) allocated to P_i from L_j in round k. In reality, this number of units should be an integer. Thus we can write α_{ijk} as follows:

$$\alpha_{ijk} = \lfloor \frac{C_i T_{ik}^{comp}}{W_j^u} \rfloor \tag{5.7}$$

If N_j^r is less than the chunk size α_{ijk}, then α_{ijk} is accordingly reduced to that number and all unallocated units of L_j are distributed to P_i. Accordingly, P_i will still be available after being distributed the chunk since it still has some idle time before phase $(k+1)$ ends. In this case, P_i can be allocated another chunk in round k from remaining tasks and P_i shall execute these different chunks in phase $(k+1)$ following their scheduling order so that deadlines will not be violated.

P_0 iterates this load distribution process to distribute L_j in this round to other processors until all eligible processors are utilized in this round or communication time is used up ($T_k^{comm} = kT_p$). Then P_0 starts to distribute L_j in round $(k+1)$ and repeats these steps until the entire L_j is allocated. Afterwards P_0 can start to distribute next task selected by scheduling policies.

From Figure 5.1 we can observe that in phase $(k+1)$, while computing the chunks of round k, the processors could simultaneously receive the chunks of round $(k+1)$. In this way, communications and computations are pipelined and overlapped to save time. Prior works that addressed real-time issues in DLT literature [17, 18, 19] adopted a single-round approach; in these strategies processors need to wait for P_0 to transmit loads to other processors before they can start their own communication and computation. But in AEP such idle-time waste is significantly minimized by efficient overlapping. In addition, AEP offers efficient resource utilization since processors can continuously work throughout consecutive phases once processors become available. However, in [17] it is assumed that all participating processors assigned to a given task are released simultaneously and thus a processor can start to compute the task only after all the participating processors become available, which can cause severe wastage of compute power.

5.2.3 The LCF Load Distribution Strategy

The AEP strategy does not consider the contention of various loads which share the same compute resources but have various processing requirements. This leaves some room for performance optimization. Therefore, we revisit AEP to design an alternative load distribution strategy, LCF, which selects the processor with the least capability first. The capability parameter θ_i of processor P_i is defined as the total amount of all remaining loads that can be processed by P_i. We denote θ_i as follows:

$$\theta_i = \sum_{j \in S} C_{ij} W_j^u N_j^r \tag{5.8}$$

where S is a set containing all loads to be scheduled. In LCF, P_0 first selects the processor with the smallest θ_i among all eligible processors of the selected load L_j. For a selected processor P_i, P_0 starts from round 1 and sequentially distributes load chunks from L_j to P_i in multiple rounds according to Equations (5.5), (5.6), and (5.7) until L_j's deadline. Then P_0 iterates to select other processors and to distribute chunks to the processors until the entire task is allocated. Afterwards P_0 can start to distribute next task selected by scheduling policies.

Below we compare the benefits and the drawbacks of AEP and LCF. LCF handles the contention of tasks by first selecting the processors with the least θ_is and leaving the processors with high θ_is to remaining tasks. Thus LCF guarantees high resource utilization as the processors with high θ_is are likely to be utilized by remaining loads. On the contrary, in AEP it is possible that the processors with low θ_is are left to remaining tasks, and such processors may not be utilized due to their restricted capabilities. This leads to low resource utilization and may finally cause some tasks to be rejected. Therefore, for the current set of tasks, LCF can optimize processor utilization.

On the other hand, although LCF can efficiently utilize processors for the current set of tasks, yet it may not achieve high resource utilization for future tasks once contention is not intensive. For example, if there are only a few tasks and the tasks' deadlines are not urgent, some processors will remain idle since LCF tends to employ the minimum number of processors to meet tasks' deadlines. This leads to low processor utilization and may finally cause some future tasks to be rejected since current tasks are finished late. Therefore, when contention is not intensive (e.g., deadlines are not urgent), it is beneficial to use AEP which employs all eligible processors so that current tasks can be completed as soon as possible and the processors can be released timely for computing unexpected future tasks. The time complexity of AEP for scheduling J tasks is $O(J^2)$ and the complexity of LCF is $O(J^2 + JM)$ as LCF needs additional efforts in processor selection.

5.2.4 Scheduling Architecture of RARTS

When a new task arrives at time T_0, the scheduler runs an admissibility test to dynamically determine whether it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. To run the admissibility test, the scheduler generates a task list containing the new task and previously admitted tasks. The scheduler then executes a scheduling policy (EDF or MSCF) to determine task scheduling orders and a load distribution strategy (AEP or LCF) to distribute selected tasks to processors. Upon completion of the test, if all tasks can be completed within their respective deadlines, a new schedule is developed to replace the existing schedule and the new task is accepted. Otherwise, the new task is rejected and the scheduler maintains the existing schedule.

To generate a new schedule, P_0 will reset system parameters. The new task and remaining unsent loads are considered together for scheduling. In the new schedule, P_0 starts round 1 from T_0 and thus the start time of each phase k is $T_0 + T_p(k-1)$. Processors' finish times T_i^Fs are updated according to the finish time of the chunks which have been sent before the arrival of the new task.

5.3 Performance Evaluation

In this section, we evaluate the performance of RARTS strategies by rigorous simulation experiments. Our interested metric is *acceptance ratio*, which is defined as the ratio of the number of tasks found schedulable to the number of tasks arriving for scheduling. With the aforementioned scheduling polices and load distribution strategies, we can generate 5 algorithms, EDF-AEP, LLF-AEP, MSCF-AEP, EDF-LCF, and MSCF-LCF. The nomenclature of the algorithms includes two parts. The first part represents the scheduling policy adopted: EDF, LLF, or MSCF. The second part indicates the load allocation strategies adopted: AEP or LCF. We only combine LLF with AEP because when LCF is applied, the slack time of LLF would be zero for a load since its finish time is equal to its deadline.

To understand the merits of our algorithms, we shall compare them with algorithms proposed in prior works. Among the strategies proposed in prior works on real-time divisible load scheduling [17, 18, 20, 19], we found that the strategies in [17, 18, 19], can be used for comparison because [20] focus on factors irrelevant to our work. Although the strategies in[17, 18, 19] are designed for homogeneous computing platforms and homogeneous divisible loads, yet we can modify them so that they can also work on heterogeneous computing platforms and handle heterogeneous loads. Since the strategy in [19] outperforms the strategies in [18] (as reported in [19]), and the strategies in [18] outperform the strategies in [17] (as reported in [18]), we choose the strategy proposed in [19] for comparison. This strategy, extended from works in [18], adopts the same basic mechanism as [18]. That is, the strategy attempts to schedule loads to the minimum number of processors in a single round of workload allocation so that the processors can finish computing at the deadline. The modification we made for this strategy is that when determining the minimum number of processors for a selected load, only the processors which are capable of processing the load are considered. Below we denote this modified single-round strategy as "SRS".

The simulated system consists of randomly generated processing nodes. The initial simulation configurations are set as follows: The number of processors is 20. For each processor P_i the computation power parameter C_i is uniformly distributed among $[100, 500]$ and the communication speed B is set as 100. To generate a set of real-time divisible loads, we assume that the inter-arrival times follow an exponential distribution with a mean of $1/\lambda$ and we initially set $1/\lambda$ as 10 time units. A task's relative deadline (defined as $D_j - A_j$) is assumed to be uniformly distributed among $[T_j^{sc}*(1+\frac{1}{2}R), T_j^{sc}*(1+\frac{3}{2}R)]$ where T_j^{sc} is the estimated shortest compute time for load L_j ($T_j^{sc} = \frac{L_j}{\sum_{i=1}^{M} C_{ij}C_i}$) and R is the laxity parameter which denotes the tightness of the deadlines. In addition, to study the effect of different

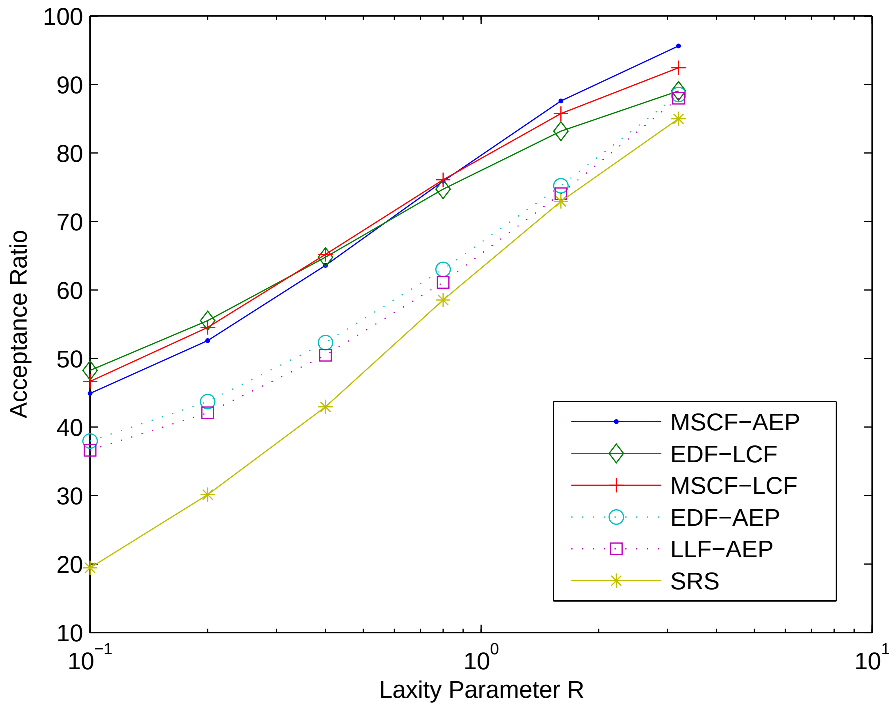


Figure 5.2: Average acceptance ratio versus R.

load types and sizes, we let S_j^u be uniformly distributed among $[0.01, 0.05]$, let W_j^u be uniformly distributed among $[1, 5]$, and let N_j be uniformly distributed among $[10,000, 30,000]$. The length of each phase T_p is set to be 1 time unit. To study the influence of multiple loads with heterogeneous requirements, we use parameter p_j to denote the probability that a load L_j can be processed by any processor. Thus for a given load L_j, p_j is a constant, but for different loads, this parameter can be different. We initially let p_j be uniformly distributed among [0.1, 0.9].

In the following experiments we vary our interested parameters while fixing other parameters as their initial values to study the effect of the interested parameters. We first evaluate the influence of the laxity parameter R on our algorithms. To simulate different deadlines we vary R among the range [0.1, 3.2] and Fig. 5.2 depicts the acceptance ratio versus R. In addition, we vary $1/\lambda$ from 2 to 20 and

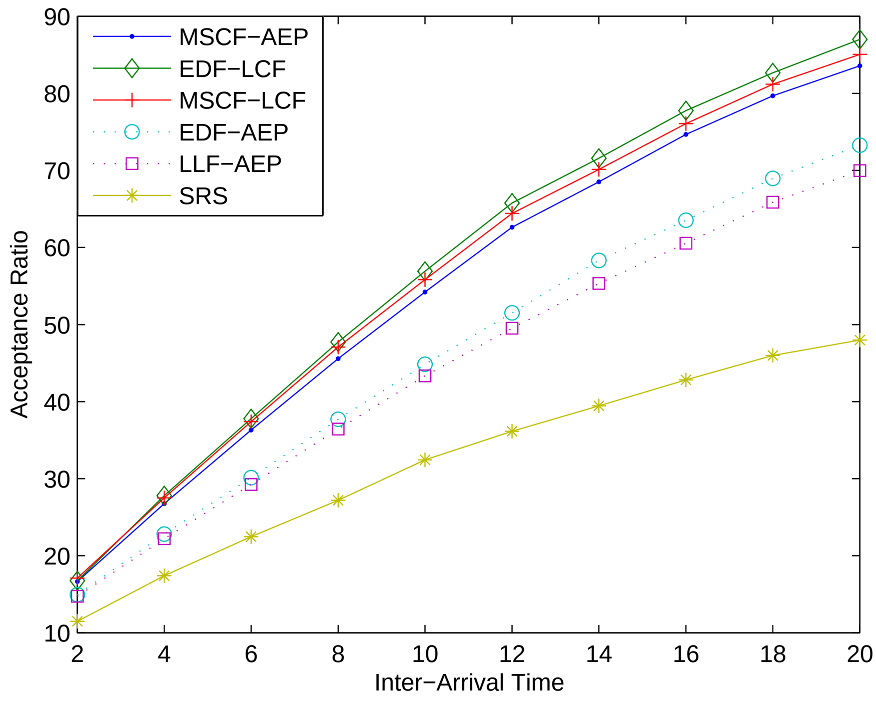


Figure 5.3: Average acceptance ratio versus $1/\lambda$ ($R = 0.2$).

Fig. 5.3 and 5.4 show the acceptance ratio versus $1/\lambda$ for $R = 0.2$ and 1, respectively. To study the effect of p_js, we let p_js be uniformly distributed among $[p_a - 0.2, p_a + 0.2]$ and we vary p_a among $[0.3, 0.8]$, and the corresponding acceptance ratio is shown in Fig. 5.5 and 5.6 for $R = 0.2$ and 1, respectively. Moreover, we vary B among $[20, 200]$ and the corresponding acceptance ratio is shown in Fig. 5.7 and 5.8 for $R = 0.2$ and 1, respectively. Finally, we vary T_p among the range $[1/32, 16]$ and the corresponding acceptance ratio is shown in Fig. 5.9 and 5.10 for $R = 0.2$ and 1, respectively.

The simulation results show that the RARTS algorithms outperform SRS in most cases. This may be due to 3 plausible reasons. Firstly, RARTS algorithms distribute loads in multiple rounds which can efficiently overlap communications with computations to save time, while SRS dispatches loads in a single round

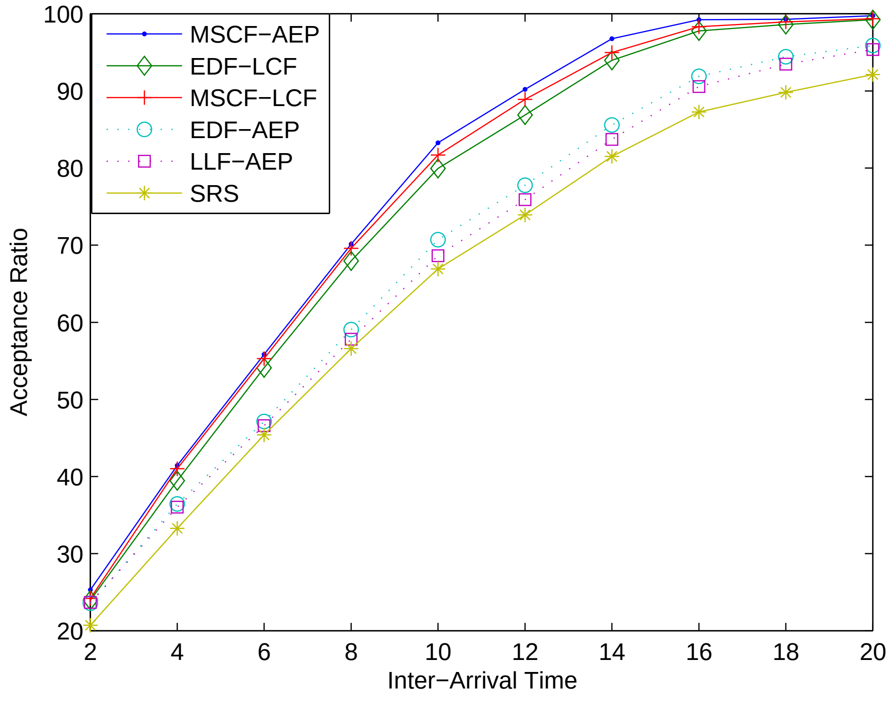


Figure 5.4: Average acceptance ratio versus $1/\lambda$ ($R = 1$).

of allocation and hence consumes more communication time than RARTS. This explains the fact that EDF-AEP delivers better performance than SRS in most cases. Secondly, SRS does not take measures to handle different requirements of various tasks. Thirdly, since SRS tends to employ the minimum number of processors, some nodes will remain idle when deadlines are not urgent. This leads to low processor utilization and may finally cause some future loads to be rejected.

