DESIGN AND ANALYSIS OF LOAD BALANCING/SCHEDULING STRATEGIES ON DISTRIBUTED COMPUTER NETWORKS USING VIRTUAL ROUTING APPROACH

ZENG ZENG

(*M. Eng., Huazhong University of Science and Technology, PRC*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

Firstly of all, I would like to express my deepest gratitude and appreciation to my supervisor, Assistant Professor Bharadwaj Veeravalli, for his continuous guidance, constant encouragement and rigorous research attitude during the course of my research. It is a pleasant time to work with him during the past three year and he has made my research experience at the National University of Singapore (NUS) an invaluable treasure for my whole life.

My special thanks to my devoted parents for their love, encouragement and support throughout my life. They always stand by my side and provide me the safest harbor in the world. As the only child of the family, I have done little for them and own them too much.

My thanks also go to all my friends in Open Source Software Lab, in NUS, for their help and support to solve the technical and analytical problems. The friendship with them makes my study and life in NUS fruitful and unforgettable.

Finally, I would like to thank NUS for granting me the research scholarship and providing me the facilities that make the research a success.

Contents

Acknowledgements **i**

Contents **ii**

List of Figures **v**

List of Tables **viii**

Summary **ix**

1 Introduction **1**

1.1 Related Work 4

1.2 Issues to Be Studied and Main Contributions 7

1.3 Organization of the Thesis 11

2 System Modelling **13**

2.1 Models for Processing Loads 13

2.2 Arbitrary Network Topology 15

2.3 Mathematical Models and Some Definitions 15

2.3.1 Processor models 16

2.3.2 Communication link models 17

2.3.3 Some notations and definitions 18

2.3.4 Correspondence between routing and load balancing 19

2.4 Concluding Remarks 20

3 Distributed Static Load Balancing Strategies for Multi-class jobs 21

3.1 Problem Formulation 23

3.2 Proposed solution 27

3.3 Proposed Algorithm and An Optimal Solution 29

3.3.1 Optimal solution 29

3.3.2 Design of the proposed algorithm 34

3.3.3 Rate of convergence 38

3.4 Experimental Results and Discussions 40

3.4.1 LK algorithm in brief 41

3.4.2 Demonstration of LBVR algorithm: An example of load balancing . . . 42

3.4.3 Studies on more network topologies 49

3.5 Concluding Remarks 53

4 Distributed Dynamic Load Balancing Strategies 55

4.1 System Model and Classification of Dynamic Load Balancing Algorithms . . . 57

4.2 Comparative Study on the Algorithms 60

4.2.1 ELISA: Estimated Load Information Scheduling algorithm 61

4.2.2 The proposed algorithm: RLBVR 61

4.2.3 The proposed algorithm: QLBVR 67

4.3 Performance Evaluation and Discussions 69

4.3.1 Two-processor system model and some important issues 70

4.3.2 Effect of system loading 72

4.3.3 Effect of T_s: Length of status exchange interval 74

4.4 Extensions to Large Scale Multiprocessors System 77

4.4.1 Static or slowly varying system loading 80

4.4.2 Experiments when arrival of loads is varying rapidly 82

4.5 Concluding Remarks 84

5 Extensions to Divisible Loads Scheduling on Arbitrary Networks 86

5.1 Mathematical Model and Problem Formulation 88

5.1.1 Description of system, assumptions and notations 88

5.1.2 Problem formulation 91

5.2 Proposed Strategy for Optimal Solution 93

5.2.1 Sub-algorithm for a node and optimal sequence 93

5.2.2 Scheduling strategy for an arbitrary topology 98

5.2.3 Convergence and complexity of the algorithm 106

5.3 Simulation Results and Some Discussions 111

5.3.1 Demonstration: An example of optimal scheduling 112

5.3.2 Performance comparison of algorithms 114

5.3.3 Divisible loads originating from multiple sites 119

5.4 Concluding Remarks 121

6 Conclusions and Future Work 124

Bibliography 129

Author's Publications 140

Appendix: Queue Length Estimation 141

List of Figures

1.1 A distributed/parallel computer system. 2

2.1 A distributed/parallel computer system. 15

2.2 Server model of node i. 16

3.1 Job flows in node i. 24

3.2 Example of job flows and the delays. 26

3.3 Job flows in a system with a virtual node. 27

3.4 Routing paths for each node. 30

3.5 Operation of the proposed algorithm. 40

3.6 An example of a 9-node distributed computer system. 43

3.7 A distributed computer system with a virtual node d. 47

3.8 Comparison of the algorithms with respect to computational time. 47

3.9 An example of a 9-node Ring computer system. 50

3.10 Comparison of algorithms on Ring network. 50

3.11 An example of a 9-node arbitrary network. 51

3.12 Comparison of algorithms on an arbitrary network. 52

4.1 Node model for queue adjustment policy. 58

4.2 Node model for rate adjustment policy. 59

4.3 Node model for the combination of queue and rate adjustment policy. 60

4.4 Intervals of estimation and status exchange. 61

4.5 Model of a 2-processor system. 70

4.6 Job arrival rate pattern. 71

4.7 Effect of system loading. 73

4.8 Effect of T_s: System loading is light. 75

4.9 Effect of T_s: System loading is moderate. 76

4.10 Effect of T_s: system loading is high. 77

4.11 A mesh-connected multiprocessor $M[8, 8]$ system. 78

4.12 Mean response time of jobs for 5 different algorithms under different system utilization: System utilization is light or moderate ($\rho < 0.75$). 81

4.13 Mean response time of jobs for 5 different algorithms under different system utilization: System utilization is high ($\rho > 0.75$). 81

4.14 An example of the job pattern. 83

5.1 An arbitrary computer network system with multiple loads. 89

5.2 Single-level tree topology. 93

5.3 Single-level tree with virtual node and virtual links. 96

5.4 Results of Example 5.1: d_1^{max} and $d_1^{max} - d_1^{min}$ in each iteration r. 96

5.5 The iteration procedure of the proposed algorithm. 109

5.6 Experiment 5.1: (a) An arbitrary network; (b) Optimal sequence of node 1; (c) Optimal sequence of node 5. 112

5.7 Results of Experiment 5.1: $T^{max(r)}$ and $T^{max(r)} - T^{min(r)}$ in each iteration r. . . 114

5.8 Results of Experiment 5.1: Timing diagram. 115

5.9 Experiment 5.2: An arbitrary network with 10 nodes. 116

5.10 Results of Experiment 5.2: Timing diagram of the proposed algorithm when divisible loads originating from node 10. 117

5.11 Example 5.2, load flow: (a) A simple example with three nodes; (b) MST with root node 1; (c) The single-level tree; (d) Our proposed algorithm. 118

5.12 Experiment 3, a 9 nodes system: (a) A sparse connection; (b) A medium dense connection; (c) A dense connection. 120

5.13 The timing diagram of the 9-node system when the connection is medium dense. 121

List of Tables

3.1 Proposed Load Balancing Algorithm (LBVR) 39

3.2 Parameter values of node model 45

3.3 Average external job arrival rates for class-1 and class-2 jobs 46

3.4 Completed job rate (β_i^k) in node i 48

3.5 Parameters of each node in the system 52

4.1 Main structure of ELISA 62

4.2 Procedure for Node 68

4.3 MRT of 5 algorithms in the 8×8 mesh multiprocessor system (sec.) 84

5.1 Brute-force search results for optimal sequence of Example 5.1 98

5.2 Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 1) 104

5.3 Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 2) 105

5.4 Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 3) 106

5.5 Computational results for load distribution with different load origination . . . 116

5.6 Computational results of Experiment 5.3 (unit load) 120

Summary

Parallel and distributed heterogeneous computing has been proven to be an efficient and successful way for various applications. There are several performance metrics to quantify the performances of a distributed system. In this thesis, we consider the problem of load balancing in distributed systems. Specifically, we consider balancing indivisible loads across the network nodes so as to achieve an optimal response time. We first present the underlying mathematical model that takes into account several complex and influencing real-time scenarios of load balancing and scheduling. In this thesis, we attempt to employ a novel idea in which we use the concept of virtual routing for balancing the work loads among the nodes. This is the first time in this domain such an attempt is made. For each of the real-life scenarios considered, the problem is carefully decomposed into sub-problems and distributed strategies by virtual routing approach are derived systematically.

For indivisible jobs, minimizing the mean response time of the jobs submitted for processing is a critical performance metric to be considered for improving the overall performance of the distributed computer system. In this thesis, we propose both the *static* and *dynamic* load balancing algorithms for handling *single-class* or *multi-class* jobs in the distributed network system for minimizing the mean response time of the jobs, using the concept of virtual routing. We employ a novel approach to transform the load balancing problem into an equivalent routing problem and propose a static algorithm, referred to as *Load Balancing via Virtual Routing* (LBVR). We show that the design of LBVR subsumes several interesting properties

and guarantees to deliver a *super-linear* rate of convergence in obtaining an optimal solution, whenever it exists.

We classify the distributed, dynamic load balancing algorithms into three policies: *queue adjustment policy* (QAP), *rate adjustment policy* (RAP) and *Queue and Rate Adjustment Policy* (QRAP). On the basis of LBVR, we propose two efficient dynamic algorithms, referred to as *Rate based Load Balancing via Virtual Routing* (RLBVR) and *Queue based Load Balancing via Virtual Routing* (QLBVR), which belong to the above RAP and QRAP policies, respectively. Our focus is to analyze and understand the behaviors of these algorithms in terms of their load balancing abilities under varying load conditions (light, moderate, or high) and the minimization of mean response time of the jobs. We compare the above classes of algorithms by a number of rigorous simulation experiments to elicit their behavior under some influencing parameters such as, load on the system and status exchange intervals. We also extend our experimental verification to large scale multiprocessor systems such as Mesh architecture that is widely used in real-life situations. Recommendations are drawn to prescribe the suitability of the algorithms under various situations.

Finally, we extend our analysis and design of algorithms to the case of scheduling large volume computational loads (divisible loads) originating from single or multiple sites on arbitrary networks. It is first time in divisible load theory (DLT) that such a generalized mathematical model is presented and the scheduling problem is formulated as an optimization problem with an objective to minimize the processing time of the loads. We present a number of theoretical results on the solution of the optimization problem. On the basis of these results, we propose an efficient algorithm for scheduling divisible loads using the concept of load balancing via *virtual routing* for an arbitrary network configuration. When divisible loads originate from single node, we compare the proposed algorithm with a recently proposed RAOLD algorithm in the literature which is based on minimum cost spanning tree. When divisible loads originate from multiple sites, we testify the performance on sparse, medium and densely connected

networks. Detailed performance analysis and comparison are conducted.

Further study in the research areas of indivisible load balancing and divisible load scheduling are quite promising. Several possible extensions of our research are addressed at the end of this thesis.

Chapter 1

Introduction

Distributed computer systems have emerged as a powerful computing means for real-time applications, such as nuclear plant control and avionic control [1], image feature extraction [2] and biological sequence alignment [3], etc. We consider a generic distributed/parallel computer system shown in Fig. 1.1. The system consists of n heterogeneous nodes, which represent host computers, interconnected by a generally configured communication/intercommunication network. Each processor in the system may receive one or more classes of jobs independently and each node consists of one or more resources (such as CPU, I/O devices, etc), contended for by the jobs processed at that node. Further, these nodes may differ in configurations such as, speed characteristics, buffer sizes, and number of resources. However, we assume that they have the same processing capabilities. For instance, a job can be processed at any node without interruption. Compared to a single computer system, a distributed computer system generally provides significant advantages, such as better performance, better scalability, better reliability and better resource sharing [4], and distributed computer systems have attracted more and more research efforts in the past two decades [5–11].

Balancing or scheduling the work loads over a distributed computer network system is important to improve the overall performance. In such a system, if some hosts remain idle while others are extremely busy, system performance will be affected drastically. To prevent this,

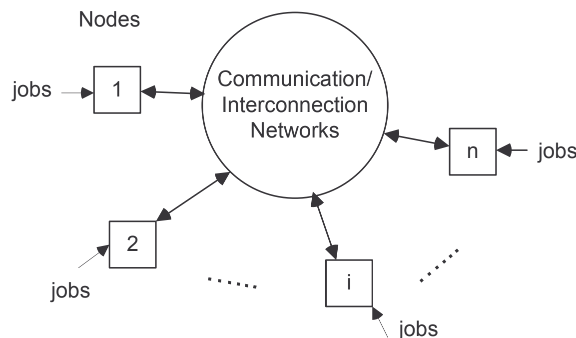

Figure 1.1: A distributed/parallel computer system.

load balancing and *load scheduling* are often used to distribute the loads and improve performance measures such as the mean response time (MRT) of a job, which is the time difference between the time instant at which a job arrives at the system and the time instant at which the job gets processed [12,13], the total time of processing all the loads [14,15]. The design of such load balancing and load scheduling algorithms, in general, considers several influencing factors, such as the underlying network topology, communication network bandwidth at each processor in the system, etc. The computers in the system are also considered and they can be classified as either homogeneous with the same computing characteristics or heterogeneous with different processing capabilities, buffer sizes limited or infinite, etc. Furthermore, while considering job characteristics, there may exist several variations, such as priority assignment for jobs in processing, jobs with or without deadlines, etc. For convenience, in the rest of this thesis, we use load and job interchangeably.

Based on the types of loads under processing, load balancing or scheduling problems can be classified into two categories: indivisible load balancing and divisible load scheduling. Indivisible loads are atomic and cannot be divided into smaller sub-tasks, and have to be processed in its entirety on a processor. The indivisible jobs are assumed to arrive at each node according to an ergodic process (e.g., Poisson process). Each node determines whether jobs will be processed locally or transferred to another node for processing. Load balancing

strategies attempt to distribute the indivisible jobs to be processed, according to some optimal solutions, to make the whole system balanced. Divisible loads are data parallel loads that are arbitrarily partitionable amongst nodes of the network. In contrast to the indivisible loads model, divisible loads are assumed to be very large in size, homogeneous, and arbitrarily divisible in the sense that, each partitioned portion of the loads can be independently processed on any processor in the system [15, 16]. The theory of scheduling and processing of divisible loads, referred to as *Divisible Load Theory* (DLT), has stimulated considerable interest among researchers in the field of parallel and distributed systems since its origin in 1988 [17, 18]. Hence, load scheduling is the study of how to obtain an optimal fraction of a large divisible load for each node in a distributed computer system.

Load balancing algorithms can be classified as either *dynamic* or *static*, based on the information that can be used. A dynamic algorithm [19–23] makes its decision according to the current status of the system, where the status could refer to some types of information such as the number of jobs waiting in the queue, the current job arrival rate, the job processing rate, etc, at each processor. On the other hand, a static algorithm [5, 12, 24, 25] is carried out by a predetermined policy, without considering the status of the system. The primary concern in the research of DLT is to determine the optimal fractions of the entire loads to be assigned to each of the processors in such a way that the total processing time of the entire loads is a minimum. Compilation of all the research contributions in DLT until 1995 can be found in monographs [1, 15]. Two recent survey articles [16, 26] consolidate all the results until 2002.

Below, we shall discuss some related work in both load balancing and scheduling research areas.

1.1 Related Work

For studying the load balancing problems, many models of computer networks, processors and jobs have been developed. For example, the models of networks can be classified according to the topologies of the networks [27], such as star, ring and bus. The processors can be modelled as *M/M/1* queuing systems with a single or several queues to hold the jobs waiting for processing [28]. In the existing literature, several combinations of different models of computer network, processor and jobs are discussed and other issues such as sender-initiated strategies, receiver-initiated strategies are proposed for load balancing [29]. In sender-initiated policies, congested processors attempt to transfer jobs to lightly loaded processors. In receiver-initiated policies, lightly loaded processors search for congested processors from which jobs may be transferred. In [30, 31], the authors compared the performance of the two policies and found that in most situations, sender-initiated policies provided generally better performance than receiver-initiated policies [7]. An excellent compilation of most of the load balancing/sharing algorithms until 1992 can be found in [8].

Static load balancing algorithms are widely used in large-scale simulations [1], parallel program [32], etc. For static algorithms, there are some differences among the network configurations. In [5, 33], Tantawi and Towsley studied optimal static load balancing in star and bus network configurations [34]. On the basis of their work, Kim and Kameda [35] proposed two improved algorithms for load balancing in star and bus network configurations, respectively. Load balancing problems in star and tree network configurations with two-way traffic were studied in [36, 37]. In [5, 38], the algorithms proposed were concerned about an arbitrary network configuration and hence, became more applicable in a practical distributed computer system. However, the contributions mentioned above considered only a single class of jobs. In practice [25], the jobs in the system were divided into several classes and each class of jobs had its own priority. Kim and Kameda studied the optimal load balancing problem for multi-class jobs in bus configured distributed computer system [39]. In [12], Li and Kameda

proposed a load balancing algorithm for multi-class jobs in an arbitrary network. It is an important work to analyze the load balancing problems for multi-class jobs. The study of multi-class jobs makes the system more flexible to handle different classes of jobs and is a right step in generalizing the study.

Dynamic load balancing algorithms offer the possibility of improving load distribution at the expense of additional communication and computation overheads. In [23, 40], it was pointed out that the overheads of dynamic load balancing may be large, especially for a large heterogeneous distributed system. Hence, most of the research works in the literature focused on centralized dynamic load balancing [23, 41], in which an Management Station (M-Station)/Balancer kept checking the system status and balanced the arriving jobs among the processors by some strategies, such as Backfilling [42], Gang-Scheduling, and Migration [81], etc. By centralization, the M-Station/Balancer can handle most of the communication and computation overheads efficiently, and improve the system performance. However, the centralization limits the scalability of the parallel system and the trend of larger distributed computer system makes M-Station/Balancer to become the system bottleneck. Because of its scalability, flexibility and reliability, distributed dynamic load balancing offers more advantages than the centralized strategies, and thus has obtained more and more focuses recently [19, 43].

To realize a distributed working style, each processor in the system shall handle its own communication and computation overheads independently [10,12]. In order to reduce the communication overheads, Anand et. al., [19] proposed an estimated load information scheduling algorithm (ELISA) and Michael [44] analyzed the usefulness of the extent to which old information can be used to estimate the status of the system. In [45–47], the authors have proven the correctness using randomization techniques, leading to an exponential improvement in balancing the loads. To obtain optimal solutions among the system, the computation overheads remain still high. For example, Jie-Kameda algorithm needs more than 400 seconds (approx) and even a well-known FD algorithm [48] needs more than 10^5 seconds to solve a

generic case [12]. However, in this thesis, we propose an algorithm named load balancing via virtual routing (LBVR) and prove that the convergence rate of LBVR is *super-linear*. High convergence rate can reduce the computation overheads significantly.

Numerous studies have been conducted in the DLT literature and a criterion that is used to derive optimal solution is as follows. It states that in order to obtain an optimal processing time, it is *necessary and sufficient* that all the processors participating in the processing must stop at the same time instance. This condition is referred to as an *optimality principle* in the DLT literature and analytic proof can be found in [15, 49, 50]. In 1998, Barlas [51] presented an important result concerning an optimal sequencing in a tree network and it is one of the important studies that demonstrates the performance of the load distribution algorithm when result collection phase is included in the problem formulation. In [52], a multi-level tree is considered and it is assumed that the load distribution takes place concurrently from a source processor to all its immediate child processors. This is one of the earliest attempts in using a *multi-port* model of communication, in which all the incident links on a processor are concurrently used. The advantage of the multi-port model was also demonstrated in a two-dimensional mesh network [53]. In [54], load partitioning of intensive computations of large matrix-vector products in a multi-cast bus network was investigated.

To determine the ultimate speedup using DLT analysis, Li [55] conducted an asymptotic analysis for partitionable network topologies. Here speedup is the ratio of solution time on one processor to solution time on N processors and is thus a measure of achievable parallel processing advantage. Most recent studies focus on system dependent constraints such as, scheduling under finite buffer capacity constraints [56], estimating the processor and link speeds by some methods of probing and estimating [57], scheduling divisible loads on Mesh multiprocessor architectures [58], to quote a few. Also, the applicability of DLT concepts to schedule and process loads generated from large scale physics experiments (RHIC at BNL, USA) on computational grids were investigated in [59]. Finally, use of affine delay models for

communication and computation components for scheduling divisible loads were extensively studied in [60, 61], etc.

We observe that the evolution of indivisible load balancing begins from static situations, evolves to dynamic situations, and the evolution of indivisible load balancing and divisible load scheduling begin from particular network topology, evolve to arbitrary network configurations and now flourish to retrieval strategies in a variety of practical areas with real-life constraints. This thesis is an attempt to contribute proactive efforts and interesting research results to the prosperous and active research areas.

1.2 Issues to Be Studied and Main Contributions

The contributions in the thesis are multi-fold. We will discuss the main contributions systematically below. From the review of the existing literature in load balancing and DLT highlighted above, we observe that, up to now, most works in these research areas attempt to obtain closed-form solutions of the problems under consideration. In load balancing field, the researchers propose their system models and formulate the problems as optimization problems with constraints. In order to solve the optimization problems, *Lagrangian* functions are constructed according to the original functions and very complex mathematical closed-form equations are derived [12, 38]. In order to solve the lagrangian functions, the *Lagrangian multipliers* become the keys and some research methods, such as *Golden Section Search* [62], are used to obtain the exact values of the multipliers. It is a time-consuming procedure. In DLT, the closed-form solutions are obtained in various network topologies. For example, in [63], the authors considered the linear daisy chain networks with the constraints of the arbitrary processor release time, and proposed different solutions for different situations. However, in some real-life situations, such as scheduling of divisible loads originating from multiply sites, it is very hard to obtain the closed-form solutions. In the literature, some search methods

such as Genetic Algorithms (GA) [64,65], Ant Algorithms [66], Tubo Search [67], etc., have been proposed and it has been proven that the search methods can solve some complex problems efficiently. This thesis attempts to transfer the load balancing and scheduling problems into equivalent virtual routing problems first and then, use some search methods to obtain the optimal solutions to these problems, if they exist. Detailed analysis and discussions on studying the strategies are conducted with the consideration of practical constraints. In the following, the main contributions of this thesis on indivisible load balancing and divisible load scheduling are discussed respectively.

For static load balancing problem, we assume that the network configurations are arbitrary and there are several classes of jobs. The problem addressed here is closely related to the earlier works reported in [5,12,38], however, the key differences are as explained below. In [5,38], the formulated non-linear constrained optimization problem considered only one class of jobs and showed that the delay functions were indeed convex and increasing functions. Whereas, in [12] the delay functions were assumed to be convex and the proposed algorithm was proven to be faster than the standard FD algorithm [48], consuming larger amount of computations in carrying out certain inverse functions. Also, the formulated problem in this work considered the process of load distribution in a different manner. For instance, for each class of jobs and for each processor, say i, the neighboring processors were categorized into four different sets such that, processors in each set sent the jobs of this class to a processor i based on certain rules. The rationale for this may be driven from application needs.

In our formulation, we relax this assumption and we consider all the jobs of a class that arrive at node i as a cumulative amount regardless of their origin. Thus, in our model, each processor is considered as an unbiased resource capable of processing the submitted jobs. Also, the delay functions are considered as arbitrary non-linear functions for the analysis to be more generic. Of course, as a possible extension, we also analyze the performance when convexities of the delay functions are to be considered. As a solution approach, we propose a

novel methodology for the posed problem. We transform the problem into a routing problem and derive an optimal solution to the transformed problem. The correspondence between the load balancing problem and the routing problem is also discussed. Thus, in this thesis, we propose a *static*, *distributed* load balancing algorithm for *multi-class* jobs in distributed network system for minimizing the mean response time of the jobs, using the concept of virtual routing. Extensive simulations are conducted to demonstrate the significant advantages of our proposed algorithm.

In this thesis, according to the job assignment methods, we classify the distributed, dynamic load balancing algorithms into three policies: *Queue Adjustment Policy* (QAP), *Rate Adjustment Policy* (RAP) and *Combination of Queue and Rate Adjustment Policy* (QRAP). Based on LBVR, we propose two efficient algorithms referred to as Rate based Load Balancing via Virtual Routing (RLBVR) and Queue based Load Balancing via Virtual Routing (QLBVR). It is the first time in the literature that a distributed system can adjust its scheduling according to optimal solutions dynamically using RLBVR algorithm. We introduce an algorithm called Estimated Load Information Scheduling Algorithm (ELISA) and an algorithm named Perfect Information Algorithm (PIA) reported in the literature [19], for the purpose of continuity.

The algorithms ELISA and PIA belong to QAP whereas RLBVR and QLBVR belong to RAP and QRAP, respectively. We carry out large number of rigorous simulation experiments to capture and analyze the effect of time-varying loads and different lengths of time intervals on the algorithms. As our focus is to analyze and understand the behaviors of the algorithms in terms of their load balancing abilities, minimization of mean response time, in our rigorous simulation experiments, we consider a single-class of jobs for processing. One of our added considerations in this study is to gain an intuition regarding the relative metrics of the different approaches under consideration. We extend our simulations to a large scale multiprocessor system such as Mesh architecture that is of practical use in real-life applications.

Based on the mesh topology, many prototype and commercial multiprocessor systems, such as Paragon [85], have been built. Our contribution elicits certain important behaviors of the distributed dynamic load balancing algorithms that serve to quantify the performances under different situations. From the simulations, we observe that when system utilization is light or medium, RAP performs much better than QAP and QRAP with relatively longer status exchange interval, which means less communication overheads. When system utilization is very high ($\rho > 0.9$), QAP performs the best among the three load balancing policies with high communication overheads. When the system utilization changes rapidly, QRAP is suitable and can achieve good performance with moderate communication overheads.

For processing arbitrary divisible loads, we formulate the scheduling problem with divisible loads originating from single or multiple sites on arbitrary networks as a real-valued constrained optimization problem. We design a distributed scheduling strategy to achieve the optimal processing time of all loads in the system. In our proposed algorithm, each processor can determine the amount of loads that should be transferred to other processors and the amount of loads that should be processed locally, according to some local information. It is the first time that a distributed algorithm is attempted and proposed in the DLT literature. In all the earlier DLT literature, timing diagrams were used to precisely define the load distribution process. This timing diagram representation would be meaningful if one could easily conceive of strategies that address scheduling loads from single site, regardless of the underlying topology. However, when one needs to schedule multiple divisible loads originating from several sites, it is rather impossible to capture the load distribution process by a single timing diagram. The main difficulty lies in this approach would be to explicitly schedule several timing components such as computation and communication from one site to other sites, identifying which processor-link pairs are redundant [15], etc. Thus, in this thesis, we take a radically different approach to address this complex problem by carefully formulating as a generalized minimization problem, thus avoiding the need for a timing diagram

representation.

We derive a number of theoretical results on the solution of the optimization problem. In case divisible loads originate from single site, we compare our proposed algorithm with a recently proposed RAOLD algorithm [14]. It is demonstrated that the proposed algorithm performs better than RAOLD in terms of the processing time. We analyze the difference between the divisible loads originating from single and multiple sites in our proposed algorithm. When divisible loads originate from multiple sites on arbitrary networks, we prove that our proposed algorithm also can solve the scheduling problem efficiently by numerous simulation experiments.

The above contributions are novel to the load balancing and scheduling literature. Thus, the scope of this thesis is essentially in addressing all the above-mentioned issues by developing a strong theoretical framework and to evaluate the performance via rigorous simulation experiments.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2, we introduce the basic system models adopted in load balancing and scheduling fields, and the general definitions and notations used throughout this thesis.

In Chapter 3, we analyze the static load balancing problem for multi-class jobs on arbitrary networks. We design and conduct a distributed strategy via virtual routing to minimize the mean response time of all the jobs arriving at the system. We also demonstrate that the convergence rate of our proposed algorithm is *super-linear*.

In Chapter 4, we classify the distributed, dynamic load balancing algorithms in the literature into three categories. We propose two efficient distributed, dynamic algorithms. Through extensive simulation experiments, certain important behaviors of dynamic load balancing al-

gorithms are elicited.

In Chapter 5, we consider the scheduling problems of divisible loads originating from single or multiple sites. We present a generic mathematical model for this problem and formulated it as a real-valued constrained optimization problem. The *necessary and sufficient* conditions for the optimal solution are derived and a novel distributed strategy is proposed.