Fig. 5.2 shows that MSCF-AEP, EDF-LCF, and MSCF-LCF significantly outperform EDF-AEP, LLF-AEP and SRS for different R values since MSCF-AEP, EDF-LCF and MSCF-LCF utilize either MSCF or LCF to handle heterogeneous tasks. In addition, we observe that when R is small, EDF-LCF, and MSCF-LCF outperform MSCF-AEP because when the contention among tasks is intensive LCF can optimize resource utilization by leaving the processors with high capa-

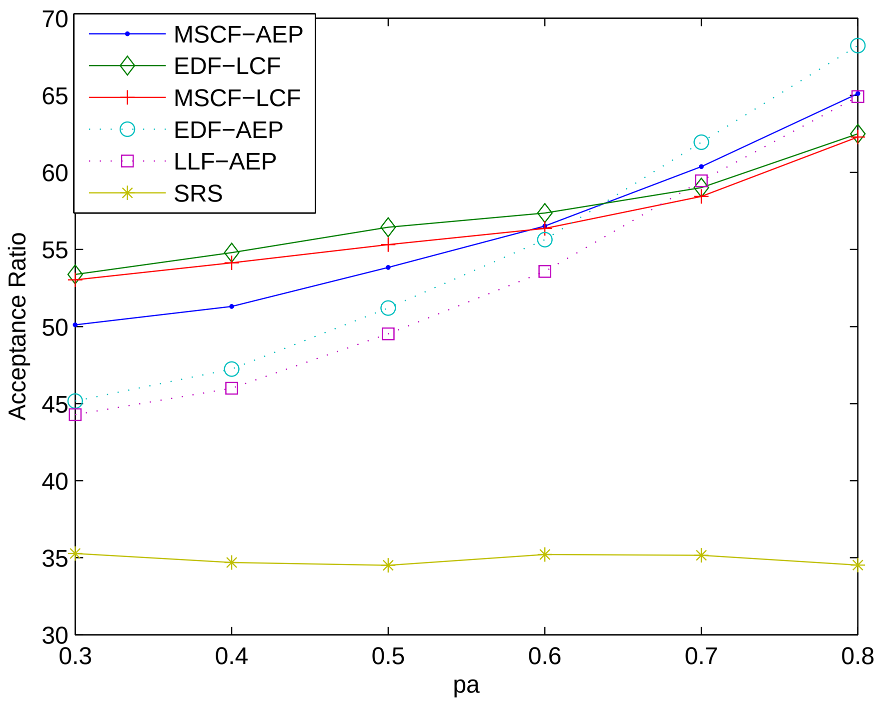


Figure 5.5: Average acceptance ratio versus p_a $(R = 0.2)$.

bilities to remaining loads. But for AEP it is possible that the processors with low capabilities are left to remaining loads and such processors may not be used due to their limited capabilities. On the other hand, as R increases, MSCF-AEP gradually exhibits its efficiency and delivers better performance than EDF-LCF and MSCF-LCF. A plausible explanation is that since LCF employs a minimum number of nodes to meet loads' deadline, some nodes will remain idle when deadlines are not urgent. This leads to low resource utilization and may finally cause some future loads to be rejected. Therefore, when deadlines are not urgent, it is beneficial to use AEP which employs all processors so that current tasks can be completed soon and the processors can be released timely for computing future loads. Moreover, we observe that when R is small EDF-AEP significantly outperforms SRS but their performance gap decreases as R grows. This is due to the fact

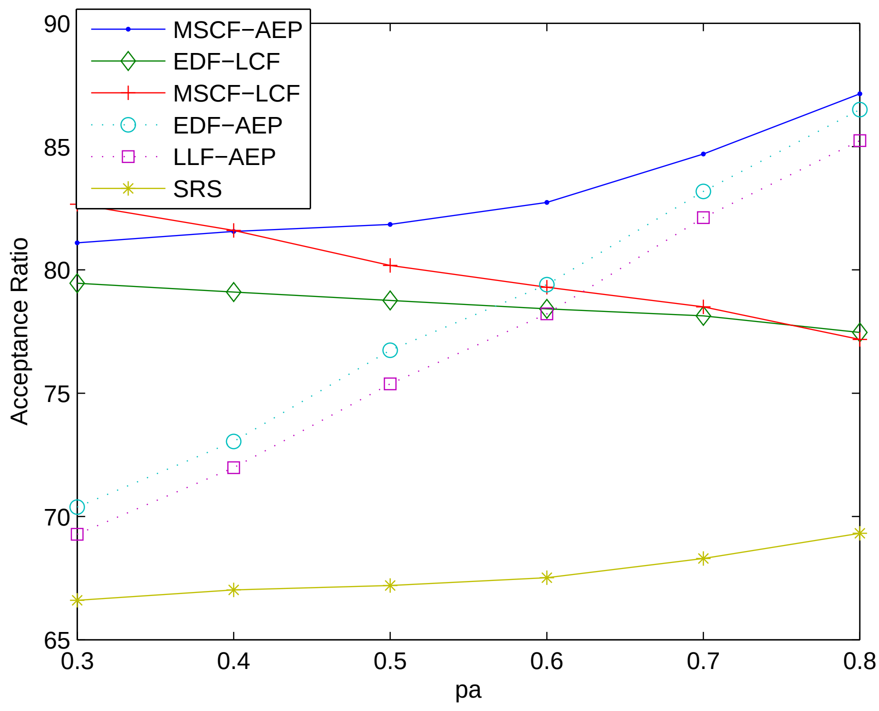


Figure 5.6: Average acceptance ratio versus p_a $(R = 1)$.

that the communication time is more critical to acceptance ratio when deadlines are urgent.

Fig. 5.3 and 5.4 show that as the inter-arrival time $1/\lambda$ increases the load acceptance ratio of each experiment set increases since reducing the loads arrived in a period of time can decrease the contention on the computing resources and hence increase the acceptance ratio. In addition, we can observe that no matter how $1/\lambda$ varies, when R is 0.2 EDF-LCF outperforms MSCF-LCF and MSCF-AEP; when R is 1 MSCF-AEP delivers better performance than EDF-LCF and MSCF-LCF. These observations can be explained by the above analysis.

From Fig. 5.5 and 5.6 capture the influence of p_a, which signifies the capability of a node. We can observe that when p_a is small and deadlines are urgent EDF-LCF and MSCF-LCF deliver better performance than MSCF-AEP. On the other hand,

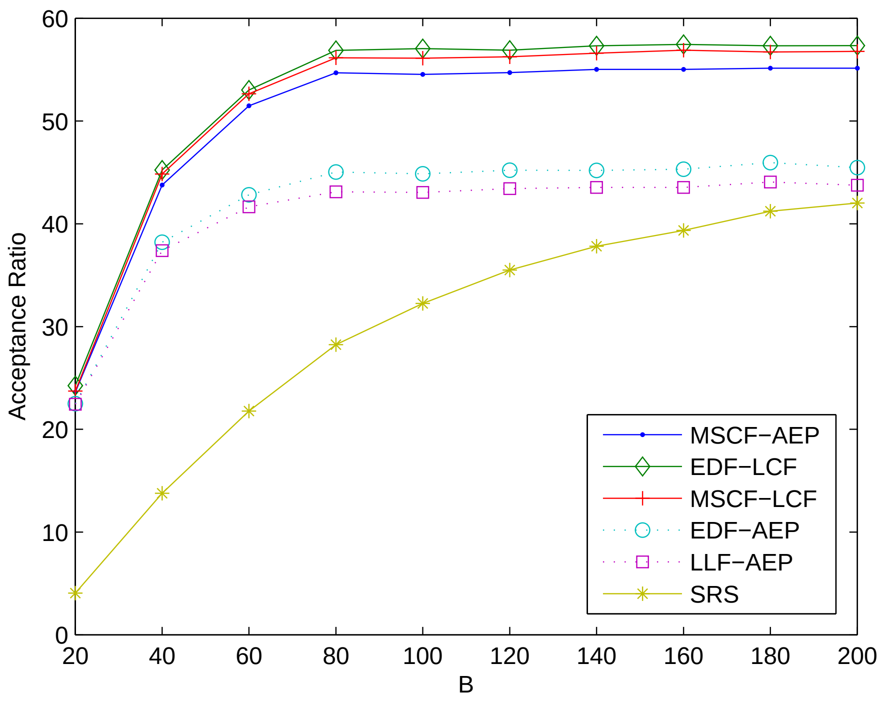


Figure 5.7: Average acceptance ratio versus B ($R = 0.2$).

when p_a is large and deadlines are not urgent, MSCF-AEP outperforms EDF-LCF and MSCF-LCF. Moreover, we can observe that EDF-AEP and LLF-AEP are more sensitive than MSCF-AEP, EDF-LCF and MSCF-LCF to the fluctuation of p_a. This is due to the fact that MSCF-AEP, EDF-LCF and MSCF-LCF utilize either MSCF or LCF to handle heterogeneous loads. In this case, when p_a is small MSCF-AEP, EDF-LCF and MSCF-LCF significantly outperform EDF-AEP and LLF-AEP since the contention among heterogeneous loads is more crucial when p_a is small. On the other hand, when p_a is large, EDF-AEP and LLF-AEP become more competitive. This is because when p_a is large, a task can be processed by most processors and hence the contention among heterogeneous loads becomes slight, which benefits EDF-AEP and LLF-AEP.

Fig. 5.7 and 5.8 show that as communication speed parameter B increases the

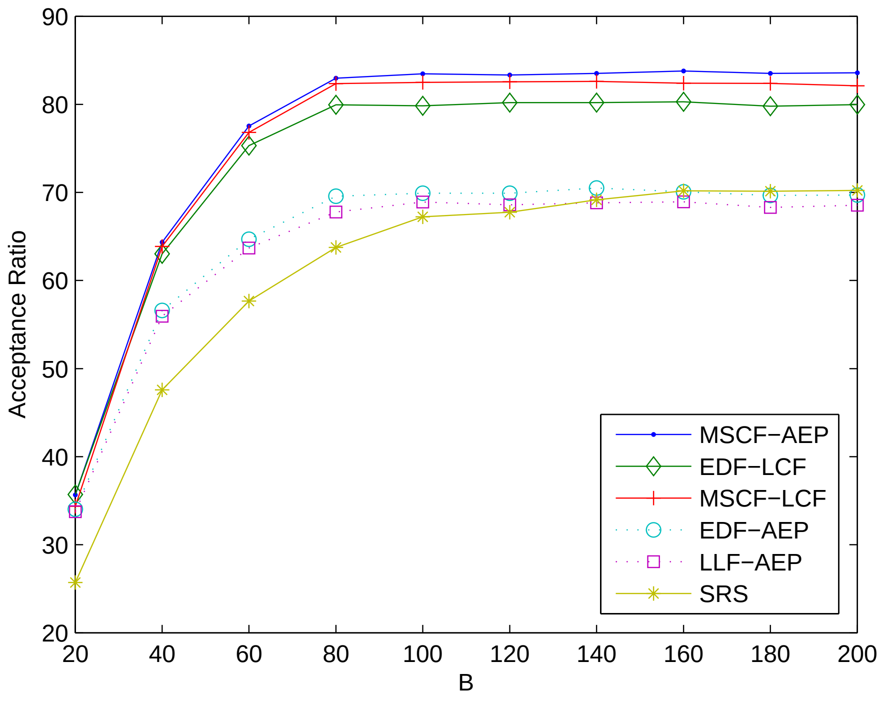


Figure 5.8: Average acceptance ratio versus B $(R = 1)$.

load acceptance ratio of each experiment set increases since decreasing communication time reduces the processing time of the loads. Fig. 5.7 shows that SRS is more sensitive than others to the variation of B because SRS is a single-round approach and hence consumes more communication time. Also, Fig. 5.8 shows that SRS becomes less sensitive to the variation of B due to slack deadlines. Further, the acceptance ratios of our strategies stabilize when B is large. This is because when B is large enough varying B impacts little on the processing time.

Fig. 5.9 and 5.10 show that when T_p is large the acceptance ratios of MSCF-AEP, EDF-LCF and MSCF-LCF decrease as T_p increases. This is because if T_p is large, processors may not become available soon since their finish time increases as T_p becomes large. In addition, when T_p is less than 0.125 the acceptance ratios of the 5 proposed algorithms sharply reduce as T_p decreases. This is due to the

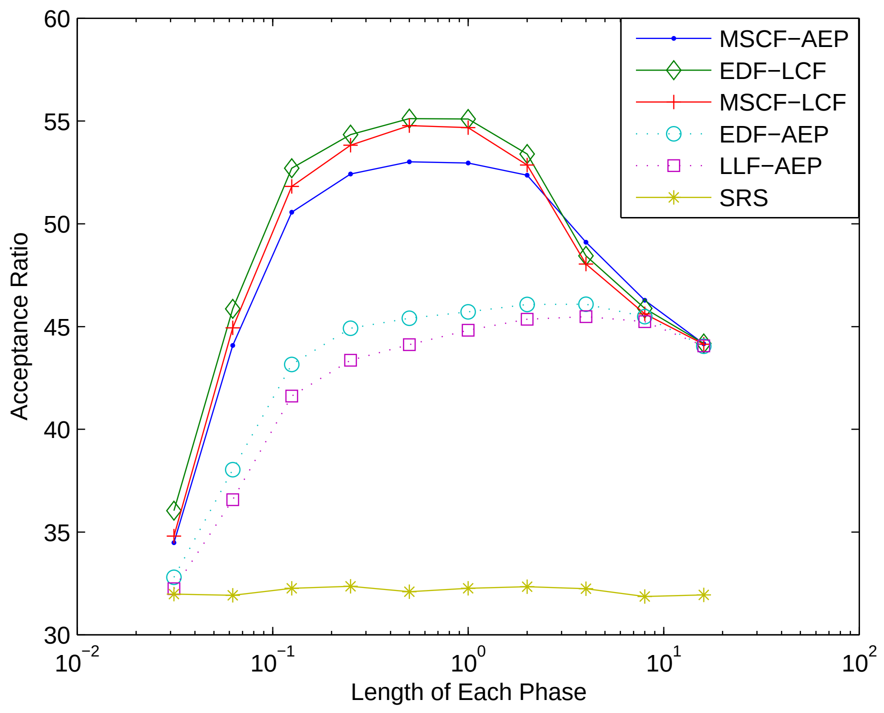


Figure 5.9: Average acceptance ratio versus T_p ($R = 0.2$).

fact that if T_p is too small probably some processors cannot receive a unit as they cannot finish 1 unit in 1 phase. Further, the performance of MSCF-AEP, EDF-LCF and MSCF-LCF stabilizes when T_p is among $[0.125, 2]$. This shows that although T_p cannot be too small or too large, there is still a wide range in which varying T_p only slightly influences the performance. This range is related to the granularity and the sizes of the loads. As the proposed algorithms are designed for processing divisible loads consisting of many low-granularity computations, choosing satisfactory values of T_p may not be challenging. When implementing RARTS for real-world applications, simulations or experiments can help determine satisfactory values of T_p once the granularity and size of the data are given.

The simulation results have shown that MSCF-AEP, EDF-LCF, and MSCF-LCF are more effective than EDF-AEP, LLF-AEP and SRS in most cases. Lessons

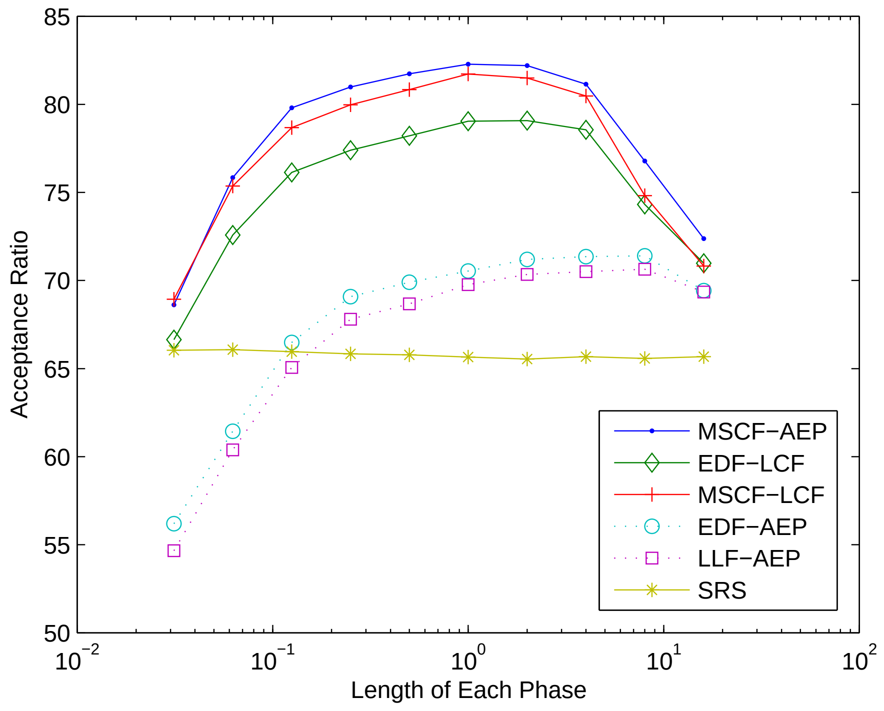


Figure 5.10: Average acceptance ratio versus T_p ($R = 1$).

learnt from this experience seem to offer a set of recommendations on the choice of scheduling policies and load allocation strategies. In a realistic system, the scheduler can first use MSCF-AEP to handle new tasks since AEP is beneficial for admitting future tasks. Once MSCF-AEP cannot admit a new task, the scheduler can use EDF-LCF/MSCF-LCF as they outperform MSCF-AEP when the contention among tasks are intensive.