In Chapter 6, we conclude our research work up to now and envision the prospect extensions.

Chapter 2

System Modelling

A distributed computer system consists of a comprehensive set of components, such as processors, communication links, storage units, etc. In general, while modelling the system we consider only the essential components in order to understand and analyze the system performance. In this chapter, we shall give a brief introduction of our system models that are used in solving the problems concerned. The models are widely used and details can be found in [12, 15, 19, 68]. We present the terminology, definitions, and notations that are used throughout of the thesis. We use a novel approach – virtual routing, to solve the problems discussed in this thesis. We shall also discuss the correspondence between the routing and load balancing/scheduling problems.

2.1 Models for Processing Loads

As we mentioned before, computation data or loads (jobs), in general, can be classified into two categories, namely indivisible and divisible loads. Indivisible loads are independent loads, of different sizes, which cannot be further subdivided, and hence must be processed by a single processor. Balancing these loads are known to be NP-complete problems in the literature [12]. According to the jobs arrival patterns, indivisible load balancing can be classified into *static*

and *dynamic* situations. In static situation, the job arriving rates are constant and according to some ergodic processes, such as homogeneous Poisson process. The static load balancing strategies can be determined before the systems start processing and the optimal solutions can be obtained by the off-line calculations [35, 36]. In dynamic situation, the job arriving rates are changed rapidly and the load balancing strategies must change the balancing according to the current system information [22, 23]. These on-line strategies only yield sub-optimal solutions with some communication and computation overheads.

On the other hand, divisible loads are loads that can be divided into smaller portions and hence they can be distributed to more than one processors to achieve a faster overall processing time. Some large linear data files, such as those found in image processing, large experimental processing, and cryptography, to quote a few, are considered as divisible loads. Divisible loads can be further categorized into modularly and arbitrary divisible loads, based on the characteristic of the loads. Modularly divisible loads can only be divided into smaller fixed size loads, while arbitrary divisible loads can be divided into any smaller size loads.

In the real-life system, there are several classes of loads and each class of loads has its own priority. We assume there are m ($m \geq 1$) classes of jobs that can arrive to the distributed computer system for processing and we use J to denote the set of jobs. For convenience, we define class-1 has the highest priority, class-2 has the second highest priority, and, so on. Due to the homogeneous characteristic of the divisible loads, when divisible loads are considered, we assume $m = 1$ that means there is only one class of loads in field of DLT. For indivisible jobs, we assume that each class of jobs demands different processing rate, depending on the nature of the jobs. Further, In our model, we consider the non-preemptive priority rule whereby a job undergoing process in the node is allowed to complete process without interruption even if a job of higher priority arrives in the mean time.

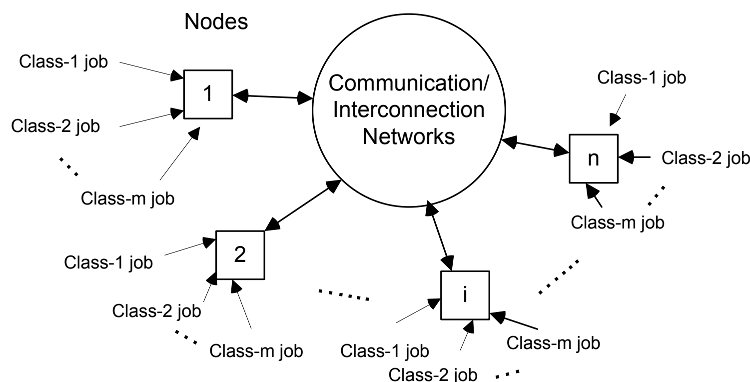

Figure 2.1: A distributed/parallel computer system.

2.2 Arbitrary Network Topology

Loads in the system can be transferred from one node to another through the communication links, according to the load balancing or scheduling strategies. As shown in Fig. 2.1, the processors in the system may be equipped with front-ends. Front-ends are actually co-processors which can handle the communication duties for the processors. Thus, a processor that has a front-end can communicate and compute at the same time. The communication links my be specified as some standard interconnection architecture, such as tree, bus, linear daisy chain, etc. In this thesis, we consider an arbitrary network configuration and attempt to obtain generic solutions for the different underlying network topologies.

2.3 Mathematical Models and Some Definitions

In general, the system models contain the essential components that we need to consider to the problems under studies. A system model shall clarify our objective, include the main entities in the system and model their characteristics [69]. In our model, we assume that there is a communication delay incurred when a class-k job is transferred from one node to the other through the communication link and there is a processing delay incurred when a class-k job

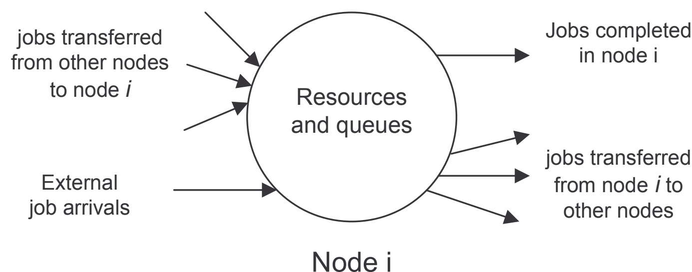

Figure 2.2: Server model of node i.

is processed on one node in the system. In DLT, the above two kinds of delay functions are assumed to be linear, increasing functions [15, 16, 70]. We adopt this model in solving the problems of divisible load scheduling concerned in this thesis. For the indivisible jobs, the processing delay and communication delay functions may be very complex and there are many delay models proposed for network and processor. Without loss of generality, we choose the queueing models which are widely used in the literature for the network and processor [12,27]. In the following, we will discuss the mathematical models of processor and communication links for the divisible and indivisible loads, respectively.

2.3.1 Processor models

The central server queuing model [28] is the common model for a host computer. In this thesis, the central server model is used as the node model as shown in Fig. 2.2. This model includes a queue that holds the incoming jobs, and a CPU that processes jobs according to FCFS discipline. Here, we use N to denote the set of nodes, i.e., $n = |N|$.

For indivisible job, for ease of simplicity, here we only present the node delay model of the system where there is only one class of jobs. The expected node delay of a job in such a node i model is given as:

$$T_i = \frac{1}{\mu_i - \beta_i}, \tag{2.1}$$

where, μ_i is the processing rate of node i and β_i is the current processing rate at node

i. For multi-class jobs, the node model is more complex and it will be discussed in detail in Chapter 3.

For divisible loads, the processors in the system are of different speeds. We denote the speed of a node i by E_i defined as:

$$E_i = \text{Time taken by node } i \text{ to process a unit load.} \tag{2.2}$$

Assume that the total divisible load in the system is L and the amount of loads is assigned to node i is l_i, then, the processing time for node i to process the assigned units loads is proportional to the amount of loads, which is $E_i \times l_i$.

2.3.2 Communication link models

For the communication link models, we use E to denote a set whose elements are unordered pairs of distinct elements of N. Each unordered pair $e = \ < i, j >$ in E is called an *edge*. For each edge $< i, j >$, we define two ordered pairs (i, j) and (j, i) which are called *links* and we denote C as the set of links. A node i is said to be a *neighboring node* of j, if i is connected to j by an edge. For a node j, let $V_j = \{i, \mid (i, j) \in C\}$ denote a set of neighboring nodes of node j.

Because the different characteristics of indivisible jobs and divisible load, for the communication link models, we consider the two cases respectively.

For indivisible jobs, in our model, we assume that there is a communication delay incurred when a class-k job is transferred from one node to the other in the system. For ease of simplicity, we assume that there is no difference among the classes of jobs as far as link model is concerned. Let x_{ij}^k be the class-k job flow rate from node i to node j, and $x_{ij} = \sum_{k=1}^{m} x_{ij}^k$, $m = |J|$, which is called the *total traffic of jobs* on link (i, j). For a job, the communication delay includes the time delay when the job is sent from node i to node j and the time delay when node j sends back a response to node i after the job has been processed.

For divisible load, communication link speed is modelled by the time taken for the individual link (i, j) to communicate a unit load. To describe the time performance of link (i, j), we shall define the time delay for communication, as:

$$C_{ij} = \text{Time taken to transmit a unit load through link } (i, j). \tag{2.3}$$

Now assume that the amount of total loads is L and the time delay of the loads transmitted through link (i, j) is proportional to the amount of loads l_{ij}, $(l_{ij} \leq L)$, i.e., $C_{ij} \times l_{ij}$. Such kind of linear models of processor speed and link speed is experimentally well supported by researchers in industry and academia [34, 51].

2.3.3 Some notations and definitions

We shall introduce some notations and definitions that are used throughout of this thesis.

N: The set of processors in the system.

n: The number of the processors.

V_i: The neighboring nodes of node i, where $i = 1, 2, \ldots, n$.

J: The set of jobs in the system and $m = |J|$.

k: Indicator of class-k jobs, where $k = 1, 2, \ldots, m$.

β_i^k: The job processing rate of class-k indivisible jobs at node i.

x_{ij}^k: The transfer rate of class-k indivisible jobs on link (i, j).

x_{ij}: The rate of indivisible loads or the amount of divisible loads transferred on link (i, j).

μ_i^k: The processing rate for class-k indivisible jobs at node i.

l_i: The amount of divisible load processed at node i.

E_i: Time taken by node i to process a unit divisible load.

C_{ij}: Time taken by link (i, j) to transfer a unit divisible load.

2.3.4 Correspondence between routing and load balancing

Below, we identify certain key equivalences between the two problems of routing and load balancing/scheduling in a systematic fashion.

Firstly, routing decisions can be made depending on whether the network uses datagrams or virtual circuits. In a datagram network, two successive packets of the same *S-D* pair (source and destination nodes) may travel along different routes, and a routing decision is necessary for each individual packet. In a virtual circuit network, a routing decision is made at the time of setting up a virtual circuit. The routing algorithm is used to choose the communication path for the virtual circuit. All packets of the virtual circuit subsequently use this path up to the time that the virtual circuit is terminated. There are two main performance measures that are substantially affected by a routing algorithm - *throughput* and *average packet delay.* In the routing context, the throughput is simply the amount of traffic from a source node to the destination node. In the load balancing problem, throughput is equivalent to amount of processed load by the system and an average packet delay is equivalent to the *mean response time* of jobs in the system. Thus, in our transformed problem defined by (3.4), a routing decision may aid to balance the loads that traverse via different nodes before reaching the destination d. Hence, a set of nodes in each path generated by a node i to reach d will be considered as potential receivers in our load balancing problem.

Secondly, another classification of routing algorithm relates to whether there is any route change in response to the traffic input patterns. In *static* routing algorithms [71], the path used by the sessions of each S-D pair is fixed regardless of traffic conditions. Further, in *adaptive* or *dynamic* routing [72, 73], the paths used to route new traffic between S-D pairs change occasionally in response to network congestion. For load balancing, static algorithms do not depend on the current state of the nodes in the system, and dynamic policies offer the possibility of improving load distribution at the expense of additional communication and processing overheads [40, 74].

In our problem context, by adding a virtual destination node d into our system, we combine the problems of routing and load balancing together as explained by the above correspondence.

2.4 Concluding Remarks

In this chapter, we introduced the mathematical models in load balancing and scheduling research domain. Some important notations and definitions that will be used frequently in the rest of this thesis were introduced.

Chapter 3

Distributed Static Load Balancing Strategies for Multi-class jobs

Minimizing the *mean response time* (MRT) of the jobs submitted for processing in a distributed computer system is a critical performance metric to be considered for improving the overall performance of the system. Load balancing algorithms thrive to meet this objective of minimizing the mean response time, the average time interval between the time instant at which a job is submitted and the time instant at which the job leaves the system after processing. Further, while considering job characteristics, there may exist several variations, such as priority assignment for jobs in processing, jobs with or without deadlines, etc. In this chapter, we consider static load balancing problems for multi-class indivisible jobs on arbitrary network configurations. Our objective is to design efficient static load balancing strategies that can minimize the mean response time of all classes of jobs arriving at the system for processing.

We attempt to formulate a static load balancing algorithm as a non-linear constrained optimization problem. Specifically, we consider the following real-life situation in our problem setting/definition. We consider a network of processors to which several classes of jobs arrive with a constant flow-rate for processing. Each processor may receive one or more classes

of jobs and considers the entire set of jobs submitted for processing to it as its total input loads. As with the principle of load balancing, jobs are allowed to migrate from heavily loaded processors to lightly loaded processors for minimizing the mean response time [36,38]. For each class of jobs, the communication delay is modelled as a non-linear function that depend on the network traffic, and the delays of jobs are different on different links [27, 75]. Consequently, the nature of this function, either as a convex or a non-convex, influences the optimality of the solution. Also, we assume that each class of jobs demands different processing rate, depending on the nature of the jobs [12, 25, 76]. All these influencing factors are captured as constraints in our optimization problem. Further, we consider a non-preemptive style of processing of the jobs at a processor, i.e., a job that is currently being processed cannot be interrupted by any other class of job for processing [28].

As a solution approach, we propose a novel methodology for the posed problem. We transform the problem into a routing problem and derive an optimal solution to the transformed problem. The correspondence between the load balancing problem and the routing problem is discussed. In our strategy, each node in the system will calculate the local optimal solutions independently, according to some information of its neighboring nodes and links. Thus, in this thesis, we propose a *static*, *distributed* load balancing algorithm for *multi-class* jobs in distributed network systems for minimizing the mean response time of a job, using the concept of virtual routing. We also prove that the convergence rate of our proposed algorithm is 2, *super-linear*.

The organization of the chapter is as follows. In Chapter 3.1, we formulate the problem. In Chapter 3.2, we discuss the solution via virtual routing approach. In Chapter 3.3, we propose our algorithm and derive conditions for obtaining an optimal solution. In Chapter 3.4, we report all our experimental results and present a detailed illustrative example to show the complete workings of our proposed algorithm for ease of understanding. We shall also present simulation study to quantify the performance in terms of rate of convergence and

solution quality. Finally, in Chapter 3.5, we conclude this work.

3.1 Problem Formulation

We consider a generic distributed/parallel computer system and we assume that there are m classes of jobs that can arrive to the system for processing as shown in Fig. 2.1. In our model, we consider the non-preemptive priority rule whereby a job undergoing process in the node is allowed to complete process without interruption even if a job of higher priority arrives in the mean time. When the node becomes free, the first job of the highest priority is considered for processing. Further, we assume that Class-k jobs arrive at node i, $i \in N$, according to an ergodic process, such as Poisson process, with the average external job arrival rate of ϕ_i^k. A class-k job arriving at node i may either be processed locally or transferred through the network to another node for remote processing. We denote β_i^k as the rate at which class-k jobs are processed at node i.

In general, computing on a network based environment incurs additional overheads such as communication delays due to the underlying links. In a generic treatment of a load balancing problem, it is imperative to account such additional overheads that influence the overall performance. We assume that each link (i, j) can transfer the load at its own transmission capability (otherwise referred to as transmission rate, commonly expressed as bytes/sec). We denote c_{ij} as the transmission capability of a link (i, j). In [27], there are many delay models proposed for data networks. Here, we choose M/M/1 model for the network. Let x_{ij}^k be the class-k job flow rate from node i to node j, and $x_{ij} = \sum_{k=1}^{m} x_{ij}^k$, $m = |J|$, which is called the *total traffic of the jobs* on link (i, j).

Note that in Jie Li and Hisao Kameda's model of multi-class jobs distributed computer system [12], the class-k jobs transferred from the neighboring nodes and external arriving class-k jobs are treated differently. In their model, for class$-k$ jobs and for each processor,

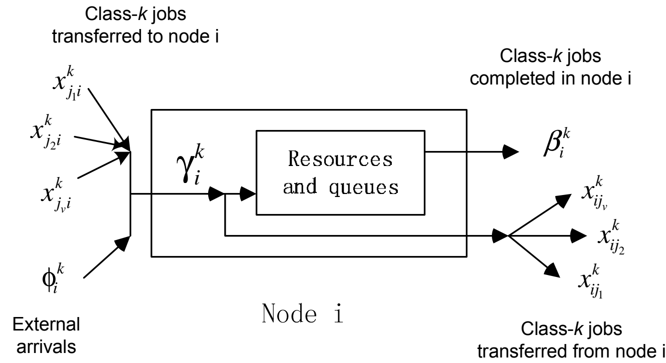

Figure 3.1: Job flows in node i.

the neighboring processors are divided into four different sets such that, processors in each set send the class$-k$ jobs to a processor i based on certain rules. However, in our model, we relax this assumption and consider all the jobs of class$-k$ that arrive at node i as a cumulative amount of class$-k$ type, regardless of their origin. Our model for a node is as shown in Fig. 3.1. From this figure, the following conservation equations hold:

$$\gamma_i^k \;=\; \phi_i^k + \sum_{j \in V_i} x_{ji}^k = \beta_i^k + \sum_{j \in V_i} x_{ij}^k, \tag{3.1}$$

where, γ_i^k is the total class-k jobs arriving rate at node i. Thus, we can say that in order to obtain load balancing, each node i must determine β_i^k and each x_{ij}^k $(j \in V_i)$, according to γ_i^k, satisfying the above equation (3.1). The *mean response time* (MRT) of a job in the system is also influenced by the mean nodal delay at the processing node in addition to a (possible) mean communication delay incurred during job transfer phase. Hence, the load balancing policy should determine the values of β and x, the job processing rate vector and the job transferring rate vector.

Further, we denote $P_i^k(\beta_i)$ as the mean node delay for a class-k job at node i and $G_{ij}^k(x)$ as the mean communication delay for a job transfer from node i to j. Some generic models of $G_{ij}^k(x)$ include the delay of sending a class-k job from node i to node j and the delay of sending the response back from node j to node i. In general, the path taken in each of the

above mentioned transfers (job and response transfers) may be different. Note that the profile of functions $P_i^k(\beta_i)$ and $G_{ij}^k(x)$ may be very complicated. In practice, for analytical ease, it is often assumed that the functions $P_i^k(\beta_i)$ and $G_{ij}^k(x)$ are differentiable, increasing and convex functions [5, 27, 62].

Let $D(\beta, x)$ denote the mean response time (MRT) of jobs averaged over all classes, which is the mean time a job spends in our system from the time of its arrival. As in [38], we obtain:

$$D(\beta, x) = \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{i \in N} \beta_i^k P_i^k(\beta_i^k) + \sum_{(i,j) \in L} x_{ij}^k G_{ij}^k(x_{ij}^k) \}, \tag{3.2}$$

where, $\Phi = \sum_{i \in N} \sum_{k \in J} \phi_i^k$, $\beta = [\beta_1, ..., \beta_i, ..., \beta_n]$, $x = [x_{ij}]$ with $\beta_i = [\beta_i^1, ..., \beta_i^m]$, and $x_{ij} = [x_{ij}^1, ..., x_{ij}^m]$, $m = |J|$.

Our objective is to balance the loads that arrive to our system such that the MRT (performance measure) is minimized. Note that the MRT is influenced by β_i^k and the transfer rates x_{ij}^k, in order to minimize the mean response time. Load balancing is done by transferring loads from heavily loaded nodes to lightly loaded nodes. However, in doing so a number of factors, such as job arrival rates, processing capability of a node, transmission capability of a link, etc must be taken into account. Also, one must avoid a situation wherein a load simply traverses across several nodes without getting processed. Other performance measures of interest, such as a weighted sum of mean response times relative to jobs entering at different nodes, can also be considered in a similar fashion. A simple example of a three-node system is shown in Fig. 3.2.

It may be noted that, although we only use MRT as the main objective in this chapter, one can consider other metrics to quantify the performance in our problem formulation. For example, if we consider a min-max metric besides MRT, we can keep track of the number of times a job has been transferred. Thus, if it exceeds a pre-defined maximum, the job will not be transferred any more. Hence, we can limit the number of transfers of the jobs within some scale. It is beyond the scope of this chapter to consider this metric. In this chapter, we shall formally define the problem that we want to address. In essence, we formulate the problem

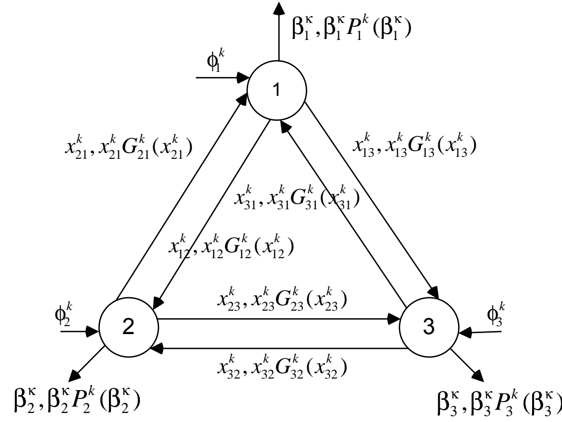

Figure 3.2: Example of job flows and the delays.

as a real-valued optimization problem with the objective of minimizing the MRT defined in (3.2). We state the following.

$$Minimize: \quad D(\beta, x) = \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{i \in N} \beta_i^k P_i^k(\beta_i^k) + \sum_{(i,j) \in L} x_{ij}^k G_{ij}^k(x_{ij}^k) \}, \tag{3.3}$$

$$\begin{aligned} Subject \;\; to: \quad \phi_i^k + \sum_{j \in V_i} x_{ji}^k = \beta_i^k + \sum_{j \in V_i} x_{ij}^k, k \in J, i \in N, \\ \phi_i^k \geq 0, k \in J, i \in N, \\ x_{ij}^k \geq 0, (i,j) \in L, k \in J. \end{aligned}$$

Thus, the solution to our problem lies in determining the optimal values of β and x, respectively. In all the earlier studies [5, 12, 62, 77], the solution to (3.3) is obtained by using the method of *Lagrangian multipliers* and key idea is to determine the set of *Lagrangian multipliers* to obtain an optimal solution. In order to find the Lagrangian multipliers, *linear section* searches are used, such as *Golden Section Search* [77], which needs fairly very long computational time. Below we shall present our solution approach.

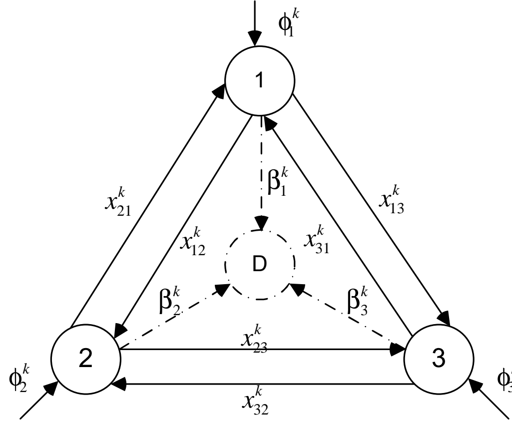

Figure 3.3: Job flows in a system with a virtual node.

3.2 Proposed solution

In this section, we propose our solution approach to the problem defined in (3.3). As a first step, we add a *virtual* node, which is referred to as the *destination* node (node d), into our network system. Also, we connect node d with each node i, $i \in N$, by a direct link (i, d). Note the virtual node d and each direct link (i, d) do not exist in the real system. The job flow in our "new" system is shown in Fig. 3.3. According to this modification, we redefine the set of links C in the system as $C = \{(i, j), \mid i, j \in N\} \cup \{(i, d), \mid i \in N\}$ and denote x^k_{id} as the class-k job flowing on link (i, d), which is equal to β^k_i. After these modifications, we introduce another function F^k_{ij} to unite the two different delay functions of P^k_i, the mean node delay of class-k jobs, and G^k_{ij}, the mean communication delay of class-k jobs. Function F^k_{ij} is expressed as the mean link delay of class-k jobs on link (i, j) as follows:

$$F^k_{ij}(x^k_{ij}) = \begin{cases} P^k_i(x^k_{ij}), (i, j) \in L \ and \ j = d, \\ \\ G^k_{ij}(x^k_{ij}), (i, j) \in L \ and \ j \neq d, \end{cases}$$

where, d is the virtual destination node. Then, (3.2) can be rewritten as:

$$\begin{aligned} D(\beta, x) &= \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{i \in N} \beta^k_i P^k_i(\beta^k_i) + \sum_{(i,j) \in L, j \neq d} x^k_{ij} G^k_{ij}(x^k_{ij}) \} \\ &= \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{i \in N} x^k_{id} F^k_{id}(x^k_{id}) + \sum_{(i,j) \in L, j \neq d} x^k_{ij} F^k_{ij}(x^k_{ij}) \} \end{aligned}$$

$$= \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{(i,j) \in L} x_{ij}^k F_{ij}^k(x_{ij}^k) \}. \quad (3.4)$$

Thus, from (3.4), we can describe the process of load balancing in another way. As shown in Fig. 3.3, the three-node system has been transformed into a datagram network, in which node i, $i \in N$, basically acts as a *router*. The way in which the loads are shared by the nodes can be described as follows. Class-k jobs that arrive at node i according to a Poisson process with an average external job arrival rate of ϕ_i^k are routed to destination node d via every node $j \neq i$. This can be understood from Fig. 3.3. Referring to this figure, we observe that for each node i, $i = 1, 2, 3$, there exist five paths to reach node d. For example, from node 1 to node d, the paths are: $1 \to 2 \to d, 1 \to 3 \to d, 1 \to 2 \to 3 \to d, 1 \to 3 \to 2 \to d, 1 \to d$, respectively. Thus, node i must determine a set of paths independently for the class-k jobs arriving at it via every other node to node d. Also, note that the class-k jobs may spend some time in the system due to the link delays through the path from node i to node d. In practice, it is reasonable to assume that the mean link delay $F_{ij}^k(x)$ of link (i, j) depends only on the job flow rates x_{ij} on link (i, j), where $x_{ij} = [x_{ij}^1, ..., x_{ij}^m]$, $m = |J|$. Thus, our goal of load balancing can be alternatively (and equivalently) stated as a problem which attempts to minimize the mean link delay for each job in the system and our method basically *transforms* the load balancing problem into a routing problem with all the jobs in the system having the same destination of node d. At the same time, we treat the job as a single entity travelling on the links. Notice that if function $D(x)$ is strictly convex, we have the unique solution to problem (3.4). Since we do not assume that $D(x)$ is strictly convex, the solution may not be unique. However, we can get an optimal solution that minimizes the mean response time $D(x)$ although it may not be a unique solution.

3.3 Proposed Algorithm and An Optimal Solution

Once the load balancing problem has been transformed into a routing problem, there are several algorithms proposed in the literature to solve such problems [27, 75, 78, 80]. However, in our problem context, it is important to establish the correspondence between the problems of routing and load balancing. Since our solution methodology proposes to use routing to balance the load in the network, we refer to our algorithm as *load balancing via virtual routing* (LBVR). As discussed in Chapter 2, we can clearly take advantage of the routing algorithms for load balancing. We shall first present the optimal solution below.

3.3.1 Optimal solution

Here we propose our algorithm by using some concepts underlying a routing problem.

When a class-k job arrives at node i, it has many routing paths to choose to reach the destination node d (as can be seen from Fig. 3.3). We can observe that if the number of nodes increases, the potential routing paths for node 1 will increase dramatically. However, when we consider the decentralized policy, each node i in the system only knows about the flow status of link $(i, j),\ j \in V_i$, its own load x_{id}^k and the loads of its neighboring nodes x_{jd}^k, respectively. Hence, the number of routing paths for node i can be reduced to $(v+1)$ where $v = |V_j|$. This means that a job arriving at a node i can reach node d directly through link (i, d), or it can go to node d via node $j,\ j \in V_i$, through link (i, j) and link (j, d). An example of paths for each node in a three-node system is shown in Fig. 3.4, where $p(i)_l$ means the l-th path of node i. Because of the distributed working style, each node in the system determines the routing paths according to its neighboring nodes and calculates its transfer rates and processing rate by itself. When node j receives some jobs from node i, node j may retransfer some jobs to another node, and so on. Eventually, the jobs come from node i may be processed in one node which is unknown to node i. However, it may be noted that this does not mean that

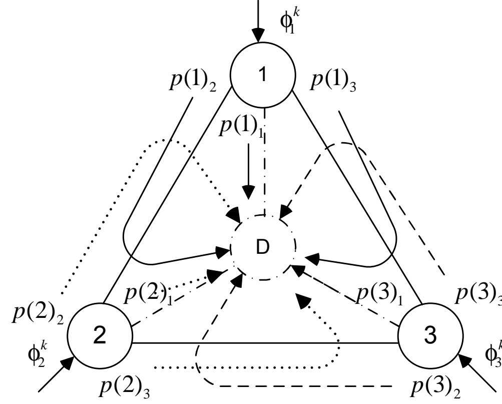

Figure 3.4: Routing paths for each node.

the search of eligible paths is restricted to the neighboring nodes, since some other routing paths with two or more hops can also be added according to the situations.