Chapter 6

Scheduling Hybrid Loads

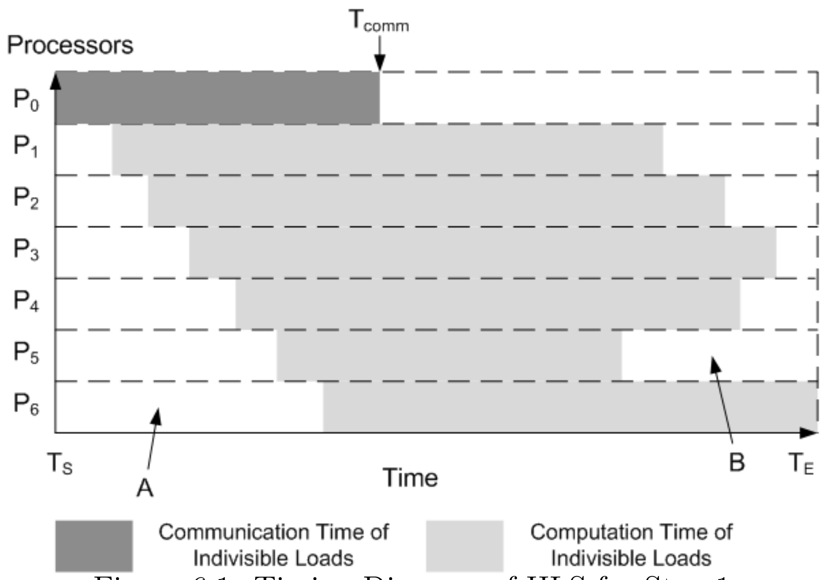
In this chapter we investigate the scheduling problem for processing a set of tasks comprising both divisible and indivisible tasks on cluster systems. Indivisible loads are characterized by the property that they need to be processed on their entirety on a single processor while divisible loads can be distributed across several processing nodes by exploiting the underlying data parallelism. We propose a novel scheduling algorithm referred to as *Hybrid Load Scheduling* (HLS) algorithm for handling a set of tasks comprising both divisible and indivisible tasks on clusters. HLS fully exploits the parallelism in divisible loads to achieve high resource utilization. It again applies the pipelining technique to hide communication time. However, differing from the phase-based algorithms presented in Chapters 4 and 5, in HLS the schedule begins with the last phase and the communication follows a "latest phase first" rule. These designs are made to fit the context of scheduling hybrid tasks.

6.1 Problem Formulation

We consider scheduling J load applications $(L_1, L_2, ..., L_J)$ onto M processors. The objective is to minimize the total processing time (i.e., makespan) of all the loads. We let S_j be the size (in bytes) of L_j and W_j be the workload of L_j, i.e., the amount of computations (e.g., floating point operations) required to compute L_j. Let CI_j be the computation intensity value of L_j and $CI_j = W_j/S_j$. Following prior works [71, 72], we assume that the workloads of all tasks (W_js) are known a priori thanks to prediction mechanisms such as code profiling and statistical prediction.

As we consider hybrid tasks, the loads can be either indivisible or divisible. The divisible loads are a typically embarrassingly parallel type of applications, which consist of large numbers of identical, independent, and low-granularity data units which can be distributed to networked computers for parallel processing. Following the model in [2, 11], we partition each divisible load into a large number of identical, independent, and low-granularity load units and let N_j be the total number of units of load L_j. Also, we let S_j^u be the size (in bytes) of a load unit of L_j and let W_j^u be the workload of a unit of L_j, which is defined as the amount of computations required to compute a unit of L_j. Thus for a divisible load L_j, $S_j = S_j^u * N_j$ and $W_j = W_j^u * N_j$. In this paper, the start-up time overheads for divisible loads are considered to be negligible and we adopt a linear and fixed communication and computation model, which is widely used in many prior DLT papers including [2, 3, 4]. In this linear model, it takes $\frac{XS_j^u}{B}$ time units to send X units of load L_j from P_0 to each processor P_i and it takes $\frac{XW_j^u}{C_i}$ time units to compute X units of load L_j on P_i.

The indivisible loads are independent tasks without inter-task communications or dependencies. Each indivisible load can only be executed on one processor.


T_comm
Processors
P0
P1
P2
P3
P4
P5
P6
TS
A
Time
B
TE
Communication Time of Indivisible Loads
Computation Time of Indivisible Loads
Figure 6.1: Timing Diagram of HLS for Step 1.

Accordingly, if a load L_j is indivisible, then $N_j = 1$. For each indivisible load L_j, it takes $\frac{S_j}{B}$ time units to send L_j from P_0 to each processor P_i and it takes $\frac{W_j}{C_i}$ time units to compute L_j on P_i.

6.2 Proposed HLS Algorithm

The HLS algorithm consists of 2 steps. Step 1 first schedules indivisible loads s-ince they are inflexible and cannot be partitioned for parallel computing. Step 2 then schedules divisible loads. To schedule indivisible loads, the Max-Min policy described below can be used. Max-Min first schedules the task whose earliest completion time over all processors is the longest among all unscheduled tasks. The selected task is then allocated to the processor on which the earliest completion time can be achieved. After all indivisible loads are allocated, a partial schedule containing the indivisible loads is generated. As shown in Fig. 6.1, the partial schedule starts at time instant T_S and terminates at T_E. Also, the load transmission of the indivisible loads is finished at T_{comm}. We can observe that between T_S

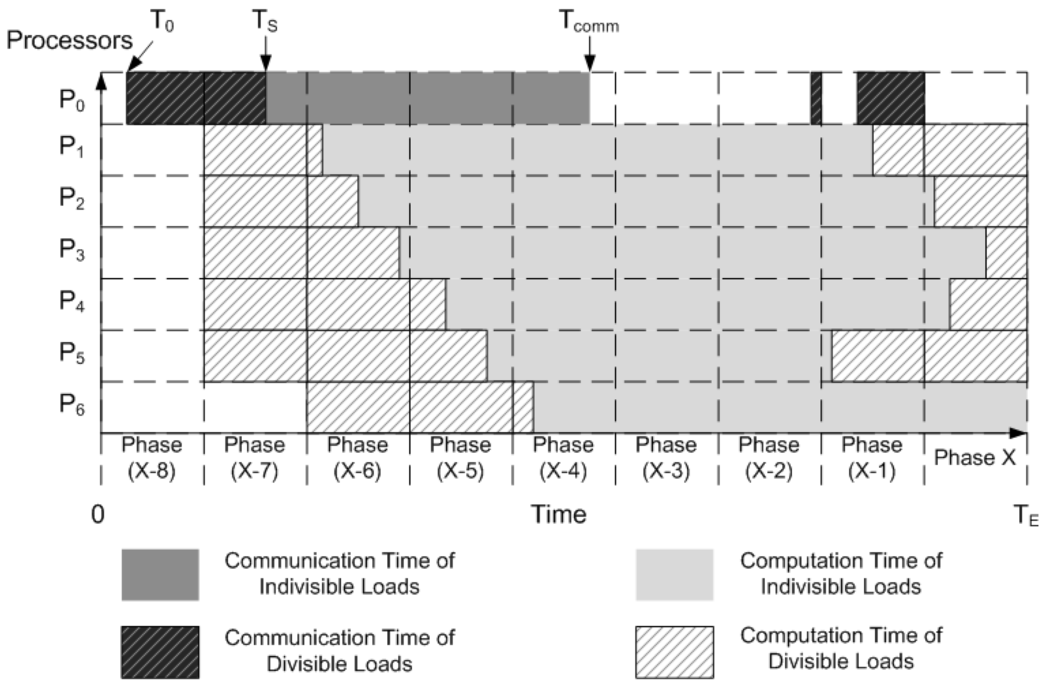


Figure 6.2: Timing Diagram of HLS for Step 2.

and T_E there is considerable idle processor time, which can be exploited to process divisible loads. The idle time results from two cases. Firstly, processors need to wait for P_0 (thus, wasting time) to transmit loads to other processors before they can start their own communication and computation (Area A in Fig. 6.1). Secondly, processors may finish computing at different times. The processors that finish earlier than others will remain idle and this leads to considerable wastage of processor time (Area B in Fig. 6.1). Accordingly, these leave large space for optimization. Since scheduling divisible loads between T_S and T_E does not increase the makespan, we aim to first utilize such idle time to schedule divisible loads.

To enhance the utilization of the idle time, communication time should be efficiently hidden. Therefore, in Step 2 we adopt a pipelining technique in scheduling divisible loads. Fig. 6.2 shows the timing diagram of HLS for an example of scheduling both divisible and indivisible loads. HLS works in a periodic fashion, processing divisible loads in multiple phases. The duration of each phase is

T_p, which is predetermined. Let T_E be the end of phase X and T_S be in phase $(X - Y)$. Hence the partial schedule spans $(Y + 1)$ phases. While Y is known, X can be determined at the end of the scheduling. Also, we denote the computation time of a processor P_i in phase k ($k = X - Y, X - Y + 1, \cdots, X$) as a slice $Slice_{ik}$. If a slice is not fully utilized in computing indivisible loads, it is categorized as an idle slice and can be exploited to compute divisible loads. As all idle slices are known, we can utilize the idle time by filling divisible loads in the idle slices in multiple phases. We use the term "load chunk" to denote a number of load units executed in each idle slice. To achieve high performance, we again apply the pipelining technique in designing HLS. That is, the load chunks executed in the slices of phase k are sent before phase k. The chunks of phase k are transmitted prior than the chunks of phase $(k + 1)$. While computing the chunks in phase k, processors can simultaneously receive the chunks of future phases. In this way, communication time can be effectively hidden.

We can observe that to utilize idle time slices in Area A in Fig. 6.1 we need additional time for communication before T_S since the communication time between T_S and T_{comm} has been used up. But there exists available communication time between T_{comm} and T_E, which means the idle slices in Area B may be utilized without additional communication time. Hence, to optimize performance we shall start to exploit idle slices from Area B. This means the scheduler shall first determine the load distribution of the last phases. In other words, the scheduler should schedule divisible loads to the idle slices of multiple phases according to the descending order of the phases. In this case, the scheduling of divisible loads begins with the latest phase X and reverses to phase $(X - Y)$ to generate a complete schedule. Notice that if there are abundant divisible loads, the schedule of divisible loads can even be out of this range (between phase $(X - Y)$ and phase X). In each phase, the scheduler attempts to utilize all idle slices and allocate chunks to the slices. Suppose that an idle slice $Slice_{ik}$ is (sequentially) selected. The idle time of $Slice_{ik}$ is T_{ik}^{comp}. The scheduler then selects a divisible load for allocation.

To efficiently utilize the available communication time between T_{comm} and T_E, the scheduler selects the divisible load L_j with the lowest computation intensity value (CI_j) because this load requires much communication time and can effectively utilize the available communication time between T_{comm} and T_E. The scheduler then allocates a chunk from load L_j to the selected idle slice. One can calculate the chunk size α_{ijk} (in terms of number of units) allocated to $Slice_{ik}$ from L_j as follows:

$$\alpha_{ijk} = \lfloor \frac{C_i T_{ik}^{comp}}{W_j^u} \rfloor \tag{6.1}$$

Now we consider the communication (or transmission) of each load chunk. For the convenience of scheduling, the transmission of the divisible loads processed in phase k should be finished before phase k. It is preferable to send the chunk in the latest phase before the computation of the chunk. This helps to efficiently utilize communication time because the communication time in a phase k can only be utilized by load chunks computed in phase $(k + 1)$ or later phases. Since the computation time of different phases is allocated in reverse order, if the communication time of phase k is not fully utilized, later load distributions which are scheduled for prior phases may not be able to utilize such communication time. In this case, the policy for allocating communication time follows a "latest phase first" rule. For a given chunk executed in phase k, the scheduler first attempts to allocate enough communication time from phase $(k - 1)$. If there is no enough communication time in phase $(k - 1)$, the scheduler allocates all available communication time in phase $(k-1)$ to the chunk and attempts to allocate communication time from the second latest phase $(k-2)$. This process repeats until enough communication time is allocated to the chunk.

The scheduler iterates this load distribution process to distribute loads to all idle slices in phase k. Once there is no idle slice in phase k, the scheduler starts to distribute loads to the idle slices of phase $(k-1)$. In this way, the scheduler can distribute all loads to the idle slices of multiple phases. If there are abundant divisible loads, some loads will be scheduled to slices before phase $(X-Y)$ and accordingly the start time of the complete schedule will be before phase $(X-Y)$. In this case, k can be less than $(X-Y)$. After all divisible loads are scheduled, the overall start time T_0 and the total processing time can be determined. Suppose T_0 is in phase $(X-Z)$. Let $X = Z+1$. Accordingly, T_0 is in phase 1 and one can obtain X because Y and Z are known after all load distributions are determined. In the example shown in Fig. 6.2, X is 9 and the complete schedule spans 9 phases.

Finally, we conclude the mechanisms and benefits of HLS. Although the scheduling of indivisible loads causes idle processor time, HLS can flexibly schedule divisible loads to different slices so that the idle time can be easily utilized to optimize processing time. Also, HLS exploits pipelining to overlap communication with computation, which can further improve the resource utilization. The time complexity of HLS for scheduling J_d divisible loads and J_i indivisible loads is $O(J_i^2 + J_iM + J_d^2)$.

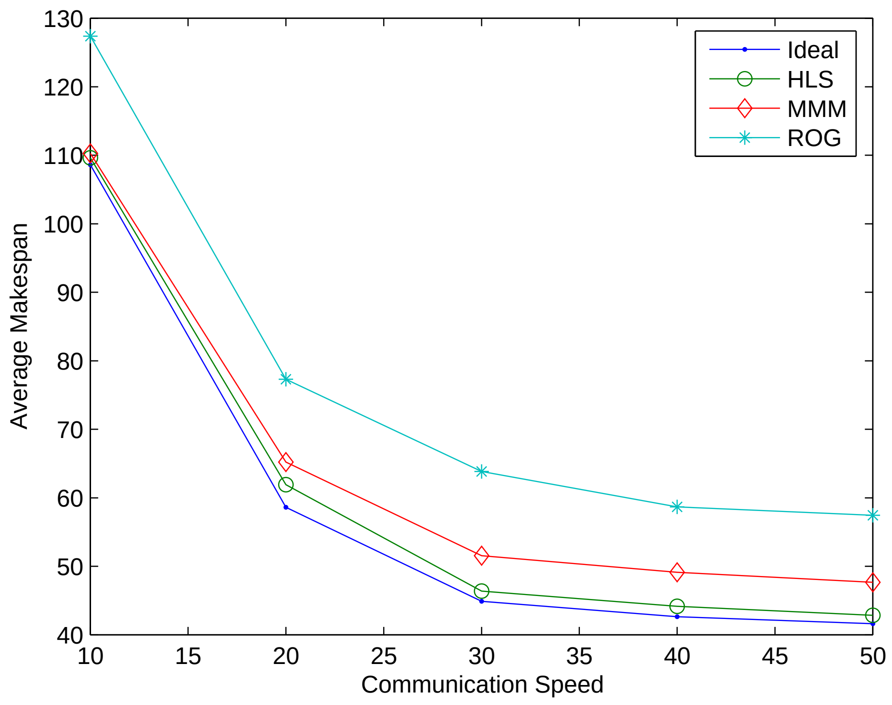


Figure 6.3: Average total processing time versus B.