We denote P_i^k as the set of paths from node i to destination node d for class-k job and hence, we obtain $|P_i^k| = v + 1,\ v = |V_i|$. Further, let x_i^k be class-k job path flow vector of node i, where $x_i^k = \{x_p^k,\ |\ p \in P_i^k, i \in N\}$. Here, x_p^k is the job flow of path p for class-k job.

We define $D_{ij}^k(x_{ij}^k) = x_{ij}^k F_{ij}^k(x_{ij}^k)$ and refer to this as a *delay function*. Substituting this delay function in (3.4), we obtain:

$$D(x) = \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{(i,j) \in L} D_{ij}^k(x_{ij}^k) \}, \tag{3.5}$$

where, x_{ij}^k is the total amount of class-k job flow carried by link (i, j) and is given by:

$$x_{ij}^k = \sum_{\substack{all\ paths\ p \\ containing\ (i,j)}} x_p^k. \tag{3.6}$$

From (3.5), it is easy to observe that the function D is an increasing function. Now, we can completely rewrite the optimization problem as follows:

$$Minimize:\ D(x) = \frac{1}{\Phi} \sum_{k \in J} \{ \sum_{(i,j) \in L} D_{ij}^k(\sum_{\substack{all\ paths\ p \\ containing\ (i,j)}} x_p^k) \}. \tag{3.7}$$

$$Subject\ \ to: \quad \sum_{p \in P_i^k} x_p^k = \gamma_i^k,\ i \in N,\ k \in J \tag{3.8}$$

$$\gamma_i^k = \phi_i^k + \sum_{j \in V_i} x_{ji}^k, \ i \in N$$
$$x_p^k \geq 0, \ p \in P_i^k, \ i \in N.$$

In the following, we will characterize the proposed algorithm in terms of the first derivatives $(D_{ij}^k)'$ of the functions D_{ij}^k. We denote $D(x)$ as the mean response time function of (3.7) and $\partial D(x)/\partial x_p^k$ as the partial derivative of D with respect to x_p^k. Then:

$$\begin{aligned} \frac{\partial D(x)}{\partial x_p^k} &= \frac{1}{\Phi} \sum_{k \in J} \sum_{\substack{all\ links \\ (i,j)\ on\ path\ p}} (D_{ij}^k)' \\ &= \frac{1}{\Phi} \{ \sum_{\substack{all\ links\ (i,j) \\ on\ path\ p}} \frac{\partial D_{ij}^1(x_{ij}^1)}{\partial x_{ij}^k} + \cdots + \sum_{\substack{all\ links\ (i,j) \\ on\ path\ p}} \frac{\partial D_{ij}^m(x_{ij}^m)}{\partial x_{ij}^k} \}, \end{aligned} \tag{3.9}$$

where, the first derivatives $(D_{ij}^k)'$ are evaluated at a value equal to the total amount of class-k job flows corresponding to x^k and $m = |J|$. It is seen that $\partial D/\partial x_p^k$ is the length of path p when the length of each link (i,j) is taken to be the first derivative $(D_{ij}^k)'$ evaluated at x_p^k. Consequently, in what follows $\partial D/\partial x_p^k$ is called the *first derivative length of path p*.

Theorem 3.1. *The set of values of $\bar{x}$ is an optimal solution to problem (3.7)* **only if** *job flow travels along minimum first derivative length (MFDL) paths for each S-D pair of class-k job, $k \in J$. In addition, if D_{ij}^k are assumed to be convex, then $\bar{x}$ is optimal* **if and only if** *job flow travels along MFDL for each S-D pair of class-k jobs, $k \in J$.*

Proof. We shall first prove the first part of the above theorem. Let $\bar{x} = \{\bar{x}^1, ..., \bar{x}^k\}, \ k \in J$, be an optimal path flow vector. Then, for class-k job flow, if $\bar{x}_p^k > 0$ for some path p of an S-D pair P_i^k, we must be able to transfer a small amount $\delta > 0$ from path p to any other path p' of the same S-D pair without decreasing the mean response time; Other wise, the optimality of $\bar{x}$ would be violated. For the first derivative, the change in mean response time from this transfer is:

$$\delta \frac{\partial D(\bar{x})}{\partial x_{p'}^k} - \delta \frac{\partial D(\bar{x})}{\partial x_p^k},$$

and since this change must be non-negative, we obtain:

$$\bar{x}_p^k > 0 \;\Rightarrow\; \frac{\partial D(\bar{x})}{\partial x_{p'}^k} \geq \frac{\partial D(\bar{x})}{\partial x_p^k},\; for\ all\ p' \in P_i^k. \tag{3.10}$$

In other words, optimal path flow is positive only on paths with *a minimum first derivative length*. Furthermore, at an optimum, the paths along which the input flow γ_i^k of a S-D pair $(i, d),\ i \in N$, that are different must have equal lengths (and less than or equal to the length of all other paths of S-D pair (i, d)).

Thus, the condition (3.10) is a *necessary* condition for optimality of $\bar{x}$. The above proof is also valid for the *only if* part of the second part of the theorem for convex functions. Now, the above proof can be back tracked to prove the second part of the theorem when we assume that the functions D_{ij}^k are convex. For example, when the second derivatives $(D_{ij}^k)''$ exist and are positive in the domain of definition of D_{ij}^k. **Q.E.D**

Having shown the optimality of the solution, we will now show that the flow on path p is indeed identical to the flow on a link (i, j), when $(i, j) \in p$. This intermediate result will be useful for us later.

Lemma 3.1. *The class-k job flow on link $(i, j),\ j \in V_i$, is equal to the class-k job flow on path $p,\ p \in P_i^k$ and $(i, j) \in p$.*

Proof. Recall the definition of path at the beginning of this section and refer to Fig. 3.4. We can observe that there is only one path passing through link $(i, j),\ j \in V_i$, which is $(i \to j \to d)$. From (3.6):

$$\begin{aligned} x_{ij}^k &= \sum_{\substack{all\ paths\ p \\ containing\ (i,j)}} x_p^k \\ \Rightarrow\ x_{ij}^k &= x_p^k,\ (i, j) \in\ p\ and\ p \in P_i^k. \end{aligned}$$

Hence the proof. **Q.E.D**

Lemma 3.2. *In an optimal solution to problem (3.7), one or both of the class-k job flows on link (i, j) and (j, i) are zero. That is, $x_{ij}^k x_{ji}^k = 0$, where $i \in N$ and $j \in V_i$.*

Proof. This can be proved by contradiction. Assume there exists class-k link traffic in an optimal solution, such as $x_{ij}^k > 0$ and $x_{ji}^k > 0$, where $i,\ j \in N$ *and* $j \in V_i$. From Lemma 3.1, we have:

$$x_{ij}^k = x_p^k > 0,\ (i,j) \in p \ and \ p \in P_i^k,$$

$$x_{ji}^k = x_{p'}^k > 0,\ (j,i) \in p' \ and \ p' \in P_j^k.$$

From Theorem 3.1, we can deduce that path p is a MFDL path for a S-D pair (i,d) and path p' is the MFDL path for S-D pair (j,d), where p is the path given by $(i \to j \to d)$ and p' is the path given by $(j \to i \to d)$.

Consider node i first. Let p_i be the path (i,d) and $p_i \in P_i^k$. Since path p is a MFDL path in P_i^k and $\bar{x}$ is an optimal solution, we obtain:

$$\begin{aligned} \frac{\partial D(\bar{x})}{\partial x_{p_i}} &\geq \frac{\partial D(\bar{x})}{\partial x_p} \\ \Rightarrow \frac{\partial D(\bar{x})}{\partial x_{id}^k} &\geq \frac{\partial D(\bar{x})}{\partial x_{ij}^k} + \frac{\partial D(\bar{x})}{\partial x_{jd}^k}. \end{aligned} \quad (3.11)$$

Similarly, for node j we obtain:

$$\begin{aligned} \frac{\partial D(\bar{x})}{\partial x_{p_j}} &\geq \frac{\partial D(\bar{x})}{\partial x_{p'}} \\ \Rightarrow \frac{\partial D(\bar{x})}{\partial x_{jd}^k} &\geq \frac{\partial D(\bar{x})}{\partial x_{ji}^k} + \frac{\partial D(\bar{x})}{\partial x_{id}^k}. \end{aligned} \quad (3.12)$$

where p_j is the path (j,d) and path $(j,d) \in P_j^k$. Since function D is an increasing function, then $\partial D(\bar{x})/\partial x_{ij}^k > 0$ and $\partial D(\bar{x})/\partial x_{ji}^k > 0$. From (3.11) and (3.12), we have:

$$\frac{\partial D(\bar{x})}{\partial x_{id}^k} > \frac{\partial D(\bar{x})}{\partial x_{jd}^k} \text{ and } \frac{\partial D(\bar{x})}{\partial x_{jd}^k} > \frac{\partial D(\bar{x})}{\partial x_{id}^k},$$

which is a clear contradiction and hence, the lemma. **Q.E.D**

Lemma 3.3. *An optimal solution to problem (3.7) is cycle free, i.e., there exists no class-k link traffic such that* $x_{i_1 i_2}^k > 0,\ x_{i_2 i_3}^k > 0, ...,\ x_{i_m i_1}^k > 0$, *where* $i_1,\ i_2, ...,\ i_m \in N$.

Proof. We again prove this by contradiction. Assume that there is a class-k link traffic in

an optimal solution, such as $x^k_{i_1 i_2} > 0,\ x^k_{i_2 i_3} > 0, ..., x^k_{i_m i_1} > 0$, where $i_1,\ i_2, ...,\ i_m$ are indices of distinct nodes. Considering $x^k_{i_1 i_2}$ and $x^k_{i_2 i_3}$, since function D is an increasing function and according to Theorem 3.1 and Lemma 3.1, for links (i_1, d) and (i_2, d), we obtain:

$$\frac{\partial D(\bar{x})}{\partial x^k_{i_1 d}} > \frac{\partial D(\bar{x})}{\partial x^k_{i_2 d}}.$$

Recursively applying the proof technique used in Lemma 3.2 using (3.11), we obtain:

$$\begin{aligned} \frac{\partial D(\bar{x})}{\partial x^k_{i_1 d}} &> \frac{\partial D(\bar{x})}{\partial x^k_{i_2 d}}, \\ &\cdots, \\ \frac{\partial D(\bar{x})}{\partial x^k_{i_m d}} &> \frac{\partial D(\bar{x})}{\partial x^k_{i_1 d}}. \end{aligned}$$

which is a clear contradiction and hence the lemma. **Q.E.D**

Lemma 3.2 and Lemma 3.3 signify the fact that, in an optimal solution, the class-k job flow rate from the node i to node j can not be transferred back to node i directly from node j or through other nodes. This is indeed an important result: it is guaranteed that no job that entered the system shall leave without getting processed.

3.3.2 Design of the proposed algorithm

In Chapter 3.3.1, it was shown that an optimal solution results only if job flow travels along MFDL paths for each S-D pair. Equivalently, a set of path flows is strictly suboptimal only if there is a positive amount of flow that travels on a non-MFDL path. This suggests that a suboptimal solution can be improved by transferring some flow to an MFDL path from other paths for each S-D path. However, by transferring all the flow of each S-D pair to the MFDL path will lead to an oscillatory behavior. Thus, it is more appropriate to transfer only part of the flow from other paths to the MFDL path. Below, we shall determine the amount of these flows from each of the other paths, between an S-D pair, to be transferred to MFDL to seek an optimal solution. To do this, we propose our algorithm based on *Newton's method* [79],

which is:

$$x^{(r+1)} = x^{(r)} - \alpha^{(r)}[\frac{\partial^2 D(x^{(r)})}{(\partial x_i)^2}]^{-1}\frac{\partial D(x^{(r)})}{\partial x_i},\ r = 0,\ 1, \dots, \text{and } i = 1, ..., n, \tag{3.13}$$

where $\alpha^{(r)}$ is a positive scalar step-size at iteration r, determined according to some rule [62].

Consider the optimization problem (3.7). Assume that the second derivatives of D_{ij}^k, denoted by $D_{ij}^{k}{}''(x_{ij}^k)$, are positive for all x_{ij}^k. Let $(x^k)^{(r)} = \{(x_p^k)^{(r)}\}$ be the class-k job path flow vector obtained after r iterations, and let $\{(x_{ij}^k)^{(r)}\}$ be the corresponding set of total amount of class-k job link flows. For each S-D pair (i, d), let $\bar{p}_i^k$ be an MFDL path for class-k job, $k \in J$.

The optimization problem (3.7) can be converted (for the purpose of the next iteration) to a problem involving only active (positive) constraints by expressing the flows of MFDL paths $\bar{p}_i^k$ in terms of the other path flows, while eliminating the passive (equality) constraints (3.8).

$$\sum_{p \in P_i^k} x_p^k = \gamma_i^k,\ i \in N,\ k \in J,$$

in the process. For each S-D pair (i, d), $x_{\bar{p}_i^k}^k$ is substituted in the cost function $D(x^k)$ using the equation:

$$x_{\bar{p}_i^k}^k = \gamma_i^k - \sum_{\substack{p \in P_i^k \\ p \neq \bar{p}_i^k}} x_p^k, \tag{3.14}$$

thereby obtaining a redefined problem of the form:

$$\begin{aligned} Minimum: \quad & \tilde{D}(\tilde{x}), \\ subject\ \ to: \quad x_p^k \geq 0, \quad & for\ all\ p \in P_i^k,\ p \neq \bar{p}_i^k,\ i \in N, \end{aligned} \tag{3.15}$$

where $\tilde{x}$ is the vector of all class-k job path flows which are not MFDL paths.