6.3 Performance Evaluation

In this section, we evaluate the performance of HLS strategies by rigorous simulation experiments. As this paper is the first of its kind to consider scheduling mixed divisible and indivisible loads, there are no strategies available in the literature to compare with HLS directly. To understand the merits of our strategy, we design 2 simple algorithms, Modified Max-Min (MMM) and Random Order Greedy (ROG), to compare with the proposed algorithm. In Both MMM and ROG, each divisible load is divided into 10 equal pieces and each piece is treated as a new indivisible load, which can be scheduled independently. Then MMM utilizes Max-Min to schedule all tasks and ROG randomly selects a task and schedules the task to the processor on which the finish time can be minimized. In

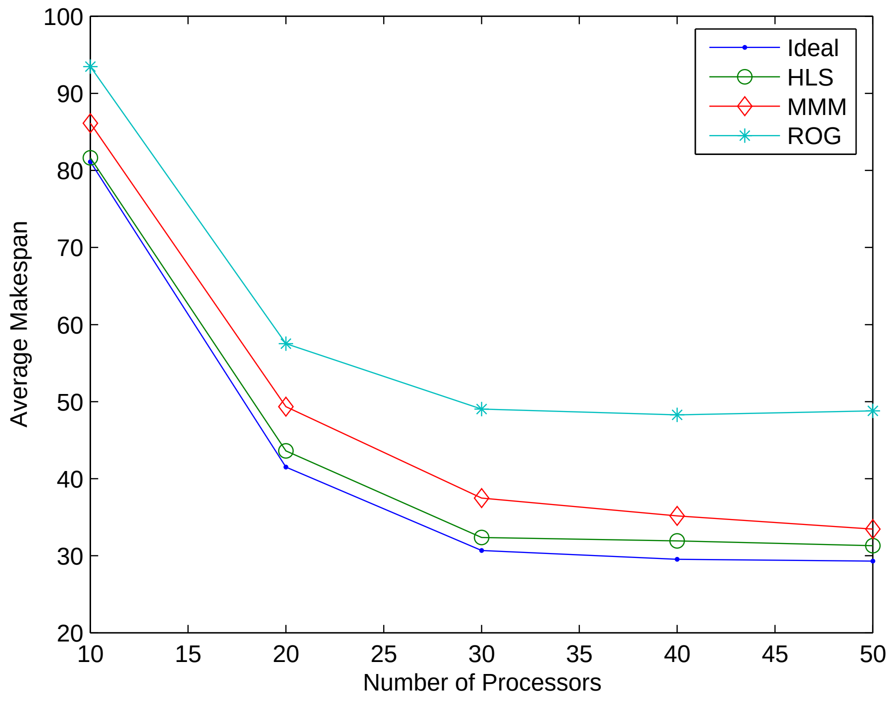


Figure 6.4: Average total processing time versus M.

addition, we compare the above 3 algorithms with an ideal case, in which all loads are assumed to be arbitrarily divisible. Also, in the ideal case communication can be "optimally" hidden by computation such that the communication time is zero. However, if the network bandwidth is too small, communication time may not absolutely hidden even if an "optimal" scheduling policy is applied. Hence we recognize a performance bound for the communication of the ideal case. That is, in the ideal case the total processing time of X units of load L_j cannot be less than the minimum communication time for transmitting such a load chunk, which is $\frac{XS_j^u}{B}$ seconds. If the calculated computation time of the ideal case is less than its minimum communication time, we let the overall processing time of the ideal case be the minimum communication time. Therefore the ideal case presents the performance bound among all possible solutions since the resource utilization of

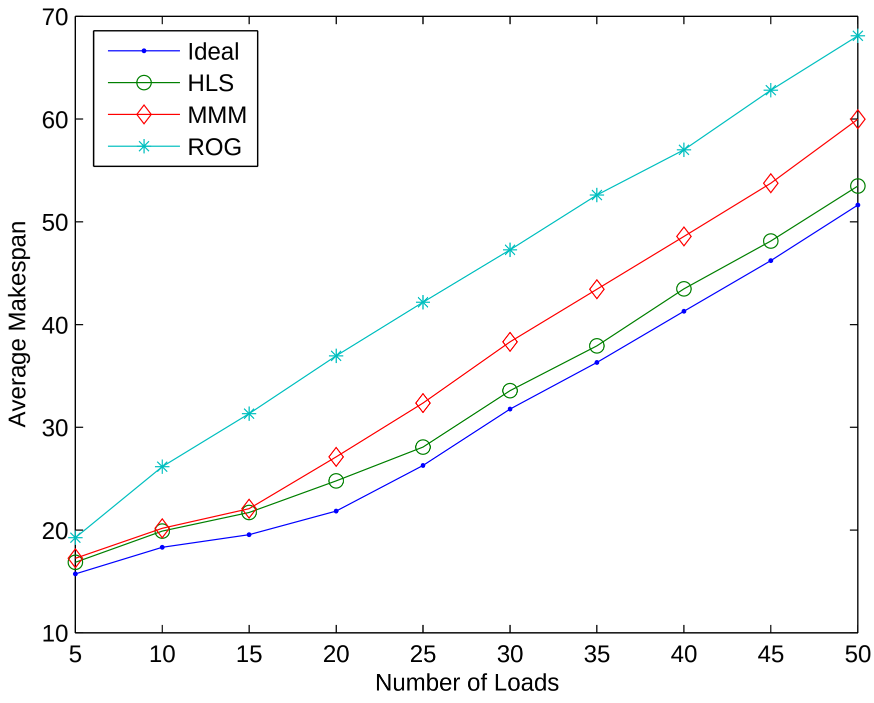


Figure 6.5: Average total processing time versus J.

either computation or communication is 100% in this case.

The simulated system consists of randomly generated processing nodes. The initial simulation configurations are set as follows: The number of processors M is 20 and the number of loads J is 40. The computation speed C_i is uniformly distributed among $[1, 5]$ and the communication speed B is set as 40. In addition, we use parameter p to denote the probability that a load L_j is indivisible. We initially set p as 0.5. Further, to study the effect of different load types and sizes, we let W_j be uniformly distributed among $[20, 100]$ and let CI_j be uniformly distributed among $[0.1, 10]$. Also, we let N_j of divisible loads be uniformly distributed among $[1,000, 10,000]$. Finally, the length of each phase T_p is set to be 1 time unit.

In the following experiments we vary our interested parameters while fixing other parameters as their initial values to study the effect of the interested pa-

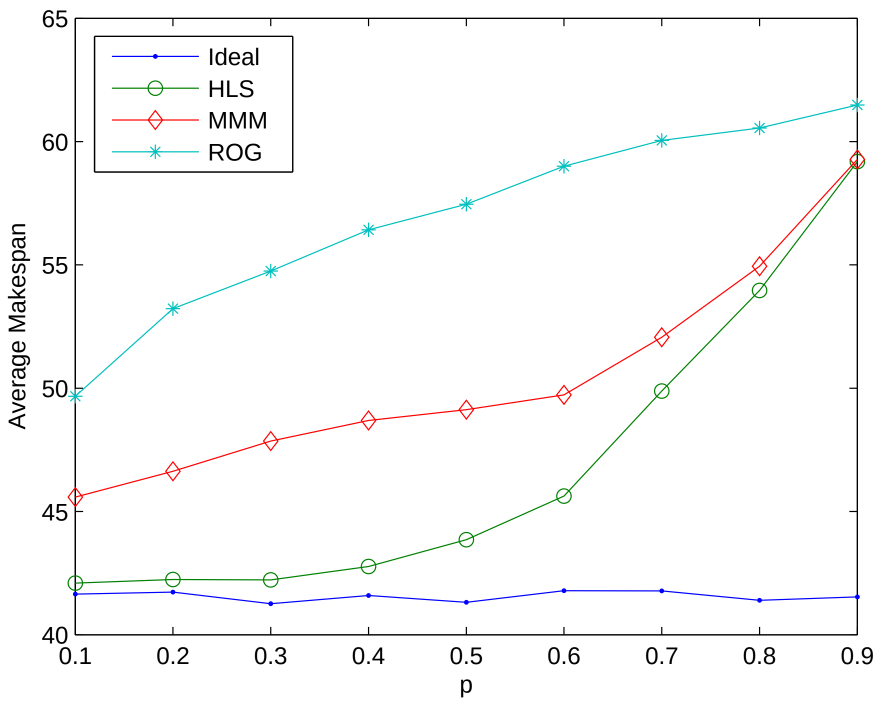


Figure 6.6: Average total processing time versus p.

rameters. We first vary B from 10 to 50 and the corresponding processing time is shown in Fig. 6.3. In addition, we vary M from 10 to 50 and Fig. 6.4 plots the processing time versus M. Further, we vary J from 5 to 50 and Fig. 6.5 depicts the processing time versus J. Moreover, to investigate the effect of different load types, we vary p among $[0.1, 0.9]$ and Fig. 6.6 plots the processing time versus p.

Fig. 6.3 shows that as B increases the processing time of all experiment sets decreases as communication time is reduced. When B is quite small, the performance of HLS and MMM is close to the ideal case, indicating that for slow networks communication delays dominate the performance. Also, when B is large the performance of all algorithms saturates, showing that once network speed is very large, communication time can be effectively hidden and hence varying B impacts little on the average makespan. Such saturation is also shown in Fig. 6.4, wherein when M is large there are abundant processors to accommodate divisible load-s. Accordingly increasing M impacts little on the performance. In this case the performance of MMM is also close to the ideal case due to abundant processors.

Fig. 6.5 shows that when J is small the performance of HLS and MMM is very close. This is because when J is small it is possible that not all processors are fully utilized throughout the execution. Thus this is similar to the case that M is large. As J grows HLS exhibits its advantages and its performance get close to the ideal case.

Fig. 6.6 captures the influence of proportion of indivisible loads to the entire loads. It is shown that when grows from 0.1 to 0.9 the total processing time of the 3 algorithms increases since indivisible loads cannot be parallelized and thus more indivisible loads lead to longer makespan. Also, Fig. 6.6 shows that HLS outperforms MMM and ROG when p is less than 0.8. When p is small, the performance of HLS is close to the ideal case since HLS can efficiently schedule divisible loads to optimize the resource utilization when there are many divisible loads. But when p grows the performance gap between HLS and the ideal case increases and the performance of HLS approaches the performance of MMM. This is due to the fact that there are only a few divisible loads so that resource utilization can not be effectively maximized in HLS.

Chapter 7

Dynamic Scheduling of Hybrid Real-Time Tasks

The general problem of scheduling hybrid loads has been addressed in last chapter. In this chapter we further consider this problem in real-time systems to enhance quality-of-service (QoS) and provide performance guarantees. We propose a dynamic (online) real-time scheduling algorithm referred to as *Hybrid Loads Push-Pull Scheduling* (HLPPS) algorithm for handling a set of tasks comprising both divisible and indivisible real-time tasks on cluster systems. HLPPS is shown to efficiently exploit the parallelism in divisible loads without undermining the schedulability of indivisible loads and thereby optimize the overall performance. We perform rigorous performance evaluation studies to quantify the performance of our algorithm on a variety of scenarios.

7.1 Problem Formulation

We consider two distinct network platforms - tightly coupled (in Section 7.2) and loosely coupled clusters (in Section 7.3) in designing the strategy. The difference between the two platforms is that for tightly coupled clusters the underlying networks are high speed networks so that communication costs are negligible, while for loosely coupled clusters the communication costs are non-zero and the communication speed (bandwidth) of the LAN is B. For the case of loosely coupled clusters, we adopt the uni-port/sequential transmission model in designing our algorithm, as described in Chapter 3.

We assume a real-time aperiodic task model in which each load L_j arrives at time A_j and D_j is its deadline. We let S_j be the size (in bytes) of L_j and W_j be the workload, i.e., the amount of computations (e.g., floating point operations) required to compute L_j. Tasks can be either divisible or indivisible and the detailed model of hybrid loads has been introduced in paragraphs 2 and 3 of Section 6.1. Following prior works [71, 72], we assume that after tasks arrive the workloads of the tasks (W_js) are known thanks to prediction mechanisms such as code profiling and statistical prediction. Finally, we assume that indivisible tasks are non-preemptable but divisible loads are preemptable due to their low-granularity and divisible nature.

7.2 Proposed HLPPS Algorithm

The scheduler has complete knowledge about currently active list of tasks, but not about future tasks which have not arrived. When a new task arrives at time T_0, the scheduler runs an admissibility test to dynamically determine whether it is feasible

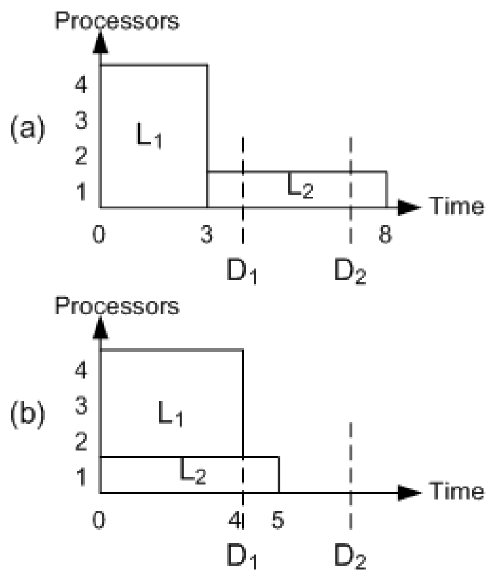


Figure 7.1: Motivating example: (a) scheduling L_1 first and (b) scheduling L_2 first.

to schedule the new task without compromising the guarantees for previously admitted tasks. The admissibility test is conducted as follows. The scheduler generates a task list containing the new task and all previously admitted tasks. It then executes HLPPS to schedule all tasks in the list. Upon the completion of the test, if all tasks can be completed within their respective deadlines, it generates a new schedule to replace the existing schedule and the new task is accepted. Otherwise, it rejects the new task and maintains the existing schedule.

Before presenting HLPPS, we first introduce the motivating idea behind it. Prior scheduling policies neglect the types of different loads, which may result in inefficient scheduling decisions. We can immediately realize this via the following example, as shown in Fig. 7.1. Consider processing two loads on 4 processors. The computation speed of each processor is 1 data unit per time unit. The divisible load L_1 has 12 units of data and its deadline is 4. The indivisible load L_2 has 5 units of data and its deadline is 7. The widely used EDF policy will schedule

L_1 first on all processors according to DLT rule [1], causing L_2 to violate its deadline, as shown in Fig. 7.1(a). However, if we schedule L_2 first on any processor, L_1 can also be successfully scheduled on the other 3 processors, as shown in Fig. 7.1(b). This example shows that in some cases even if the total amount of available computation time of the processors is greater than the required computation time of an indivisible task, yet it is still possible that none of the processors can accommodate the task without violating its deadline. Therefore, it could be beneficial to schedule indivisible tasks first since they cannot be partitioned to gain the benefit of parallel processing.

To this end, we propose the HLPPS algorithm. Fig. 7.2 shows the timing diagram of HLPPS for an example of scheduling a task list containing 4 indivisible loads (L_1, L_2, L_3, and L_4) and 1 divisible load (L_5). As shown in Fig. 7.2, HLPPS consists of 4 steps. In Step 1 indivisible loads are first scheduled since they are inflexible and cannot be partitioned for parallel computing. Step 2 delays (or "pushes forward") the scheduled indivisible loads to create time slots to accommodate divisible loads. Then, Step 3 exploits the parallelism in divisible loads and schedules them to all processors. Finally, in Step 4 indivisible loads are advanced (or "pulled backward") to enhance processor utilization and save time for unexpected future loads. Below we first present the details of HLPPS for the case of negligible communication costs. Then in next section we extend HLPPS to address non-zero communication costs for loosely coupled networks.

Step 1 first schedules indivisible loads. EDF is applied to determine the queuing order of the indivisible loads. However, the EDF rule answers only the "queuing order" question, but it does not determine the processor where the selected task is assigned. To address this dilemma in processor assignment, the *Earliest*

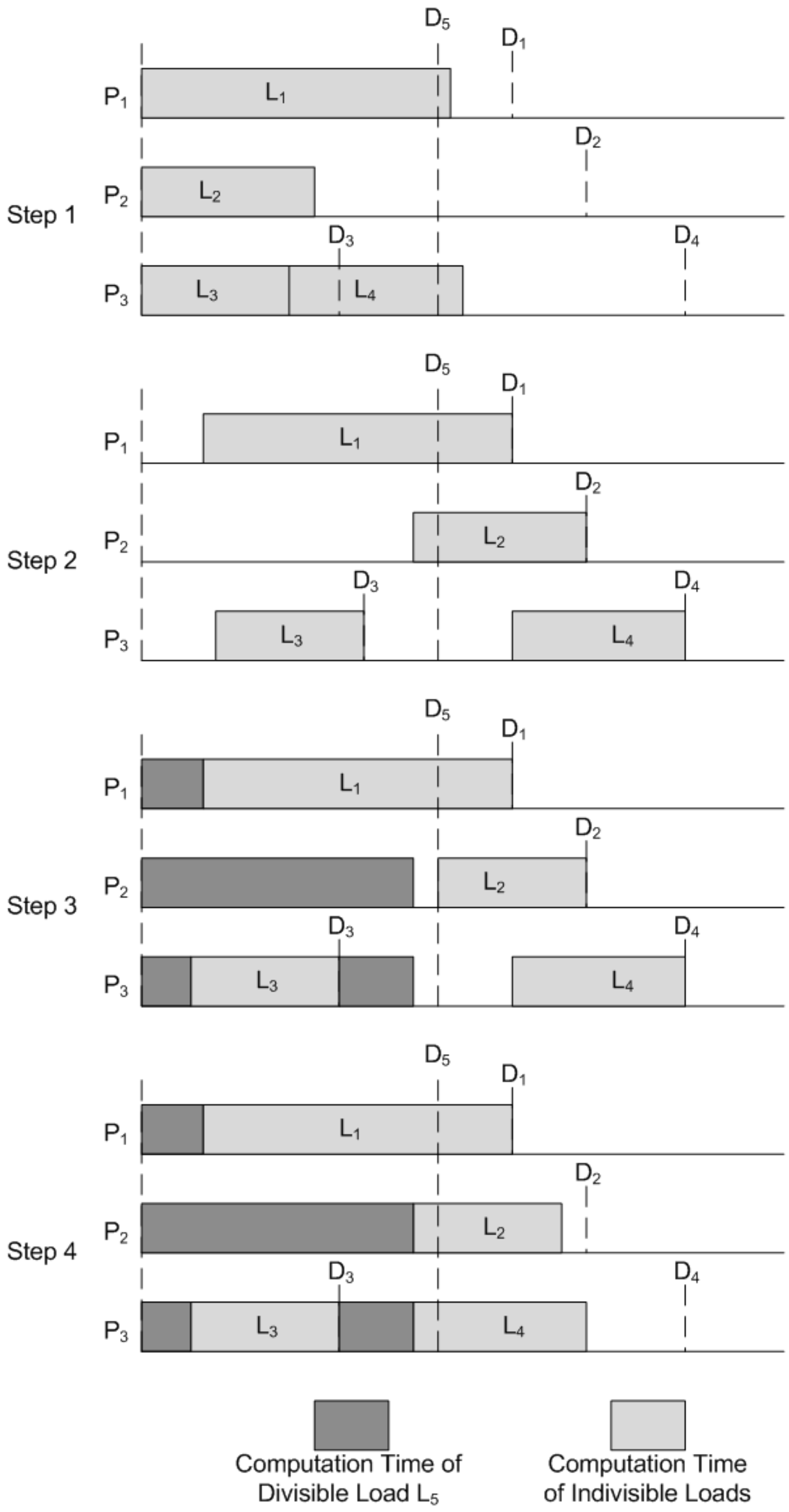

Step 1
Step 2
Step 3
Step 4
P1
P2
P3
L1
L2
L3
L4
D1
D2
D3
D4
D5
Computation Time of Divisible Load L5
Computation Time of Indivisible Loads

Figure 7.2: Timing diagram of HLPPS.

Completion Time (ECT) policy described below can be used. Suppose L_j is selected by EDF. Among the M available processors, the ECT rule selects the one that minimizes the following quantity:

$$\hat{i} = \arg \min_{i \in \{1,\cdots,M\}} \{T_i^{ES} + \frac{W_j}{C_i}\} \tag{7.1}$$

where T_i^{ES} is the earliest start time on processor P_i, which can be calculated according to prior load allocations. Now we can generate an initial schedule for indivisible tasks. If some task violates its deadline, then we fail to generate a feasible schedule and the new task should be rejected. Note that this schedule only assigns indivisible loads to processors and determines the execution order of loads allocated to the same processor, but the start time of each indivisible load can be changed in latter steps.

Step 2 postpones (or "pushes forward") the indivisible tasks in order to create time slots for accommodating divisible loads. If some indivisible loads are not urgent, processing them too early may occupy computation time which should be allocated to divisible loads with urgent deadlines and cause the divisible loads to miss their deadlines. Therefore the indivisible loads scheduled in Step 1 should be delayed to create available time slots for urgent divisible loads. We defer the indivisible tasks according to the decreasing order of their deadlines. This is because for the indivisible tasks allocated to the same processor, if we do not first delay the task with the latest deadline, we cannot delay other tasks without violating the executing order of the tasks. Suppose L_{ij} is the j-th indivisible load ($j = 1, 2, ..., Y_i$) among Y_i indivisible loads which are allocated to P_i. Let T_{ij}^S be the start time and T_{ij}^E be the end time of L_{ij} obtained in Step 1. Thus L_{ij}'s processing time T_{ij}^P equals to $(T_{ij}^E - T_{ij}^S)$. Also, let T_{ij}^{NS} be the new start time and T_{ij}^{NE} be the new end time of L_{ij} after deferring. Hence the new end time T_{ij}^{NE} can be obtained as follows:

$$T_{ij}^{NE} = \begin{cases} \min(T_{i(j+1)}^{NE} - T_{ij}^{E} + T_{ij}^{S}, D_{ij}) \qquad if\ j < Y_i \\ D_{ij} \qquad if\ j = Y_i \end{cases} \tag{7.2}$$

where D_{ij} is the deadline of L_{ij}. In this case, we can recursively calculate the new start times of all indivisible tasks after delaying.

Then Step 3 schedules divisible loads. We also use EDF to schedule divisible loads. For each divisible load, we attempt to utilize all available processors to compute it since this strategy naturally minimizes processing time and guarantees high processor utilization. In the contrary, prior works [17, 18, 19] tend to employ the minimum number of processors to meet tasks' deadlines. However, if there are many processors but only a few tasks and the tasks' deadlines are not urgent, some processors will remain idle according to their strategies. This leads to low processor utilization and may finally cause some future tasks to be rejected since current tasks are finished late.

Now we suppose a divisible load L_j is selected by EDF and the distribution of the load should be determined. We first need to determine the finish time T_j^f of L_j and then map the load to the available time slots before T_j^f on all processors. As processors' available times are different, the total available computation power varies with processors becoming available or unavailable. Suppose before deadline D_j there are X time instants at which some processors become available or unavailable. We denote the x-th time instant as $t(x)$ $(x = 1, 2, ..., X)$ and $t(1)$ is the computation start time of L_j. In addition, we define $C^s(t)$ as the sum of the computation speeds of all available processors at time t. This means at each $t(x)$, $C^s(t(x))$ is updated to a new value. These time instants $t(x)$s and total computation speeds $C^s(t(x))$s can be easily calculated according to the start and end times of the indivisible loads obtained in Step 2. Let $w(t)$ be the amount of load that can be executed before time t on all available processors. Then we can write:

$$w(t(x)) = w(t(x-1)) + C^s(t(x-1)) * (t(x) - t(x-1)) \tag{7.3}$$

where $(x = 2, 3, ..., X)$. Also, $w(D_j)$ can be expressed as:

$$w(D_j) = w(t(X)) + C^s(X) * (D_j - t(X)) \tag{7.4}$$

Because $w(t(1)) = 0$, we can recursively calculate $w(D_j)$ and $w(t(x))$ for all $t(x)$s. If $w(D_j) < W_j$, then deadline D_j will be missed. Let x_0 be the largest x that satisfies $w(t(x)) < W_j$ where $x = 1, 2, ..., X$. We can obtain the finish time T_j^f of L_j as follows:

$$T_j^f = t(x_0) + \frac{W_j - w(t(x_0))}{C^s(t(x_0))} \tag{7.5}$$

According to the start and end times of the indivisible loads, we can easily calculate the total available time between T_0 and T_j^f for each processor and hence obtain the amount of load distributed to each processor. In this way, we iterate to determine the load distributions of all divisible loads. If some task misses its deadline, we fail to generate a feasible schedule and the new task should not be admitted. After divisible loads are scheduled in Step 3, there may be some unused time slots between some divisible loads and indivisible tasks, as shown in Fig.7.2. These leave some room for further optimization.

Therefore, Step 4 advances (or "pulls backward") the indivisible tasks in order to save time for future loads. This benefits future loads as current indivisible tasks can be finished earlier. Without this "pull" operation, some future tasks may be rejected since current tasks are finished late. We hasten the indivisible tasks according to the EDF rule. This is because for the indivisible tasks allocated to the same processor, if we do not advance the task with the earliest deadline, we cannot advance other tasks without violating the executing order of the tasks.

Traditional scheduling policies neglect the types of different loads. For example, when running EDF, in some cases even if the total amount of available computation times of the processors is greater than the required computation time of an indivisible task, yet it is possible that none of the processors can accommodate the task without missing its deadline. In contrast, such cases are greatly reduced for HLPPS since indivisible loads are first scheduled. Also, scheduling indivisible loads first will not impede the scheduling of urgent divisible loads as indivisible loads are delayed to create time slots in Step 2. In addition, HLPPS optimizes processor utilization by parallelizing divisible loads to all processors in Step 3 and advancing indivisible loads in Step 4. Therefore, HLPPS fully exploits parallelism in divisible loads without undermining the schedulability of indivisible loads, optimizing the overall performance. The time complexity of HLPPS for scheduling J_d divisible loads and J_i indivisible loads is $O(J_i^2 + J_iM + J_d^2)$. This guarantees the scalability of HLPPS in real-life implementations.

7.3 Extension of HLPPS for Loosely Coupled Clusters

Now we extend HLPPS to deal with non-zero communication costs for loosely coupled clusters. Since indivisible loads are inflexible, the modification for indivisible loads are rather limited and we mainly focus on optimizing the communication of divisible loads in Step 3. Below we first simply introduce the modifications in Steps 1, 2, and 4.

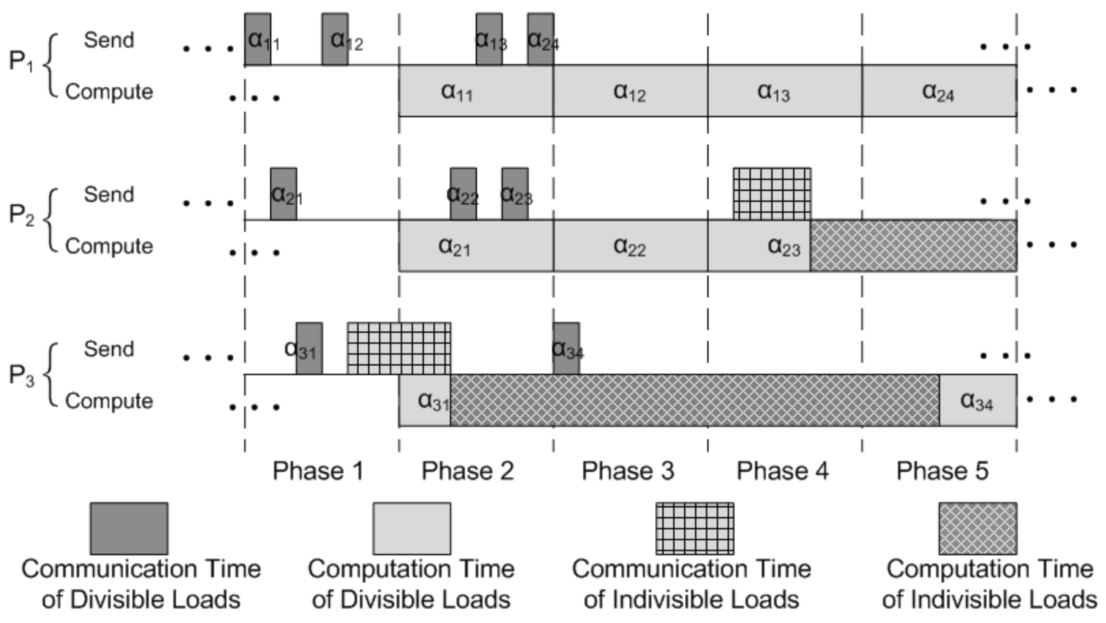

P1
Send
Compute
P2
Send
Compute
P3
Send
Compute
α11
α12
α13
α24
α21
α22
α23
α31
α34
Phase 1
Phase 2
Phase 3
Phase 4
Phase 5
Communication Time of Divisible Loads
Computation Time of Divisible Loads
Communication Time of Indivisible Loads
Computation Time of Indivisible Loads

Figure 7.3: Timing diagram of the divisible load distribution for loosely coupled clusters in Step 3 of HLPPS.

Firstly, the modification in Step 1 is that the earliest start time is modified as $T_i^{ES} = \max(T^{CC}, T_i^F)$ where T^{CC} is the communication completion time of P_0 and T_i^F is P_i's finish time of its previously allocated tasks, which are obtained according to prior load allocation. Secondly, in Step 2, when indivisible tasks are delayed, the scheduled communication time is also delayed as much as possible so that the communication of each task happens immediately before the computation start time of the task. This helps to reduce the contention of communication times and thereby create effective time slots for divisible loads. Lastly, in Step 4, when indivisible tasks are advanced, the scheduled communication time is also advanced as much as possible. This guarantees that the communication will be completed timely so that processors need not waste time in waiting for the communication.

As divisible loads can be arbitrarily partitioned for parallel computing, in Step 3 we aim to optimize the performance by distributing divisible loads in multiple rounds so that communications are overlapped with computations to save unnecessary waiting times. As shown in Fig. 7.3, we design a "phase-based" multi-round load distribution approach, which works in an incremental fashion, executing multiple rounds to distribute and process loads. The length of each phase is T_p, which is fixed and predetermined. We use the term "load chunk" to denote a number of load units distributed in a round to a processor. The identifier α_{ik} in Fig. 7.3 indicates the load chunk allocated to processor P_i in round k. The basic idea of the multi-round approach is that the load chunks of round k ($k = 1, 2, 3, \cdots$) are distributed before phase $(k+1)$ and processed in phase $(k+1)$. In phase $(k+1)$, while computing the chunks of round k, processors can simultaneously receive the chunks of future rounds. In this way, communications and computations are pipelined and overlapped. Also, the transmission of round k must be completed before phase $(k+1)$ starts so that processors can start immediately to compute the chunks of round k in phase $(k+1)$ and P_0 can also start immediately to transmit the chunks of round $(k+1)$ in phase $(k+1)$.

The allocation process starts from round 1 and here we consider an arbitrary round k as an example. As P_0 attempts to dispatch L_j to all available processors, it (sequentially) selects each processor P_i to distributes it a load chunk. We denote the computation time of the chunk as T_{ik}^{comp}. Suppose for P_i there are Q available time slots in phase $(k+1)$ before D_j. The length of each slot is specified as β_q^i. Thus the total available time in phase $(k+1)$ can be written as:

$$t_{comp} = \sum_{q=1}^{Q} \beta_q^i \tag{7.6}$$

Communication time is another constraint in determining T_{ik}^{comp}. Let T^{CS} be the time instant when P_0 starts to transfer loads to P_i. The transfer of this chunk should be finished before phase $(k+1)$ starts. Suppose for P_0 there are

Z available communication time slots between T^{CS} and the communication finish time $(T_0 + kT_p)$. We use θ_z to denote the z-th communication time slot $(z = 1, 2, ..., Z)$. Hence the total available communication time for this chunk is:

$$t_{comm} = \sum_{z=1}^{Z} \theta_z \tag{7.7}$$

Accordingly one can obtain T_{ik}^{comp} as follows:

$$T_{ik}^{comp} = \min(t_{comp}, \frac{BW_j^u t_{comm}}{C_i S_j^u}) \tag{7.8}$$

Knowing T_{ik}^{comp}, we can calculate the chunk size α_{ijk} (in terms of number of units) allocated to P_i from L_j in round k. In reality, this number of units should be an integer. Thus we can write α_{ijk} as follows:

$$\alpha_{ijk} = \lfloor \frac{C_i T_{ik}^{comp}}{W_j^u} \rfloor \tag{7.9}$$

We denote the number of remaining load units of L_j as N_j^r. If N_j^r is less than the chunk size α_{ijk}, then α_{ijk} is accordingly reduced to that number and all unallocated units of L_j are distributed to P_i. Accordingly, P_i will still be available after receiving the chunk since it still has some idle time before phase $(k + 1)$ ends. In this case, P_i can be allocated another chunk in round k from remaining tasks and P_i shall execute these different chunks in phase $(k + 1)$ following their scheduling order so that deadlines will not be violated.

The scheduler iterates this load distribution process to distribute L_j in this round to other processors until all available processors are utilized in this round or communication time is used up $(T^{CS} = T_0 + kT_p)$. Then P_0 starts to distribute L_j in round $(k+1)$ and repeats these steps until the entire L_j has been allocated. Afterwards P_0 can start to distribute next task selected by EDF.

Noted that the length of each phase T_p is not arbitrarily chosen. Since phase 1 is not involved in computation, a very large T_p will significantly reduce available computation time and lead to poor performance. On the other hand, if T_p is too small probably some processors cannot receive a unit as they cannot finish 1 unit in 1 phase. Therefore, we should avoid determining T_p as either very small or very large. When implementing HLPPS in real-world loosely coupled clusters, we should check the task granularity and the sizes of the data before T_p can be determined. In next section we will evaluate the impact of T_p on our algorithm.

7.4 Performance Evaluation

In this section, we evaluate the performance of HLPPS by rigorous simulation experiments, following the typical style of simulation study used in prior works [17, 67, 71]. Our interested metric is *acceptance ratio*, which is defined as the ratio of the number of tasks found schedulable to the number of tasks arriving for scheduling.

As there are no strategies available in the literature for the scheduling of hybrid divisible and indivisible loads, to understand the merits of our strategy, we compare it with two basic (default) algorithms: EDF-ECT-MAXP and EDF-ECT-MINP. Both EDF-ECT-MAXP and EDF-ECT-MINP adopt EDF for load scheduling and ECT for the load allocation of indivisible loads. For the load distribution of divisible loads, EDF-ECT-MAXP utilizes all processors to minimize the completion time while EDF-ECT-MINP utilizes the minimum number of processors so that the processors can finish computing when the deadline is met.