We calculate the derivatives that will be needed to apply the iteration (3.13) to the problem defined by (3.15). Using (3.14) and the definition of $\tilde{D}(\tilde{x})$, we get:

$$\frac{\partial \tilde{D}(\tilde{x}^{k(r)})}{\partial x_p^k} = \frac{\partial D(x^{k(r)})}{\partial x_p^k} - \frac{\partial D(x^{k(r)})}{\partial x_{\bar{p}_i}^k}, \tag{3.16}$$

for all $p \in P_i^k, p \neq \bar{p}_i^k, i \in N$, where $x^{k(r)}$ is the vector x^k in the $r-$th iteration and $\partial D(x)/\partial x_p$ is the first derivative length of path p, that is given by (3.9). Regarding second derivatives, a straightforward differentiation of the first derivative expressions (3.16) and (3.9) shows that:

$$\begin{aligned} \frac{\partial^2 \tilde{D}(\tilde{x}^{k(r)})}{(\partial x_p^k)^2} &= \frac{1}{\Phi}\sum_{l\in J}\sum_{(i,j)\in\ L_p} D_{ij}^l{}''(x_{ij}^{k(r)}) \\ &= \frac{1}{\Phi}\{\sum_{(i,j)\in L_p}\frac{\partial^2 D_{ij}^1(x_{ij}^1)}{(\partial x_{ij}^k)^2}+\cdots+\sum_{(i,j)\in L_p}\frac{\partial^2 D_{ij}^m(x_{ij}^m)}{(\partial x_{ij}^k)^2}\}, \end{aligned}$$

$m = |J|$, and for all $p \in P_i^k, p \neq \bar{p}_i^k, i \in N$, where, for each p, L_p is a set of links belonging to either p or to the corresponding MFDL path $\bar{p}_i^k$, but not in both. This can be realized from (3.16).

Expressions for both the first and second derivatives of the "reduced" cost $\tilde{D}(\tilde{x})$, are available and thus the scaled projection method given by (3.13) can be applied. The iteration takes the form:

$$x_p^{k(r+1)} = max\{0, (x_p^{k(r)} - \alpha^r H_p^{-1}(d_p - d_{\bar{p}_i^k}))\},\ for\ p \in P_i^k, p \neq \bar{p}_i^k, i \in N, \tag{3.17}$$

where d_p and $d_{\bar{p}_i^k}$ are the first derivative lengths of the paths p and $\bar{p}_i^k$ and can be rewritten using (3.9) as:

$$d_p = \frac{1}{\Phi}\sum_{l\in J}\sum_{\substack{all\ \ links\\(i,j)\ \ on\ \ path\ \ p}} D_{ij}^l{}'(x_{ij}^{k(r)}),\ d_{\bar{p}_i^k} = \frac{1}{\Phi}\sum_{l\in J}\sum_{\substack{all\ \ links\\(i,j)\ \ on\ \ path\ \ \bar{p}_i^k}} D_{ij}^l{}'(x_{ij}^{k(r)}), \tag{3.18}$$

and H_p is the "second derivative length" given by:

$$H_p = \frac{1}{\Phi}\sum_{l\in J}\sum_{(i,j)\in L_p} D_{ij}^l{}''(x_{ij}^{k(r)}), \tag{3.19}$$

used in (3.17). The step-size α^r is some positive scalar which may be chosen by a variety of methods. From our rigorous simulation test runs, we choose α^r to be a constant from the range $[0.5, 1]$ for all r, which is shown to work well in our method.

The proposed algorithm LBVR transforms the basic load balancing problem into a routing problem and it is based on *Newton's method* (3.13). Notice that the class-k jobs at node i,

$\gamma_i^{k(r)}$ may vary in every iteration. We choose *Newton's method*, because using this method, an optimal solution can be implemented naturally when the input rates are time varying, as discussed in [27]. Further, Newton's method becomes an obvious choice for our problem context owing to the style adopted in our problem formulation. As it is important to investigate on the rate of convergence for such an optimization problem, we will discuss the complexity of convergence of our algorithm in the following section. Due to the varying input conditions ($\gamma_i^{k(r)}$, in every iteration), it may be possible that in some iteration, $x_{\bar{p}_i^k}^{k(r)} = \gamma_i^{k(r)} - \sum x_p^{k(r)} < 0$, where $p \in P_i^k$ and $p \neq \bar{p}_i^{k(r)}$. If this happens, then this will violate the constraint $\bar{x}_{p_i^k}^k \geq 0$ and may be detrimental in the next iteration. To circumvent this problem, we iterate *Step 2* without changing the values of k and i, till the constraint $\bar{x}_{p_i^k}^k \geq 0$ is satisfied and then carry out the next iteration.

Our algorithm has some interesting and important features. The first one is due to its simple and feasible implementable design structure. Though the paths of each S-D pair (i, d) are recomputed before the main calculation phase, we can add or delete some paths according to the current status of the system during the course of main iteration. The algorithm can be implemented in a decentralized fashion in the sense that each node can use the algorithm independently and depends only on local information collected from neighboring nodes. The second important feature is that it can be applied in a *dynamic* load balancing environment with little modification. Finally, the computational time of our algorithm is significantly small. Note the complexity of the algorithm for each node is $O(v+1) \approx O(v)$, $(v = |V_i|)$. To implement this algorithm, we only need to compute the first and second derivative functions of $D(x)$, which can be easily obtained. The distributed working style is discussed as following.

Any one of the processors in the system works as a *core* processor whose role is to trigger computations among the set of processors in the system. At initial time, the core processor starts to compute an optimal solution and also informs the rest of the nodes to compute one-by-one in a pre-defined sequence. The objective of the core processor is to avoid the time

asynchrony among the processors in the system. When node i finishes computing and obtains a solution, it broadcasts the solution to its neighboring nodes. After some iterations, when the stopping rule is satisfied, all the nodes stop computing and obtain an optimal solution, if it exists. It may be noted that if the core processor fails in some situation, any other processor in the system can assume the role of a core processor. A pseudo-code of LBVR algorithm is also shown in Table 3.1.

3.3.3 Rate of convergence

We denote $h(x^r) = D'(x^r)$. If an optimal solution is at point $\bar{x}$, then we have $h(\bar{x}) = 0$. Alternatively, from (3.17), we have:

$$\phi(x^r) = x^r - \alpha^r \frac{h(x^r)}{h'(x^r)}, \tag{3.20}$$

where x^r is the solution vector x in r-th iteration. By the mean value theorem, we obtain:

$$x^{r+1} - \bar{x} = \phi(x^r) - \phi(\bar{x}) = \phi'(\xi^r)(x^r - \bar{x}), \tag{3.21}$$

where ξ^r lies between x^r and $\bar{x}$. Then, from (3.20) and (3.21), we obtain:

$$|x^{r+1} - \bar{x}| = \frac{|h''(\xi^r)h(\xi^r)|}{[h'(\xi^r)]^2}|x^r - \bar{x}|. \tag{3.22}$$

Because $h(\bar{x}) = 0$, from (3.22), we can obtain:

$$|h(\xi^r)| = |h(\xi^r) - h(\bar{x})| = |h'(\eta^r)||\xi^r - \bar{x}| \leq |h'(\eta^r)||x^r - \bar{x}|,$$

where η^r lies between ξ^r and $\bar{x}$. Hence, (3.22) can be further transformed to be:

$$|x^{r+1} - \bar{x}| \leq \frac{|h''(\xi^r)h'(\eta^r)|}{[h'(\xi^r)]^2}|x^r - \bar{x}|^2. \tag{3.23}$$

Let

$$\beta = sup\frac{|h''(\xi^r)h'(\eta^r)|}{[h'(\xi^r)]^2},$$

Table 3.1: Proposed Load Balancing Algorithm (LBVR)

Step 1: *Initialization*

$r = 0$. (r: iteration index)

For $k = 1$ to $m,\ m = |J|$.

For $i = 1$ to $n,\ n = |N|$.

Determine P_i^k, the set of class-k job flow paths of node i.

Calculate the rate of class-k jobs at node i, that is, $\gamma_i^{k(0)} = \phi_i^k + \sum_{j \in V_i} x_{ji}^{k(0)}$, where $x_{ji}^{k(0)} = 0,\ j \in V_i$.

Find a feasible $(x_p^{k(0)})$ as an initial feasible solution, which satisfies $\sum_{p \in P_i^k} x_p^{k(0)} = \gamma_i^{k(0)}$. Here, we choose $x_{p_i^k}^{k(0)} = \gamma_i^{k(0)}$ and $x_p^{k(0)} = 0$, where $p \in P_i^k$ *and* $p \neq p_i^k$, p_i^k is the path $(i \to d)$ for class-k job.

End For.

End For.

Step 2: *Solution Procedure*

Let $r = r + 1$.

For $k = 1$ to $m,\ m = |J|$ do the following for class-k jobs.

For $i = 1$ to $n,\ n = |N|$ do the following for node i.

Let $x^{k(r)} = x^{k(r-1)}$.

Calculate the class-k jobs at node i, i.e., determine $\gamma_i^{k(r)}$.

For $j = 1$ to $v + 1,\ v = |V_i|$.

Calculate the first derivative lengths of the paths p, that is $d_p^{(r)},\ p \in P_i^k$.

End For.

Find the minimum first derivative length (MFDL) paths $\bar{p}_i^{k(r)}$ among P_i^k.

For $j = 1$ to $v,\ v = |V_i|$.

To path $p_j,\ p_j \in P_i^k,\ p_j \neq \bar{p}_i^{k(r)}$, calculate the second derivative length:

$$H_{p_j} = \frac{1}{\Phi} \sum_{l \in J} \sum_{(i,j) \in L_p} D_{ij}^{l}{}''(x_{ij}^{k(r)}).$$

Let $x_{p_j}^{k(r)} = max\{0,\ x_{p_j}^{k(r-1)} - \alpha^{(r-1)} H_{p_j}^{-1} (d_{p_j}^{(r-1)} - d_{\bar{p}_i^{k(r-1)}}^{(r-1)})\}$.

Let $x_{ij}^{k(r)} = x_{p_j}^{k(r)},\ link\ (i,j) \in p_j$.

End For.

Let $x_{\bar{p}_i^{k(r)}}^{k(r)} = \gamma_i^{k(r)} - \sum_{p \in P_i^k, p \neq \bar{p}_i^{k(r)}} x_p^{k(r)}$ and $x_{ij}^{k(r)} = x_{\bar{p}_i^{k(r)}}^{k(r)},\ link\ (i,j) \in \bar{p}_i^{k(r)}$.

End For.

End For.

Step 3: *Stopping rule*

If $|D(x^{(r)}) - D(x^{(r-1)})|/D(x^{(r)}) < \varepsilon$, then **stop**, where ε is a desired acceptable tolerance for solution quality; otherwise, go to Step 2.

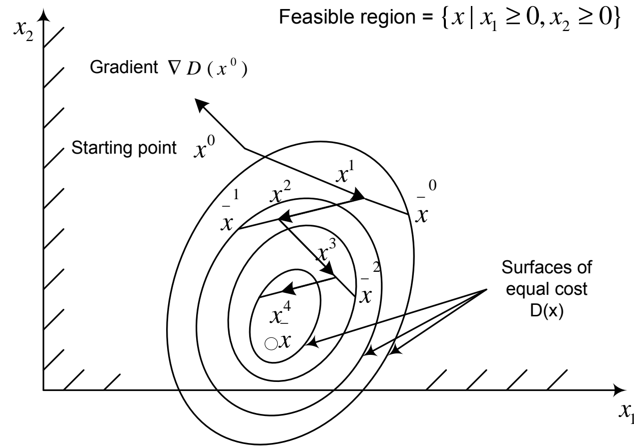

Figure 3.5: Operation of the proposed algorithm.

then from (3.23), we obtain:

$$|x^{r+1} - \bar{x}| \leq \beta |x^r - \bar{x}|^2.$$

Hence the convergence of our algorithm is 2, which exhibits a **super-linear** convergence [62]. The workings of the proposed algorithm are shown in Fig. 3.5.

3.4 Experimental Results and Discussions

Our algorithm LBVR retains two important features of practical interest: one is its decentralized style of working and the other is its simple structure, in terms of implementation ease. To appreciate one of the strengths of our algorithm, consider Fig. 3.2. In this figure, if the heavily loaded node 1 wants to send some jobs to lightly loaded node 3, when the failure of link $(1, 3)$ is occurred. In this case, our proposed algorithm LBVR considers adding a multi-hop routing path $(1 \rightarrow 2 \rightarrow 3)$ for node 1 to take advantage of additional information on link $(2, 3)$. Hence, LBVR algorithm is more robust in its working style (implementation). As a matter of fact, with the approach proposed in [12] node 1 still cannot send jobs to node 3 *directly* via node 2 due to the link failure. At the same time, based on *Golden Section*

Search, LK algorithm obtains a local optimal solution in each iteration and it is proven that by carrying such iterations, the global optimal solution can be eventually obtained. However, in [27, 62], it is pointed out that to obtain a faster convergence rate, it is better to reach a solution between the original starting point and the local optimal point in each iteration instead of reaching the local optimal point directly.

The work reported in [12] proposes an algorithm (LK) and compares with well-known FD (Flow Deviation) algorithm [48]. The FD algorithm is a standard steepest-descent algorithm, which evaluates from a feasible solution in the steepest descent direction and chooses a step-size that minimizes the mean job flow time along such a direction to form a new feasible solution in each iteration. It was proven that LK algorithm performs much better than FD algorithm in terms of its computational time (faster convergence). Thus, it is wiser to compare the performance with LK algorithm in terms of algorithmic convergence rate and basic design. Secondly, we use our proposed algorithm LBVR to study an optimal multi-class job load balancing problem in a distributed Mesh-connected network system, with a detailed illustrative example showing the complete workings. We present a detailed discussion on the performance of our algorithm. We also conduct experiments in Ring network and Ethernet network to compare the performances of LBVR and LK algorithms in different network topologies. For the purpose of continuity, we give a brief introduction of LK algorithm bellow.