The initial simulation configurations are set as follows: The number of processing nodes is 20. For each processor P_i the computation speed C_i is 1 and the communication speed B is 100. To generate a set of real-time divisible loads, we assume that the inter-arrival times follow an exponential distribution with a mean of $1/\lambda$ and we initially set $1/\lambda$ as 5 time units. Tasks can be either divisible or indivisible. To ensure that all possible distributions of hybrid loads (percentage of indivisible and divisible loads) are considered in our simulation experiments by our strategies, we introduce a parameter p to denote the probability that an incoming load L_j is indivisible and p is initially set as 0.5. Thus, by varying this parameter the fraction of the indivisible and divisible loads in a given pool of tasks can be varied and hence performance of our algorithm can be evaluated. A task's relative deadline (defined as $D_j - A_j$) is assumed to be uniformly distributed in the range $[T_j^1, T_j^1 * (1+R)]$ where T_j^1 is the compute time for load L_j on one processor ($T_j^1 = \frac{W_j}{C_i}$) and R is referred to as *laxity parameter* which denotes the urgency of the deadlines. Thus smaller values of R indicate that deadlines are more urgent. The initial value of R is set as 1. In addition, to study the effect of different load sizes, we let S_j be uniformly distributed in the range $[100, 200]$, let W_j be uniformly distributed in the range $[100, 200]$, and let N_j be uniformly distributed in the range $[10,000, 20,000]$. The length of each phase T_p is set to be 0.1 time units.

In the following experiments we vary our interested parameters while fixing other parameters as their initial values to study the effect of the interested parameters. We first evaluate the influence of the laxity parameter R on our algorithms. To simulate different deadlines we vary R among the range $[0.2, 2]$ and Fig. 7.4 depicts the acceptance ratio versus R. In addition, as divisible loads can be distributed to multiple processors to gain the benefit of parallel computing, the deadlines of incoming divisible loads may be set to be more urgent. Therefore we also modify the relative deadlines of the incoming divisible loads as uniformly distributed in the range $[T_j^1 * R_d, T_j^1 * R_d * (1+R)]$ where, $R_d \in [0, 1]$ quantifies the inverse of the

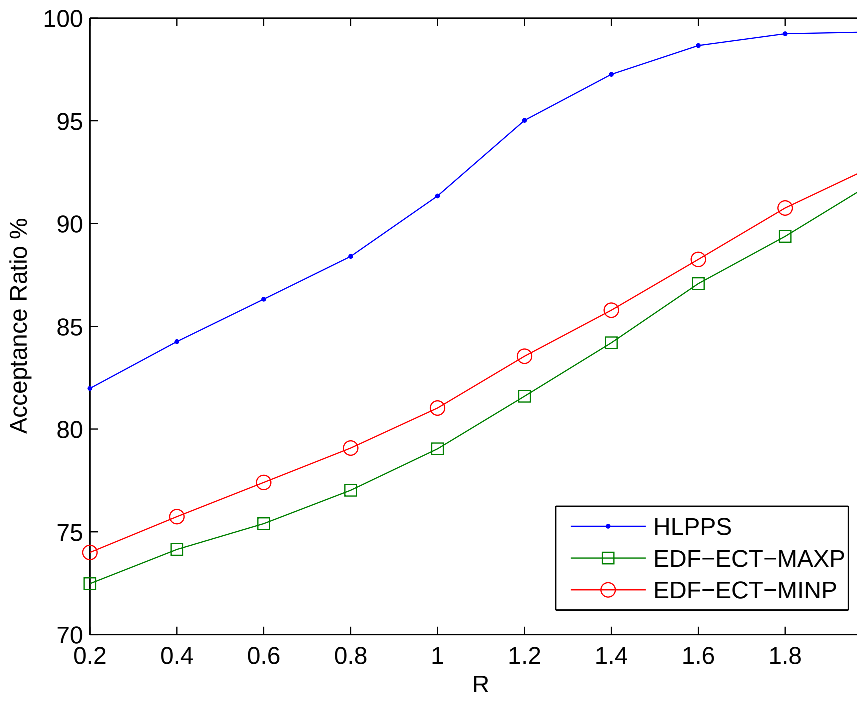


Figure 7.4: Average acceptance ratio versus R.

number of processors to be used to process the divisible loads. The R_d parameter, referred to as *divisible load laxity parameter*, denotes the urgency of deadlines of divisible loads. We typically consider using values $1/16, 1/8, 1/4, 1/2, 1$ for R_d and Fig. 7.5 depicts the corresponding acceptance ratio. To investigate the effect of various proportions of loads, we vary p from 0 to 1 and Fig. 7.6 plots the acceptance ratio versus p. Also, we vary $1/\lambda$ from 1 to 10 and Fig. 7.7 shows the acceptance ratio versus $1/\lambda$. Further, we vary M from 10 to 30 and Fig. 7.8 plots the acceptance ratio versus M. Moreover, we vary B among $[20, 200]$ and the corresponding acceptance ratio is shown in Fig. 7.9. We vary T_p from 0.001 to 1 and Fig. 7.10 plots the acceptance ratio versus T_p. Finally, as one may wonder whether the algorithms are fair to both divisible and indivisible loads, we present Figs. 7.11 and 7.12 to show the acceptance ratios of both divisible and indivisible

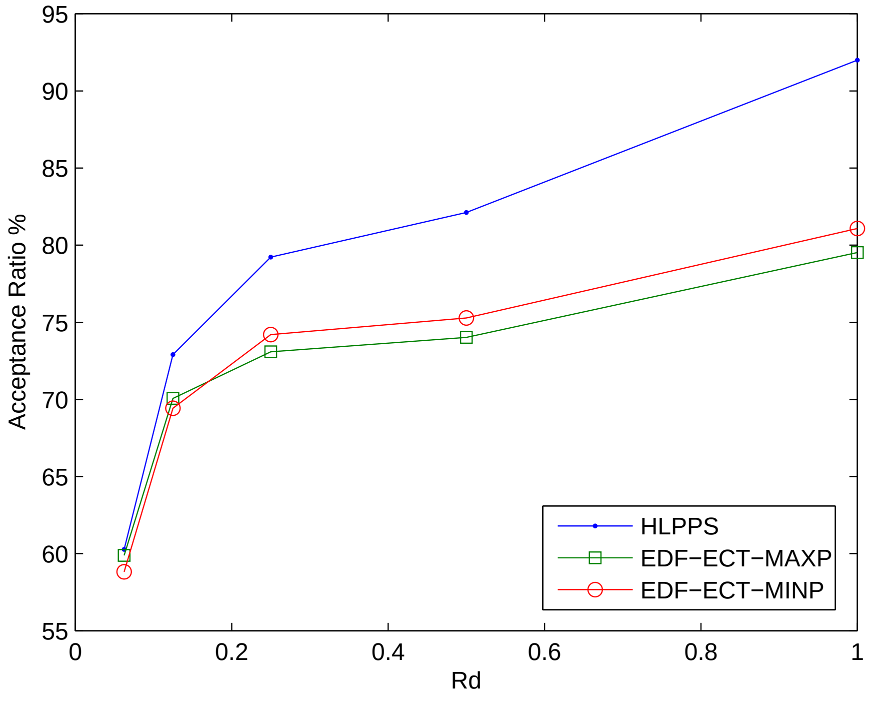


Figure 7.5: Average acceptance ratio versus R_d.

loads versus R and p, respectively. In other words, with R and p varying, Figs. 7.11 and 7.12 depict the respective acceptance ratios of divisible and indivisible loads while Figs. 7.4 and 7.6 plot the total acceptance ratios of all loads. In Figs. 7.11 and 7.12, the acceptance ratios of divisible loads for the 3 algorithms are denoted as HLPPS-DL, EDF-ECT-MAXP-DL, and EDF-ECT-MINP-DL, respectively; the acceptance ratios of indivisible loads for the 3 algorithms are denoted as HLPPS-IL, EDF-ECT-MAXP-IL, and EDF-ECT-MINP-IL, respectively.

The simulation results show that the HLPPS algorithm significantly outperforms EDF-ECT-MAXP and EDF-ECT-MINP in most cases. This is because for EDF-ECT-MAXP, in some cases even if the total amount of available computation times of the processors is greater than the required computation time of an indivisible task, yet it is possible none of the processors can accommodate the task with-

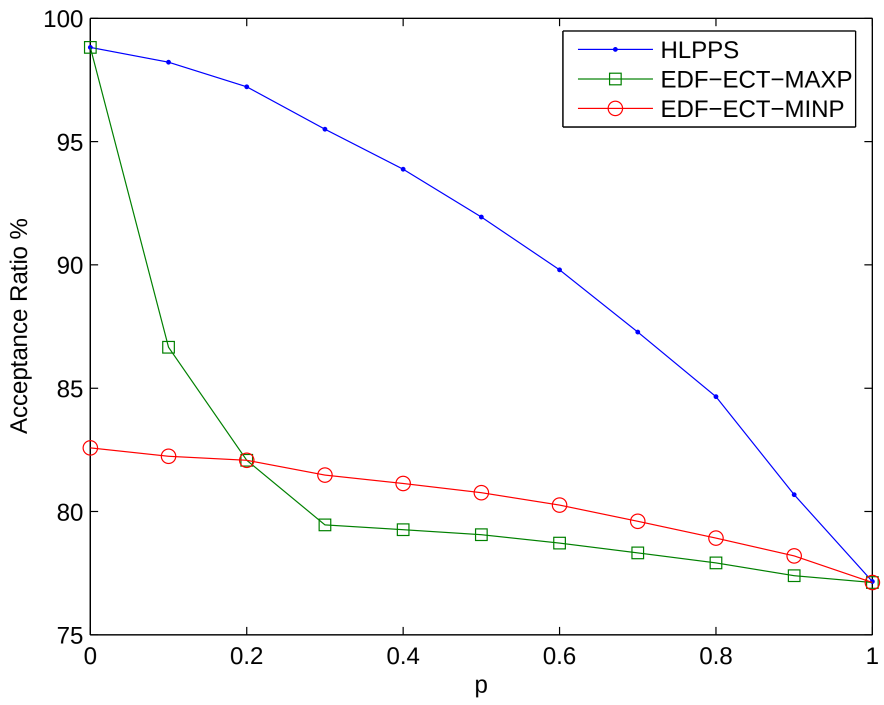


Figure 7.6: Average acceptance ratio versus p.

out violating its deadline. Although this case happens less for EDF-ECT-MINP, yet the resource utilization of EDF-ECT-MINP may be low and some future tasks may be rejected as current tasks are finished late. In the contrary, HLPPS can efficiently avoid these drawbacks as we described in Section 7.2. This explains why HLPPS algorithm outperforms EDF-ECT-MAXP and EDF-ECT-MINP in these figures.

Fig. 7.4 and 7.5 present the effect of urgency of deadlines. Fig. 7.4 shows that as R grows the acceptance ratios of the 3 algorithms increase. Also, the superiority of HLPPS is insensitive for different values of R. Fig. 7.5 shows that when parameter R_d is small (divisible loads are very urgent) the advantage of HLPPS over EDF-ECT-MAXP and EDF-ECT-MINP is not obvious. This is due to the fact that when R_d is small the choices on the scheduling of divisible loads

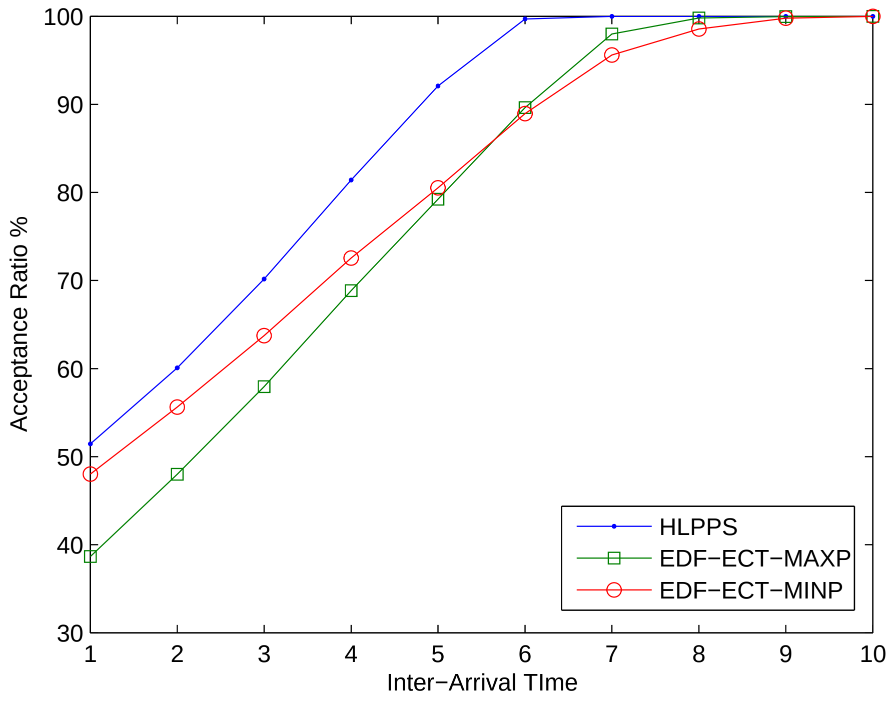


Figure 7.7: Average acceptance ratio versus $1/\lambda$.

become very limited and thus there is little room for optimizing the scheduling of hybrid loads. Therefore, as R_d grows HLPPS gradually exhibits its advantages over EDF-ECT-MAXP and EDF-ECT-MINP.

Fig. 7.6 captures the influence of proportion of indivisible loads to the entire loads. As p grows from 0 to 1 the acceptance ratio of HLPPS decreases since divisible loads can be easily scheduled and thus acceptance ratio is higher when there are more divisible loads. Also, HLPPS significantly outperforms EDF-ECT-MAXP and EDF-ECT-MINP when p is among $[0.1, 0.8]$. When p equals to 0 or 1, HLPPS is equivalent to EDF-ECT-MAXP and hence they deliver the same performance when p equals to 0 or 1.

Fig. 7.7 shows that as the inter-arrival time $1/\lambda$ increases the load acceptance ratios of all experiment sets increase since reducing the loads arrived in a

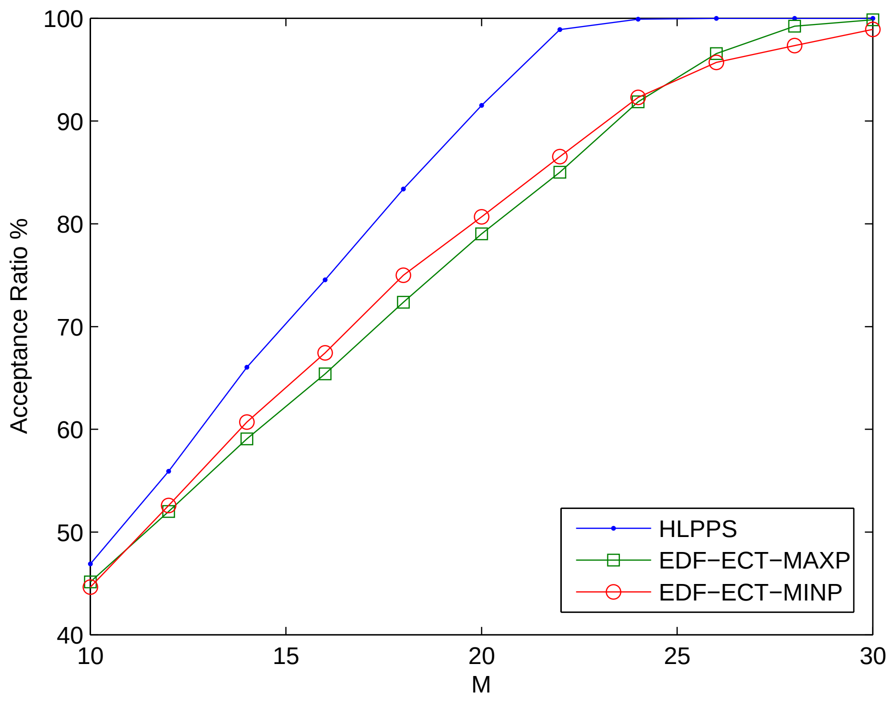


Figure 7.8: Average acceptance ratio versus M.

period of time can decrease the contention on the computing resources and hence increase the acceptance ratio. In addition, when $1/\lambda$ is small, the performance of EDF-ECT-MINP is closer to HLPPS than EDF-ECT-MAXP. This is because the drawback of EDF-ECT-MINP lies on the processor utilization, but when $1/\lambda$ is small, tasks frequently arrive so that all processors can be fully utilized. Consequently, the drawback of EDF-ECT-MINP is mitigated and EDF-ECT-MINP delivers better performance than EDF-ECT-MAXP for small values of $1/\lambda$.

From Fig. 7.8 we can observe that when M is small the performance of HLPPS is comparatively close to EDF-ECT-MAXP and EDF-ECT-MINP. A plausible explanation is that when there is only a few processors HLPPS cannot effectively exploit the parallelism of divisible loads to optimize the scheduling of hybrid loads. Therefore when M is small HLPPS cannot significantly outperform EDF-ECT-

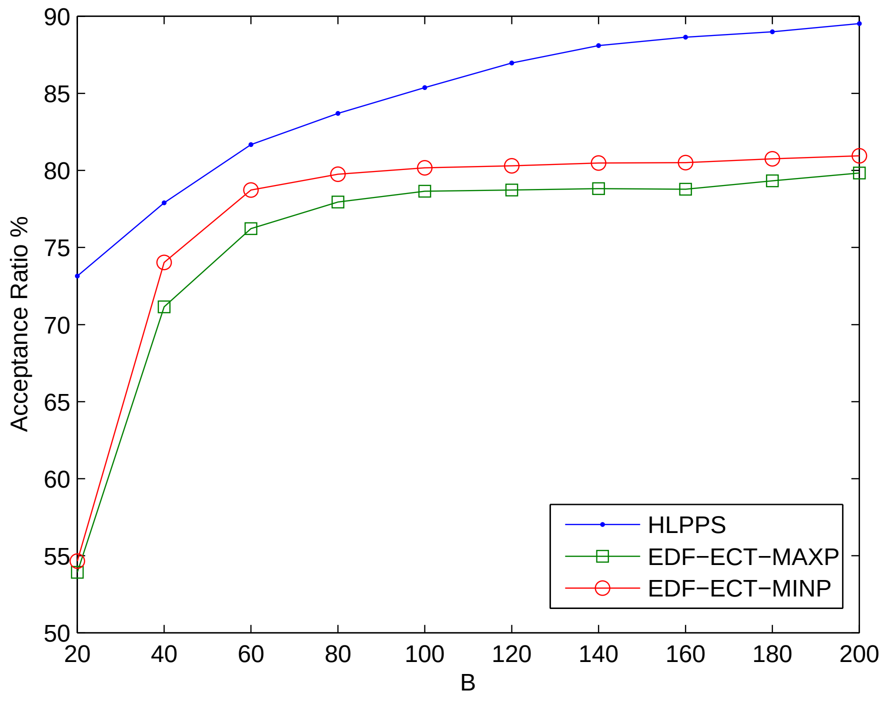


Figure 7.9: Average acceptance ratio versus B.

MAXP and EDF-ECT-MINP. As M grows HLPPS gradually exhibit its advantage over EDF-ECT-MAXP and EDF-ECT-MINP until their acceptance ratios reach 100%.

Fig. 7.9 depicts the influence of the communication speed B. It shows that when B is small the performance of EDF-ECT-MAXP and EDF-ECT-MINP sharply deteriorates. This is because in EDF-ECT-MAXP and EDF-ECT-MINP, some indivisible loads may be scheduled to be started early. When B is small the communication of such loads fills available communication time and thus cause that early phases cannot be efficiently utilized. But in HLPPS such case hardly happens since indivisible loads are delayed in Step 2.

Fig. 7.10 shows that when T_p is greater than 0.8 the acceptance ratios of the 3 algorithms sharply decrease as T_p increases. This is because when T_p is large,

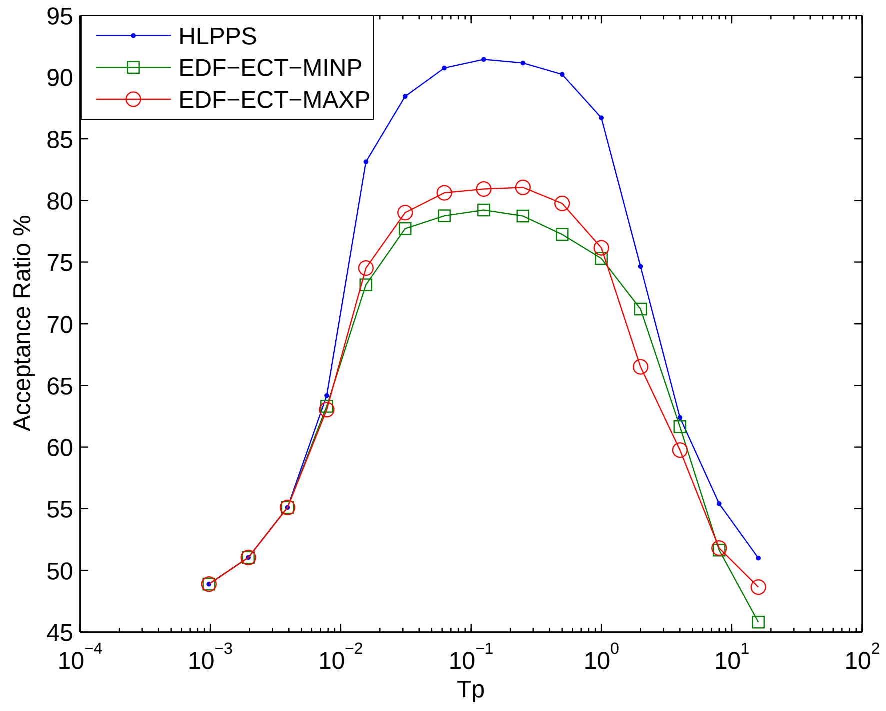


Figure 7.10: Average acceptance ratio versus T_p.

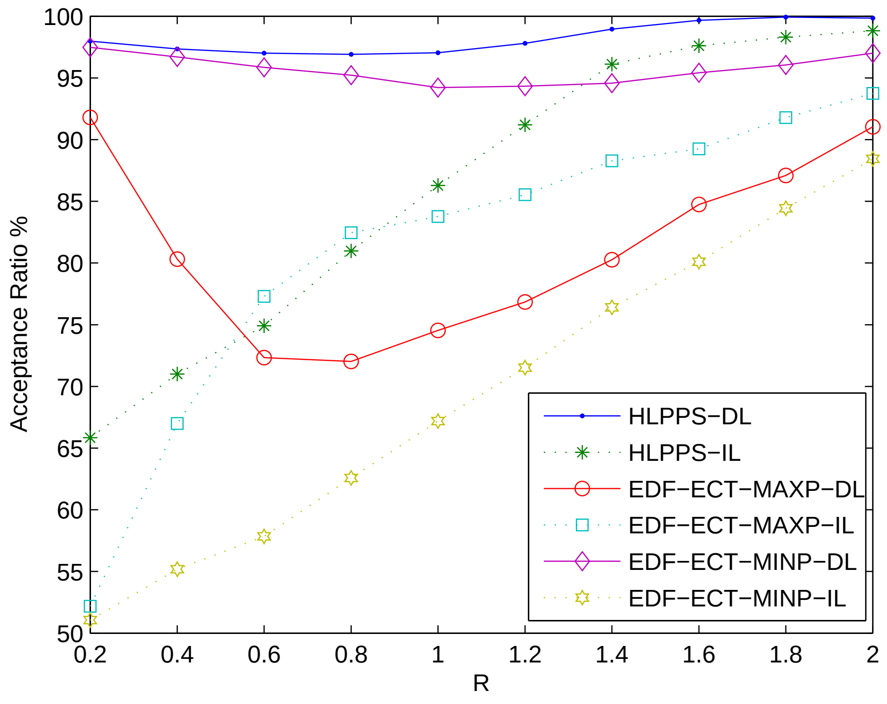


Figure 7.11: Average acceptance ratio versus R for divisible and indivisible loads.

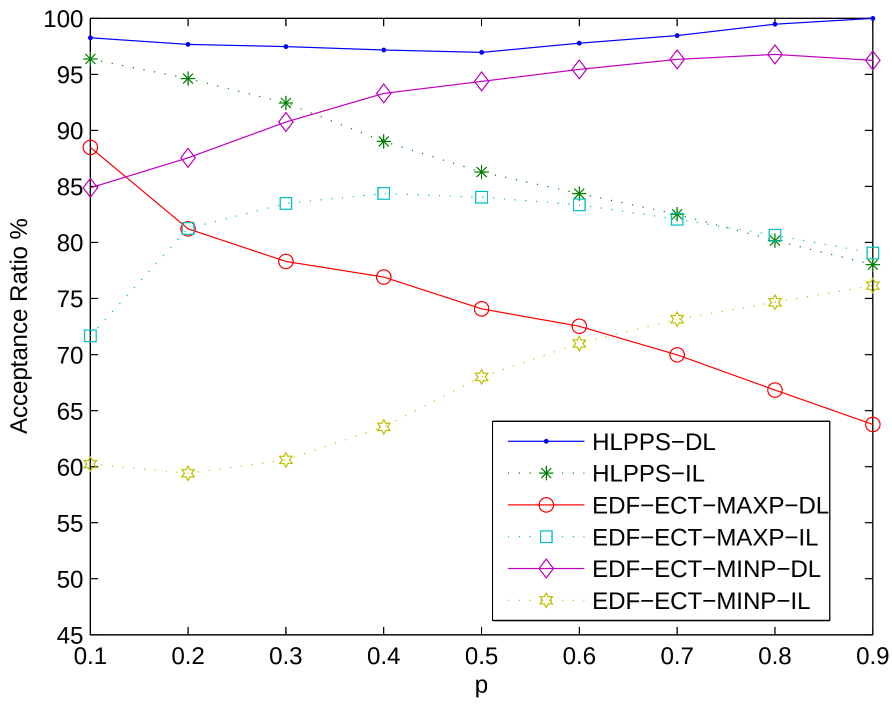


Figure 7.12: Average acceptance ratio versus p for divisible and indivisible loads.

available computation time is significantly reduced as T_p increases. In addition, when T_p is less than 0.0125 the acceptance ratios of the 3 algorithms decrease as T_p reduces. This is due to the fact that if T_p is too small probably some processors cannot receive a unit as they cannot finish 1 unit in 1 phase. Further, the performance of the 3 algorithms stabilizes when T_p is among $[0.0125, 0.8]$. This shows that although T_p cannot be too small or too large, there is still a wide range in which varying T_p only slightly influences the performance. This range is related to the granularity and the sizes of the loads. As the multi-round approach is designed for processing divisible loads consisting of many low-granularity computations, choosing satisfactory values of T_p may not be challenging. When implementing HLPPS for real-world applications, simulations or experiments can help determine satisfactory values of T_p once the granularity and size of the data are given.

Finally, Figs. 7.11 and 7.12 depict the fairness issues. In Figs. 7.11 and 7.12 we can observe that in the 3 algorithms the acceptance ratios of divisible loads are higher than indivisible loads. This is due to the difference in the divisibility of the loads. However, although divisible and indivisible loads have different acceptance ratios, yet HLPPS outperforms EDF-ECT-MAXP and EDF-ECT-MINP in terms of respective acceptance ratios of both divisible and indivisible loads.

Chapter 8

Conclusions and Future Recommendations

8.1 Conclusions

In this thesis, we have studied several critical issues on scheduling multiple divisible loads on clusters. We have contributed several efficient scheduling algorithms that are aware of tasks' different processing requirements and load types. These include both real-time and non-real-time cases. Also, we have performed extensive performance evaluation studies to quantify the performance of our algorithms on a variety of scenarios.

Firstly, in Chapter 4 we have addressed the problem of scheduling multiple divisible loads with arbitrary processor release times and heterogeneous processing requirements on cluster systems. We have proposed two novel scheduling strategies: SSS and DSS. These strategies provide efficient load balancing and resource utilization while elegantly addressing arbitrary release times. In addition, we have proposed a requirement-aware load selection policy (MRF) and integrated it into SSS and DSS to handle the contention among different loads with various processing requirements. The simulation results have shown that our strategies are very efficient for both static and dynamic cases under various system parameters.

Then, in Chapter 5 we have investigated the problem of scheduling heterogeneous divisible loads with deadlines on networked computing platforms. Unlike the prior works which only study single-round approaches, in this work we have designed multi-round approaches to handle divisible loads with deadlines. Also, in this study we have considered scheduling divisible loads with different processing requirements on heterogeneous computing systems while in prior works divisible load applications are assumed to be of the same type. We have presented RARTS scheduling algorithms, which consist of the MSCF scheduling policy, and two load distribution strategies, AEP and LCF. We have carried out rigorous simulations to evaluate and compare the performance of the proposed strategies. The result-s have demonstrated the effectiveness and competitiveness of RARTS algorithms when compared to existing methods.

Furthermore, in Chapter 6 we have tackled the problem of scheduling a set of tasks consisting of both divisible and indivisible loads on clusters. Such situations are common in real-life since clusters are deployed to address any types of loads. We have proposed the HLS algorithm, which can efficiently exploit the parallelism of divisible loads. HLS also uses a pipelining technique to hide communication time and improve the overall performance. Simulation results are presented to evaluate the performance of the proposed HLS strategy.

Finally, in Chapter 7 we have studied an important real-life problem of scheduling a set of tasks comprising a mix of both the divisible and indivisible real-time loads on clusters. As in practice, we have considered systems that are tightly coupled as well as loosely coupled cluster systems. We have proposed the HLPPS algorithm which efficiently exploits parallelism in divisible loads without undermining the schedulability of indivisible loads and thereby enhances the performance of the system. With our design, we have conducted extensive simulation tests to quantify the performance of our algorithm. The simulation results have shown that the HLPPS algorithm is very efficient and it outperforms EDF-ECT-MAXP and EDF-ECT-MINP under various system parameters.

8.2 Future Work

This thesis is expected to spur further research in one or more of the following directions. One future direction is to allow the master node P_0 to perform simultaneous communications to processors. The parallel communication mode can be beneficial for computing platforms over wide area networks (WAN) [11] to achieve higher throughput than the sequential communication mode due to bandwidth-sharing properties. In addition, computing platforms on WAN are likely to be non-dedicated. Thus the communication and computation capacities of processors in such systems may fluctuate with time. As computer grids are widely deployed over WAN to perform distributed computing, it is valuable to explore scheduling strategies accommodating the parallel communication mode [11] and fluctuated communication and computation capacities.

While this thesis only considers star networks, it will also be interesting to explore the scheduling problem on tree networks [3]. In complex network scenarios such as wide-area networks, the tree topology may be useful for scheduling. In such network scenarios, when P_0 leaves some processors unutilized under star topologies due to slow links connected to the processors, tree topologies may utilize the processors as simultaneous communications can be allowed on both the root node and leave nodes.

In addition, since in our scheduling strategies all scheduling activities occur on the master node P_0 only, this centralized fashion may result in significant overheads on P_0 when the number of nodes is large or the network bandwidth is limited. In this case, another future direction is to explore multi-source strategies [2] that simultaneously distribute workloads from multiple sources (or master nodes). Such multi-source strategies work in a decentralized fashion to achieve high performance and improve the scalability when the number of processors is large.

Moreover, one may attempt to apply an affine cost model in DLT to include start-up costs. Our strategies can be easily modified for the affine model without affecting the flow of the main content. For example, consider SSS and DSS in Chapter 4. Suppose O_{comp} and O_{comm} are computation and communication start-up overheads. The only change is to use $(T_p(k+1) - O_{comp})$ to replace $T_p(k+1)$ and use $(T^{comm} + O_{comm})$ to replace T^{comm} in Equations (4.3) and (4.5). However, in practice, start-up overheads are shown to be small, usually negligible. In [5], such overheads are usually bounded in a few seconds in real-life measurements. When the number of rounds is not too large start-up overheads are indeed negligible. On the other hand, as we demonstrated via simulations, the number of rounds for our strategies should not be large for efficient resource utilization. Thus the inclusion of affine model may not be a worthwhile attempt. However, as a future work, it would be interesting to determine an optimal, if not, an acceptable number of rounds to be used for maximizing resource utilization.

Furthermore, to implement the proposed strategies in real world, one may attempt to consider limited buffer constraints. Actually the proposed strategies can easily handle buffer constraints. Since the proposed strategies adopt multi-round fashion, processors only need to buffer a few amounts of loads simultaneously in each round. In case that only small buffers are available, reducing T_p can help to decrease required buffer space.

Chapters 6 and 7 only consider divisible and indivisible loads. But one may attempt to accommodate other types of loads, such as precedence-constrained tasks (task graphs). Indeed the proposed algorithms can be directly applied to schedule a set of tasks comprising divisible, indivisible, and precedence-constrained tasks. In this case an indivisible load can be deemed as a special case of task graphs in which there are no edges and only one node in the task graph. Then task graphs are first scheduled by existing algorithms in Step 1 of either HLS or HLPPS (presented in Chapters 6 and 7, respectively). Then divisible loads can be addressed in later steps as usual. However, since the scheduling of task graphs is quite challenging, especially in real-time context, it will be interesting to explore scheduling algorithms which may further optimize the performance in the presence of hybrid tasks comprising divisible, indivisible, and task graphs.