3.4.1 LK algorithm in brief

To obtain an optimal solution to problem (3.3) with its constraints, LK algorithm basically forms the Lagrangian functions as follows:

$$H(\beta, x, \alpha) = \Phi D(\beta, x) + \sum_{k \in J} \sum_{i \in N} \alpha_i^k (\phi_i^k + \sum_{j \in V_i} x_{ji}^k - \sum_{j \in V_i} x_{ij}^k - \beta_i^k),$$

where α_i^k $(i \in N, k \in J)$ are the Lagrangian multipliers. The solution to the problem is to determine the values of α_i^k. The set of neighboring nodes of node i (V_i) are divided into the following four sets for each class-k in the optimal solution:

1) Class-k active source nodes (A_i^k): Node j in this set sends a part of class-k arriving jobs to node i and also processes the other part of class-k jobs locally.

2) Class-k idle source nodes (Id_i^k): Node j in this set sends all class-k arriving jobs to node i.

3) Class-k neutral nodes (Nu_i^k): Node j in this set does not send (or receive) any class-k jobs to (or from) node i.

4) Class-k sink nodes (S_i^k): Node j in this set receives some class-k jobs from node i.

In LK Algorithm, in each iteration, the algorithm uses the *Golden section search* (e.g., [77]) to find the values of α_i^k. Once α_i^k are determined, the values of $\beta_j^k, x_{ji}^k, x_{ij}^k$ and $\beta_i^j, j \in V_i$ can be directly obtained by the equations deduced in [12]. The stopping rule is similar with our algorithm.

In addition, the authors compare their algorithm with the well-know FD (Flow Deviation) algorithm. The FD algorithm is a standard steepest-descent algorithm, which evaluates from a feasible solution in the steepest descent direction and chooses a step-size that minimizes the mean job flow time along such a direction to form a new feasible solution in each iteration. Then, they show that their algorithm has much faster convergence to the minimum response time than FD algorithm. We omit further details.

3.4.2 Demonstration of LBVR algorithm: An example of load balancing

We shall now show an illustrative example which demonstrates the details of the workings of our LBVR algorithm.

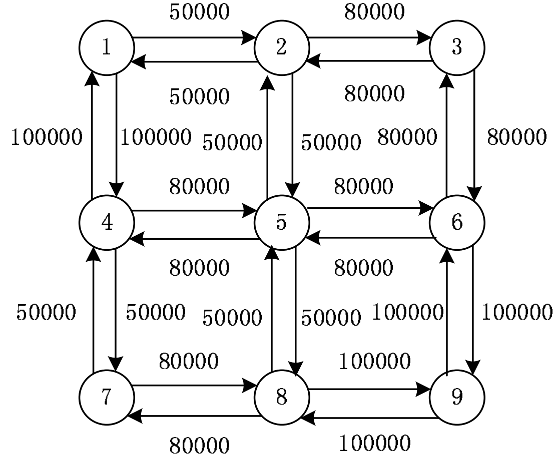

Figure 3.6: An example of a 9-node distributed computer system.

Consider a distributed computer system with 9 Mesh-connected computers as shown in Fig. 3.6. In such a system, we assume that there are two classes of jobs ($k = 1, 2$) which arrive at node i according to a time-invariant Poisson process. The inter-arrival times of jobs of each class are mutually independent. A job of class-1 is always served prior to a job of class-2. Within each class the queue discipline is first-come-first- served. A class-1 job arriving when no other jobs of its class are present will wait until the job (if any) of class-2 being served completes its service. The arrival processes of all classes are assumed independent, Poisson, and independent of service times. We assume node i is an M/G/1 system with one CPU and one queue. The first two moments of service time of each class-k at node i are denoted $\bar{X}_i^k = 1/\mu_i^k$ and $\bar{X}_i^{k2} = 2/(\mu_i^k)^2$ respectively, where μ_i^k is the average service time of class-k job of node i.

For node i, we will derive an equation for average delay for each priority class, which is similar to the $P - K$ formula [28]. We denote N_i^k as the average number of jobs in queue for priority k, W_i^k as the average queuing time for priority k, $\rho_i^k = \beta_i^k/\mu_i^k$ as the system utilization for priority k, and R as mean residual service time, respectively. We assume that the overall system utilization is less than 1, that is,

$$\rho_i^1 + \rho_i^2 + ... + \rho_i^m < 1.$$

When this assumption is not satisfied, there will be some priority class-k such that the average delay of customers of priority k and lower will be infinite while the average delay of customers of priority higher than k will be finite.

As in the derivation of the $P - K$ formula, we have, for the highest priority class:

$$W_i^1 = R + \frac{1}{\mu_i^1} N_i^1.$$

Eliminating N_i^1 from this equation using *Little's Theorem* [27], $N_i^1 = \beta_i^1 W_i^1$, we obtain:

$$W_i^1 = R + \rho_i^1 W_i^1,$$

and finally,

$$W_i^1 = \frac{R}{1 - \rho_i^1}.$$

For class-2 job, we have a similar expression for the queuing delay W_i^2 except that we have to count the additional queuing delay due to jobs of class-1 that arrive while a job is waiting in queue. This is the meaning of the last term in the above formula.

$$W_i^2 = R + \frac{1}{\mu_i^1} N_i^1 + \frac{1}{\mu_i^2} N_i^2 + \frac{1}{\mu_i^1} \beta_i^1 W_i^2.$$

Using *Little's Theorem* ($N_i^k = \beta_i^k W_i^k$), we finally obtain:

$$W_i^2 = \frac{R}{(1 - \rho_i^1)(1 - \rho_i^1 - \rho_i^2)}.$$

The mean residual service time R can be derived from $P - K$ formula as:

$$R = \frac{1}{2} \sum_{k=1}^{m} \beta_i^k \bar{X}_i^{k2}.$$

The average delay per job of class-k in node i is then given by:

$$T_i^k = \frac{1}{\mu_i^k} + W_i^k.$$

Finally, we obtain

$$T_i^1 = \frac{1}{\mu_i^1} + \frac{\sum_{k=1}^{2} \beta_i^k \bar{X}_i^{k2}}{2(1 - \rho_i^1)}, \quad (3.24)$$

$$T_i^2 = \frac{1}{\mu_i^2} + \frac{\sum_{k=1}^{2} \beta_i^k \bar{X}_i^{k2}}{2(1 - \rho_i^1)(1 - \rho_i^1 - \rho_i^2)}, \quad (3.25)$$

$Node(i)$	μ_i^1 (jobs/sec)	$\bar{X}_i^{12} = \frac{2}{(\mu_i^1)^2}$	μ_i^2 (jobs/sec)	$\bar{X}_i^{22} = \frac{2}{(\mu_i^2)^2}$
1	6	1/18	6	1/18
2	5	2/25	5	2/25
3	7	2/49	5	2/25
4	6	1/18	4	1/8
5	5	2/25	6	1/18
6	5	2/25	4	1/8
7	6	1/18	5	2/25
8	5	2/25	4	1/8
9	5	2/25	6	1/18

Table 3.2: Parameter values of node model

as the mean node delays per job of class-1 and class-2, respectively, in node i.

Table 3.2 gives the values of the parameters of the node models we used in this numerical example. Using (3.24), (3.25) and the values of the parameters, we obtain the functions of total delay for the class-1 and 2 job rates. For example, for node 1

$$\begin{aligned} \beta_1^1 F_1^1(\beta_1^1) &= D_{1d}^1(x_{1d}^1) = x_{1d}^1\left(\frac{1}{6} + \frac{x_{1d}^1/18 + x_{1d}^2/18}{2(1 - x_{1d}^1/6)}\right), \\ \beta_1^2 F_1^2(\beta_1^2) &= D_{1d}^2(x_{1d}^2) = x_{1d}^2\left(\frac{1}{6} + \frac{x_{1d}^1/18 + x_{1d}^2/18}{2(1 - x_{1d}^1/6)(1 - x_{1d}^1/6 - x_{1d}^2/6)}\right), \end{aligned}$$

where, $1 - x_{1d}^0/6 - x_{1d}^1/6 > 0$. For the communication links, we use the M/M/1 model. For the sake of simplicity, we assume that there is no difference between class-1 and class-2 jobs as far as link model is concerned. Hence, the job flow rate for link $(i, j),\ j \neq d$ is $x_{ij} = x_{ij}^0 + x_{ij}^1$. Assume that the transmission capacity of the link (i, j) is c_{ij} (packets/sec). Assume that, on an average, Q_1 packets are required to describe a job and that, on average, Q_2 packets are required in sending back a response for a job. Here, we choose $Q_1 = Q_2 = Q = 100$. Hence:

$$x_{ij} G_{ij}(x_{ij}) = \frac{Q x_{ij}}{c_{ij} - Q x_{ij} - Q x_{ji}} + \frac{Q x_{ij}}{c_{ji} - Q x_{ji} - Q x_{ij}}. \tag{3.26}$$

$Node(i)$	ϕ_i^1 (packets per min)	ϕ_i^2 (packets per min)
1	40	60
2	50	40
3	20	30
4	50	80
5	70	60
6	60	70
7	10	50
8	60	65
9	30	40

Table 3.3: Average external job arrival rates for class-1 and class-2 jobs

The first part of (3.26) signifies the communication delay of job flow on link (i, j) and the second part signifies the communication delay of the responses sent back from node j to i. The capacities of each link in packets-per-second are shown in Fig. 3.6. The average external job arrival rate ϕ_i^k for class-1 and class-2 job are shown in Table 3.3.

By adding a virtual node d into the system and connecting node i and node d with link (i, d), we obtain Fig. 3.7. Using the node delay model (3.24), the communication delay model (3.26), and the loads of two classes of jobs for each node shown in Table 3.3, we obtain the experimental results for LK algorithm and our algorithm LBVR, as shown in Fig. 3.8. Table 3.4 shows the processing rates at node i for both LK algorithm and the proposed algorithm. In this experiment, we set ε, the desired acceptable tolerance to be 10^{-9}.

From Table 3.4, we observe that both the LK and LBVR algorithms deliver more-or-less identical performance in terms of balancing the load (processing rate). However, Fig. 3.8 elicits a significant advantage of LBVR over LK in terms of rate of convergence. From this figure, we can observe that to obtain the similar mean response time of each job (as exemplified

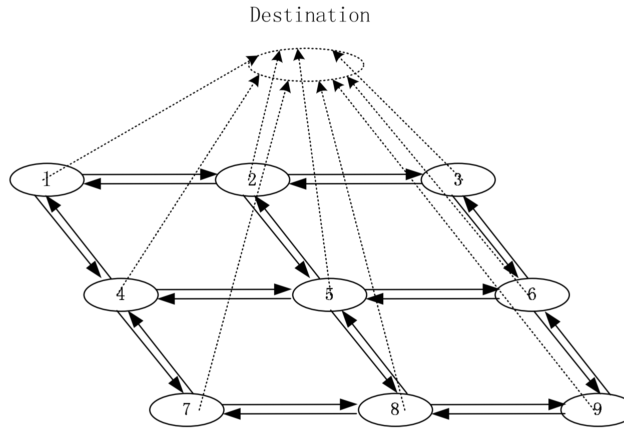

Figure 3.7: A distributed computer system with a virtual node d.

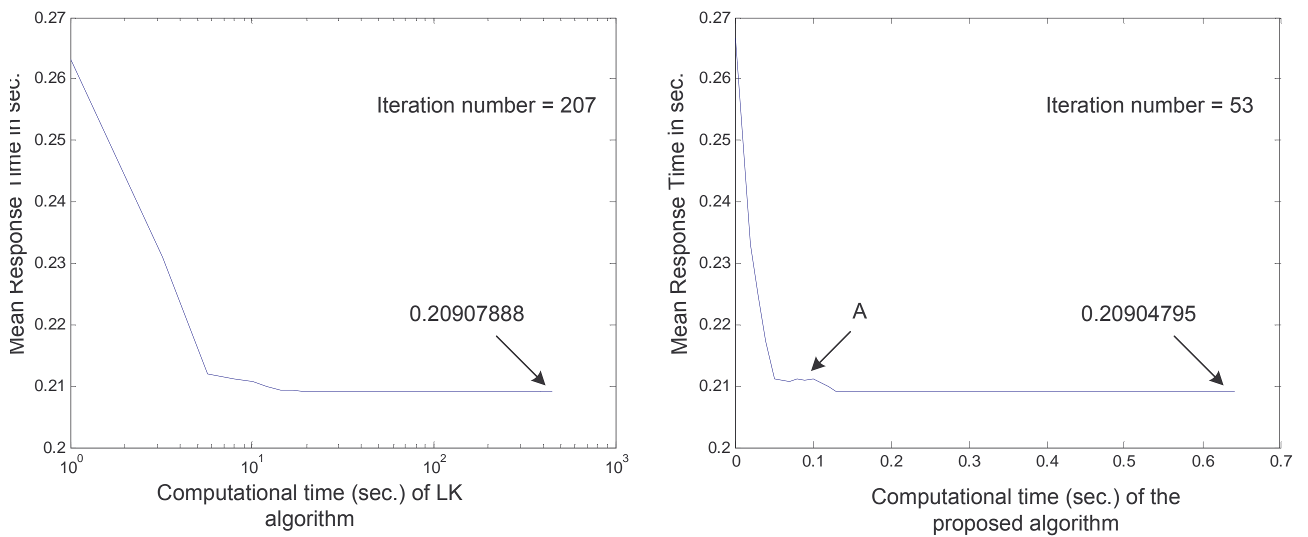

Figure 3.8: Comparison of the algorithms with respect to computational time.