Finally, in Chapter 7 we utilize EDF in HLPPS to address indivisible loads and divisible loads in Steps 1 and 3, respectively, because EDF is the most well-known and widely-used real-time scheduling policy. Nevertheless, it is also possible and interesting to employ other algorithms to replace EDF in Steps 1 and 3 of HLPPS. Such replacement will not influence the basic mechanism of HLPPS.

Bibliography

[1] B. Veeravalli, D. Ghose, V. Mani, and T.G. Robertazzi, Scheduling Divisible Loads in Parallel and Distributed Systems, *IEEE Computer Society Press*, Los Almitos, California, 1996.

[2] J. Jia, B. Veeravalli, J. Weissman, "Scheduling Multisource Divisible Loads on Arbitrary Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 520-531, Apr. 2010.

[3] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang, "Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 3, pp. 207-218, Mar. 2005.

[4] T.E. Carroll and D. Grosu, "Strategyproof Mechanisms for Scheduling Divisible Loads in Bus-Networked Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 19, no. 8, Aug. 2008.

[5] Y. Yang, K. Raadt, H. Casanova, "Multiround Algorithms for Scheduling Divisible Loads," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 11, pp. 1092-1102, Nov. 2005.

[6] D. Ghose, H.J. Kim, and T.H. Kim, "Adaptive Divisible Load Scheduling

Strategies for Workstation Clusters with Unknown Network Resources," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 10, pp. 897-907, Oct. 2005.

[7] K. Ko and T.G. Robertazzi, "Signature Search Time Evaluation in Flat File Databases," *IEEE Trans. Aerospace and Electronic Systems*, vol. 44, no. 2, pp. 493-502, Apr. 2008.

[8] D. Ghose and H.J. Kim, "Computing BLAS Level-2 Operations on Workstation Clusters Using the Divisible Load Paradigm," *Math. and Computer Modelling*, vol. 41, pp. 49-71, Jan. 2005.

[9] H.M. Wong and B. Veeravalli, "Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach," *IEEE Trans. Information Technology in BioMedicine*, vol. 9, no. 4, pp. 489-501, Dec. 2005.

[10] B. Veeravalli, and G. Barlas, "Efficient scheduling strategies for processing multiple divisible loads on bus networks". *Journal of Parallel and Distributed Computing*, vol. 62, pp. 132-151, 2002.

[11] L. Marchal, Y. Yang, H. Casanova and Y. Robert, "Steady-State Scheduling of Multiple Divisible Load Applications on Wide-Area Distributed Computing Platforms," *The International Journal of High Performance Computing Applications*, vol. 20, no. 3, 2006.

[12] M. Drozdowski, M. Lawenda and F. Guinand, "Scheduling Multiple Divisible Loads," *The International Journal of High Performance Computing Applications*, vol. 20, no.1, pp. 19-30, 2006.

[13] M. Drozdowski, M. Lawenda, "Scheduling multiple divisible loads in homogeneous star systems," *Journal of Scheduling*, vol. 11, no. 5, pp. 347-356, Oct. 2008.

[14] B. Veeravalli, H.F. Li, and T. Radhakrishnan, "Scheduling Divisible Loads in Bus Networks with Arbitrary Processor Release Times," *Computers and Math. with Applications*, vol. 32, no. 7, 1996.

[15] B. Veeravalli and H.M. Wong, "Scheduling Divisible Loads on Heterogeneous Linear Daisy Chain Networks with Arbitrary Processor Release Times," *IEEE Trans. Parallel and Distributed Systems,* vol. 15, no. 3, pp. 273-288, Mar. 2004.

[16] B. Veeravalli, and G. Barlas, "Scheduling Divisible Loads with Processor Release Times and Finite Size Buffer Capacity Constraints," *Special Issue on Divisible Load Scheduling in Cluster Computing*, Kluwer Academic Publishers, vol. 6, no. 1, pp. 63-74, Jan. 2003.

[17] X. Lin, A. Mamat, Y. Lu, J. Deogun, and S. Goddard, "Real-time scheduling of divisible loads in cluster computing environments," *Journal of Parallel and Distributed Computing*, vol. 70, no. 3, Mar. 2010.

[18] X. Lin, Y. Lu, J. Deogun, and S. Goddard, "Real-time divisible load scheduling with different processor available times," In *ICPP*, 2007.

[19] S. Chuprat and S. Baruah, "Scheduling Divisible Real-Time Loads on Clusters with Varying Processor Start Times," In *RTCSA*, 2008.

[20] A. Mamat, Y. Lu, J. Deogun, and S. Goddard, "Real-Time Divisible Load Scheduling with Advance Reservations," In *ECRTS*, 2008.

[21] X. Lin, Y. Lu, J. Deogun, and S. Goddard. "Multi-Round Real-Time Divisible Load Scheduling for Clusters," Proceedings of the 15th International Conference on High Performance Computing. (2008): 196-207.

[22] Y.C. Cheng, and T.G. Robertazzi, "Distributed Computation with Communication Delays," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 24, no. 6, pp. 700-712, 1988.

[23] R. Agrawal and H.V. Jagadish, "Partitioning Techniques for Large-Grained Parallelism," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1627-1634, 1988.

[24] B. Veeravalli, "Distributed Computation with Communication Delays: Design And Analysis of Load Distribution Strategies", PhD diss., Indian Institute of Science, Bangalore, India 1994.

[25] Bharadwaj, V., D. Ghose, and T. G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems", Cluster Computing, Kluwer Academic Publishers, vol. 6, no. 1, pp. 7-18, Jan. 2003.

[26] T.G. Robertazzi, "Ten Reasons to Use Divisible Load Theory," *Computer*, vol 36, no. 5, May 2003.

[27] Cluster Computing, Special Issue on Divisible Load Scheduling, D. Ghose and T. Robertazzi, eds., 2003.

[28] S. Bataineh, and T.G. Robertazzi, "Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job," *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1184-1196, Oct. 1994.

[29] J. Sohn, and T.G. Robertazzi, “Optimal Divisible Job Load Sharing for Bus Networks,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 32, no. 1, pp. 34-40, Jan. 1996.

[30] T.G. Robertazzi, “Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors,” *IEEE Transactions on Aerospace and Electronic System-s*, vol. 29, pp. 1216-1221, Oct. 1993.

[31] V. Mani, and D. Ghose, “Distributed Computation in Linear Networks: Closed-form solutions,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 30, pp. 471-483, 1994.

[32] Y.C. Cheng, and T.G. Robertazzi, “Distributed Computation for a Tree Network with Communication Delays”, *IEEE Transactions on Aerospace and Electronic Systems*, vol. 26, no. 3, pp. 511-516, 1990.

[33] S. Bataineh, T. Hsiung, and T.G. Robertazzi, “Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job”, *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, Ill., Aug. 1993.

[34] G. Barlas, “Collection-Aware Optimum Sequencing of Operations and Closed-Form Solutions for the Distribution of a Divisible Load on Arbitrary Processor Trees,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 5, pp. 429-441, May 1998.

[35] J. Blazewicz, and M. Drozdowski, “The Performance Limits of a Two- dimensional Network of Load Sharing Processors,” *Foundations of Computing and Information Sciences*, vol. 21, no. 1, pp. 3-15, 1996.

[36] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram, “Scheduling a Divisible Task in a Two-dimensional Mesh,” *Discrete Applied Mathematics*, vol. 94, no. 1-3, pp. 35-50, June 1999.

[37] Drozdowski, M., and W. Glazek, “Scheduling Divisible Loads in a Three-dimensional Mesh of Processors”, *Parallel Computing*, vol. 25, no. 4, pp. 381-404, Apr. 1999.

[38] K. Li, “Improved Methods for Divisible Load Distribution on k-Dimensional Meshes using Pipelined Communications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 12, pp. 1250-1261, Dec. 2003.

[39] Y. Chang, J. Wu, C. Chen, and C. Chu, “Improved Methods for Divisible Load Distribution on k-Dimensional Meshes Using Multi-Installment,” *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 11, Nov. 2007.

[40] J. Yao, and B. Veeravalli, “Design and Performance Analysis of Divisible Load Scheduling Strategies on Arbitrary Graphs,” *Cluster Computing*, vol. 7, no. 2, pp.841-865, 2004.

[41] D. England, B. Veeravalli, and J. Weissman, “A Robust Spanning Tree Topology for Data Collection and Dissemination in Distributed Environments”, *IEEE Transaction on Parallel and Distributed System*, vol.18, no.5, pp.608-620, 2007.

[42] J. Blazewicz, and M. Drozdowski, “Distributed Processing of Divisible Jobs with Communication Startup Costs,” *Discrete Applied Mathematics*, vol. 76, no. 1-3, pp. 21-41, June 1997.

[43] B. Veeravalli, X. Li, and C.C. Ko, "On the Influence of Start-up Costs in Scheduling Divisible Loads on Bus Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 12, pp. 1288-1305, Dec. 2000.

[44] M. Drozdowski and P. Wolniewicz, "Optimum divisible load scheduling on heterogeneous stars with limited memory," *European Journal of Operational Research*, vol. 172, no. 2, pp. 545-559, 2006.

[45] A. Legrand, Y. Yang and H. Casanova, "NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with AąÀne Cost-s," Technical Report CS2005-0818, CSE, UCSD, 2005.

[46] Bharadwaj, V., D. Ghose, and V. Mani,"Multi-Installment Load Distribution Strategy for Linear Networks with Communication Delays", *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, Dec. 26-29 1994.

[47] B. Veeravalli, D. Ghose, and V. Mani, "Multi-installment Load Distribution in Tree Networks With Delays," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 31, no. 2, pp. 555-567, April 1995.

[48] D. Altilar and Y. Paker, "An Optimal Scheduling Algorithm for Parallel Video Processing," *Proc. IEEE Intl Conf. Multimedia Computing and Systems*, 1998.

[49] D. Altilar and Y. Paker, "Optimal scheduling algorithms for communication constrained parallel processing," *Euro-Par* 2002.

[50] O. Beaumont, A. Legrand, and Y. Robert, "Scheduling divisible workloads on heterogeneous Platforms", *Parallel Computing*, vol. 29, pp. 1121-1152, Sep. 2003.

[51] P. Wolniewicz, “Multi-installment Divisible Job Processing with Communication Startup Cost”, *Foundations of Computing and Decision Sciences*, vol. 27, no.1, 43-57, 2002.

[52] Y. Yang and H. Casanova, “RUMR: Robust Scheduling for DivisibleWorkloads,” *IEEE International Symposium on High Performance Distributed Computing*, pp. 114-123, Jun 2003.

[53] N. Loc and S. Elnaffar, “A Dynamic Scheduling Algorithm for Divisible Loads in Grid Environments,” *Journal OF Communications*, vol. 2, no. 4, June 2007.

[54] M. Moges and T.G. Robertazzi, “Wireless Sensor Networks: Scheduling for Measurement and Data Reporting,” *IEEE Trans. Aerospace and Electronic Systems*, vol. 42, no. 1, pp. 327-340 Jan. 2006.

[55] B. Veeravalli, and S. Ranganath, “Theoretical and Experimental Study on Large Size Image Processing Applications using Divisible Load Paradigm on Distributed Bus Networks,” *Image and Vision Computing*, vol. 20, no. 13-14, pp. 917-935, Dec. 2002.

[56] T.T. Chin, B. Veeravalli, and J. Jia, “Handling Large-Size Discrete Wavelet Transform on Network-Based Computing Systems: Parallelization via Divisible Load Paradigm”, *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 143-152, Feb. 2009.

[57] K. Van, Y. Yang, and Casanova, H., “Practical Divisible Load Scheduling on Grid Platforms with APST-DV,” *Parallel and Distributed Processing Symposium*, 2005.

[58] D. Ghose, H. J. Kim, and T. H. Kim, "Agent-Based Load Balancing on Homogeneous Minigrids: Macroscopic Modeling and Characterization ", *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 10, pp. 897-907, Oct. 2005.

[59] B. Veeravalli, and G. Barlas, "Access Time Minimization for Distributed Multimedia Applications," *Special Issue in Multimedia Tools and Applications*, vol. 12, no. 2/3, Nov. 2000.

[60] G. Barlas and B. Veeravalli, "Optimized Distributed Delivery of Continuous Media Documents over Unreliable Communication Links," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 10, pp. 982-994, Oct. 2005.

[61] E. Balafoutis, M. Paterakis, P. Triantafillou, G. Nerjes, P. Muth, and G.Weikum, "Clustered Scheduling Algorithms for Mixed-Media Disk Workloads in a Multimedia Server," *Cluster Computing*, vol. 6, no. 1, pp. 75-86, Jan. 2003.

[62] P. Li, B. Veeravalli and A.A. Kassim, "Design and Implementation of Parallel Video Encoding Strategies Using Divisible Load Analysis," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 15, no. 9, pp. 1098-1112, Sep. 2005.

[63] J. Xu and L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.*, vol. 16, no. 3, pp. 360-369, Mar. 1990.

[64] K. Ramamritham, "Allocation and Scheduling of Precedence- Related Peri-

odic Tasks," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412-420, Apr. 1995.

[65] K. Ramamritham, J.A. Stankovic, and P.F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184- 194, Apr. 1990.

[66] M.L. Dertouzos and A.K. Mok, "Multiprocessor On-Line Scheduling of Hard Real-Time Tasks,ąś" *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1,497-1,506, Dec. 1989.

[67] G. Manimaran and C. Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. Parallel Distrib. Systems.* vol. 9 pp. 312ĺC319, Mar. 1998.

[68] J. Stankovic, M. Spuri, M. Natale, and G. Butazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computers*, vol. 28, pp. 16-25, June 1995.

[69] A. Burchard, J. Liebeherr, Y. Oh, and S. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. Computers*, vol. 44 pp. 1429ĺC1442, Dec. 1995.

[70] D. Babbar and P. Krueger, "On-Line Hard Real-Time Scheduling of Parallel Tasks on Partitionable Multiprocessors," Proc. Intąŕl. Conf. Parallel Processing, vol. 2, pp. 29-38, 1994.

[71] W.Y. Lee, S.J. Hong, J. Kim, "On-line scheduling of scalable real-time tasks on multiprocessor systems," *Journal of Parallel and Distributed Computing* 63 (12) (2003) 1315 - 1324.

[72] Nikolaos Doulamis, Emmanouel Varvarigos1, and Theodora Varvarigou, "Fair Scheduling Algorithms in Grids," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1630-1648, Nov. 2007.

[73] M. Eltayeb, A. Dogan, and F. Ozguner, "A data scheduling algorithm for autonomous distributed real-time applications in grid computing," in *Proc. ICPP*, Montreal, Canada, 2004, pp. 388-395.

[74] M.A.S. Netto and R. Buyya, "Offer-based scheduling of deadline-constrained Bag-of-Tasks applications for utility computing systems," in *Proc. IPDPS*, 2009, pp.1-11.

[75] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, second ed. Addison Wesley, 2003.

[76] O.H. Ibarra and C.E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *J. ACM*, vol. 24, no. 2, pp. 280-289, 1977.

[77] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Proc. Eighth IEEE Heterogeneous Computing Workshop*, pp. 30-44, 1999.

[78] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," *Proc. Ninth Heterogeneous Computing Workshop*, pp. 349-363, May 2000.

[79] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima, "Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids,"

Proc. 10-th Workshop Job Scheduling Strategies for Parallel Processing, pp. 210-232, 2004.

[80] N. Fujimoto and K. Hagihara, "Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid," Proc. International Conference on Parallel Processing, pp. 391-398, 2003.

[81] W. Cirne, F. Brasileiro, D. Paranhos, L.F.W. Gı́őes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213-234, April 2004.

List of Publications

[1] Hu Menglan and Bharadwaj Veeravalli, “Requirement-Aware Strategies for Scheduling Multiple Divisible Loads with Arbitrary Processor Release Times”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 10, pp. 1697-1704, 2011.

[2] Hu Menglan and Bharadwaj Veeravalli, “Scheduling Hybrid Tasks on Clusters”, *IEEE ICON 2011*, Singapore.

[3] Hu Menglan and Bharadwaj Veeravalli, “Requirement-Aware Strategies for Scheduling Real-Time Divisible Loads on Clusters”, submittied to *Journal of Distributed and Computing.*

[4] Hu Menglan and Bharadwaj Veeravalli, “Requirement-Aware Scheduling of Bag-of-Tasks Applications on Grids with Dynamic Resilience”, submitted to *IEEE transactions on Computers.*

[5] Hu Menglan and Bharadwaj Veeravalli, “Dynamic Scheduling of Hybrid Real-Time Tasks on Clusters”, submitted to *IEEE Transactions on Aerospace and Electronic Systems.*