	LK Algorithm		LBVR Algorithm	
$Node(i)$	Class-1(jobs/sec)	Class-2(jobs/sec)	Class-1(jobs/sec)	Class-2(jobs/sec)
1	0.000000	2.064288	0.000000	2.063941
2	0.000000	1.262917	0.000000	1.219627
3	2.463605	0.000000	2.480405	0.000000
4	1.695861	0.000000	1.683883	0.000000
5	0.000000	2.108337	0.000000	2.062730
6	0.815314	0.000000	0.837796	0.000000
7	0.684354	0.833333	0.672499	0.833333
8	0.840866	0.000000	0.825417	0.000000
9	0.000000	1.981124	0.000000	2.070368

Table 3.4: Completed job rate (β_i^k) in node i

in Table 3.4), the computational time of LK algorithm is more than 500 seconds and the system iteration number is 207, whereas LBVR consumes less than 1 second and the system iteration number is only 53. In every system iteration, each node in the system calculates according to its local information and broadcasts its local optimal solution to the neighboring nodes. Hence, less iteration number means faster convergence rate and less computational and communication overheads. This clearly shows that the proposed algorithm has much faster convergence rate. This significant improvement in performance mainly comes due to the computational nature of LBVR algorithm, the convergence rate of which is 2, super-linear. In LK algorithm, much of the time (close to 90%) is spent in *Golden-section* search phase and the remaining time is spent in determining certain inverse functions by means of recursive computations (as these do not have closed-form solutions). Further, a close scrutiny on the behavior of LBVR from Fig. 3.8, identified by an *arrow A*, reveals the following information. It may be observed that close to this interval, there is a little fluctuation in the mean response

time. The reason for this fluctuation is that, in some cases, the LBVR algorithm goes out of feasible solution space due to the choice of step size α. The algorithm, then has to take some time (consuming certain iterations) to re-enter the feasible solution space and to become normal. As the step-size choice affects the computational time, based on several rigorous test runs, we fixed the value of α to be within the range $[0.5, 1]$ for better accurate results.

3.4.3 Studies on more network topologies

In this section, we conduct more experiments on different network topologies. Firstly, we consider a homogeneous system with 9 nodes connected by a Ring network, as shown in Fig. 3.9. In this system, all 9 nodes are homogeneous and all the transmission capabilities of the links are the same. We assume there are two classes of jobs ($k = 1, 2$) and the nodes are M/G/1 systems. The first two moments of service time of each class-k job at node i are $\bar{X}_i^k = 1/\mu_i^k$ and $\bar{X}_i^{k2} = 2/(\mu_i^k)^2$, respectively, where $\mu_i^1 = 7$ and $\mu_i^2 = 4$ jobs per sec., for all the nodes in the system. c_{ij}, the transmission capability of link (i, j), is fixed to $80,000$ packets per sec in this system. Again, we use (3.24), (3.25) as the nodal delay model for class-1 and class-2 jobs, respectively, for node i. We also use (3.26) as the communication delay model for link (i, j). The average external job arrival rates ϕ_i^k for class 1 and class-2 jobs are randomly generated with uniform distributions on $[10, 200]$ and $[20, 100]$ jobs per min., respectively.

In this experiment, we loose the stopping rule and increase ε, the desired acceptable tolerance, to 10^{-4}. The experimental results are shown in Fig. 3.10, from which we can observe that with a higher ε, the iteration number of LK and LBVR algorithms are both reduced. The accuracies of both algorithms are nearly same, as that the optimal MRT of LK algorithm is 0.28695497 seconds and that of LBVR algorithm is 0.28695482 seconds. But for LK algorithm, it still needs more than 100 seconds to solve the optimal problem, whereas, LBVR algorithm only needs less than 0.05 seconds to obtain the optimal solution.

Now we conduct another experiment on a heterogeneous arbitrary network, as shown

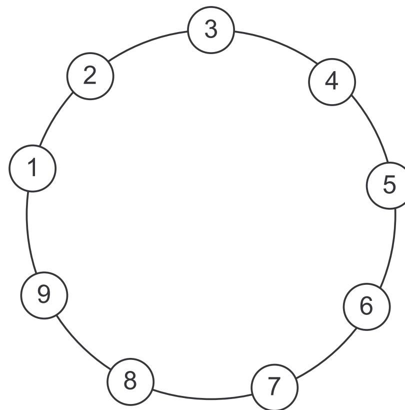

Figure 3.9: An example of a 9-node Ring computer system.

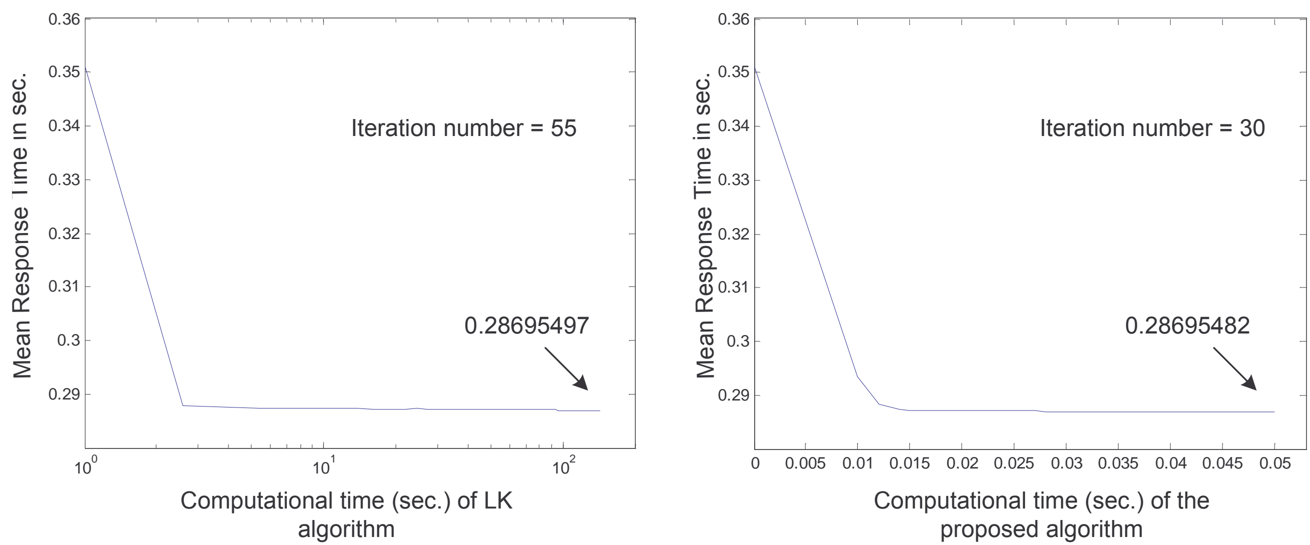

Figure 3.10: Comparison of algorithms on Ring network.

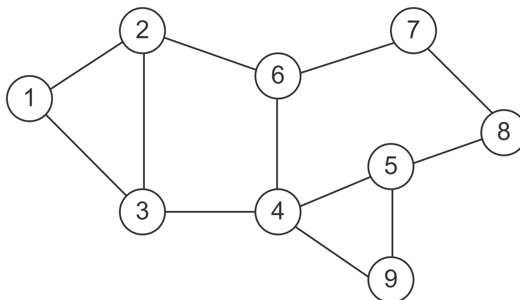

Figure 3.11: An example of a 9-node arbitrary network.

in Fig. 3.11. The topology of the network is randomly generated and the transmission capabilities of links are generated according to uniform distribution on $[50000, 80000]$ packets per sec. Here, we consider only one single class jobs. As we mentioned before, the computer model can be very complicated [4]. For convenience, we choose the central server models for the computers as in [12], which have the following mean nodal delay functions:

$$F_i(\beta_i) = \frac{a_i}{b_i - \beta_i},$$

where $\beta_i < b_i$ and a_i, b_i are constants for each computer i. Again, we use (3.26) as the communication delay functions for each link (i, j). Table 3.5 gives the parameters of the nodal models we use in this numerical experiment, where ϕ_i is the job arrival rate for each node i.

The numerical results are shown in Fig. 3.12. In this experiment, we only think single class job and use a simple nodal model. Hence, the computational time of both algorithms becomes much shorter. However, it still spends LK algorithm more than 30 seconds to solve the optimal problem while it only spends LBVR algorithm less than 0.02 seconds to obtain the similar optimal solution.

From those experiments, we can draw a conclusion that LBVR algorithm has much faster convergence to the mean response time than LK algorithm.

Note: the platform of all the experiments is P4 1.5G, 256M RAM.

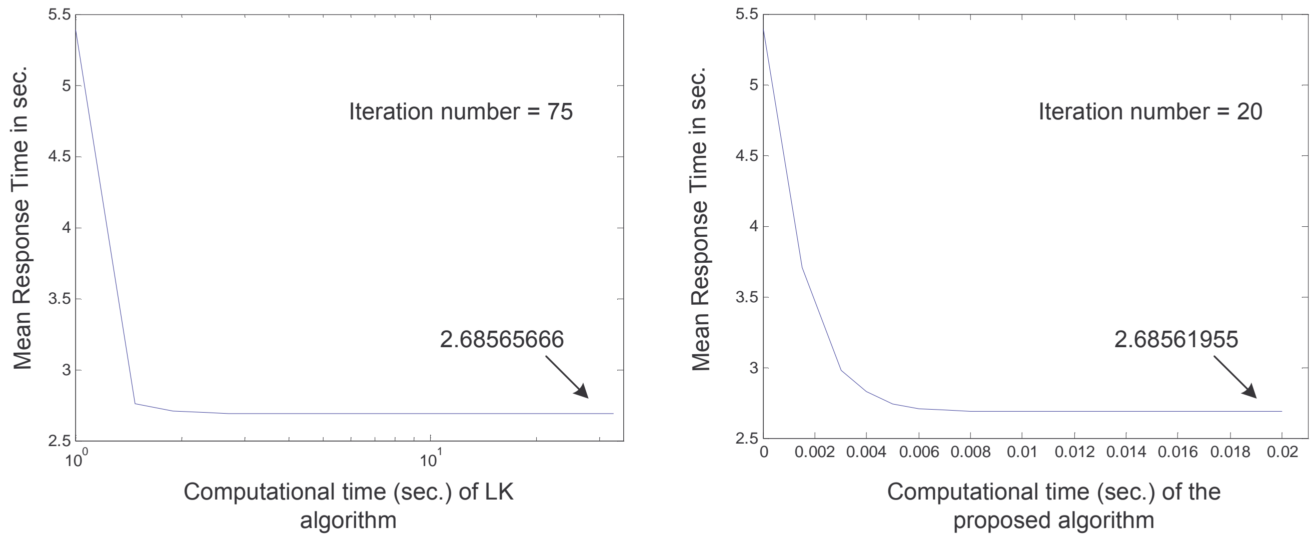

Figure 3.12: Comparison of algorithms on an arbitrary network.

Node i				Node i			
node i	a_i	b_i	ϕ_i jobs/min.	node i	a_i	b_i	ϕ_i jobs/min.
1	4	50	25	6	1	40	20
2	5	100	8	7	5	80	50
3	3	120	20	8	7	100	15
4	1	70	30	9	3	90	30
5	5	150	6				

Table 3.5: Parameters of each node in the system

3.5 Concluding Remarks

A novel *super-linear* static load balancing algorithm, referred to as *load balancing via virtual routing* (LBVR), using the concept of data *routing* in computer networks, has been proposed in this chapter. The objective is to minimize the mean response time of the jobs that arrive to a distributed system for processing. Our formulation considers a multi-class jobs that arrive at processors of a distributed network system for processing. The jobs are considered for processing in a non-preemptive manner. In this chapter, we have proposed a novel solution to the load balancing problem defined by (3.3). Several constraints such as, job flow rate, communication delays due to the underlying network, processing delays at the processors, are considered in the problem formulation. We formulated the problem of load balancing as a constrained non-linear minimization problem, and as a solution approach, we have transformed the problem into an equivalent problem of routing under the above set of constraints. The solution approach that was proposed is conclusively proven to yield an optimal solution, when the delay function defined in Chapter 3.1 is assumed to be a convex function. Whereas, when the delay function is a non-convex function, a *necessary* condition has been derived to obtain an optimal solution to the problem. Further, the design of the algorithm involves a systematic application of routing technique, through which a set of potential receivers, of an unbalanced quantum of load, is identified. This routing technique is shown to deliver a super-linear rate of convergence. The rate of convergence to optimal solution by our algorithm when compared with [12] was experimented and the results are shown to testify the theoretical findings. Finally, we have also shown that our algorithm guarantees that no job is left unprocessed once it enters the system for processing and the solution approach is cycle-free (Lemma 3.3).

Although the studies attempted in [12, 38] are the ones that are considered close to our modelling, there exist some key differences. In [38], while the formulation was generic, only one class of jobs was considered. In [12] multi-class of jobs were considered under a more-or-less

similar formulation. However, in the latter case, the underlying node model was different, in the sense that, each node i treated its neighboring nodes differently according to the transfer vector of class-k jobs. Whereas, in our model, we relax this constraint and handle a generic situation in which the entire set of class-k jobs arriving at node i (from other nodes and also externally at node i), are considered as a whole while balancing the entire load.

The current problem context poses the following challenges to be considered as possible future extensions. First attempt can be in tuning the formulation to handle the dynamic case of load balancing, wherein network routes and established paths may dynamically change. Secondly, additional constraints such as limited resources availability, such as buffer capacity at a node, bandwidth availability of channels, etc, can be considered in the problem formulation. Lastly, problem formulation and solution for handling jobs with time critical constraints (jobs having deadlines) can also be considered.

Chapter 4

Distributed Dynamic Load Balancing Strategies

Dynamic load balancing algorithms offer the possibility of improving load distribution at the expense of additional communication and computation overheads. In [23, 40], it was pointed out that the overheads of dynamic load balancing may be large, especially for a large heterogeneous distributed system. Hence, most of the research works in the literature focused on centralized dynamic load balancing [23, 41, 81, 82], in which an Management Station (M-Station)/Scheduler kept checking the system state and scheduled the arriving jobs among the processors by some strategies, such as Backfilling, Gang-Scheduling, and Migration [81], etc. Because of its scalability, flexibility and reliability, distributed dynamic load balancing offers more advantages than the centralized strategies, and thus has obtained more and more focuses recently [19, 43].

To the best of our knowledge, there is no work in the literature that systematically analyzes and compares the performance of the distributed dynamic load balancing algorithms. In this chapter, according to the job assignment methods, we classify the distributed dynamic load balancing algorithms into three policies: *Queue Adjustment Policy* (QAP), *Rate Adjustment Policy* (RAP) and *Combination of Queue and Rate Adjustment Policy* (QRAP). Based on

LBVR, we propose two efficient algorithms referred to as Rate based Load Balancing via Virtual Routing (RLBVR) and Queue based Load Balancing via Virtual Routing (QLBVR). It is the first time in the literature that a distributed system can adjust its scheduling according to optimal solutions dynamically using RLBVR algorithm. We introduce an algorithm called Estimated Load Information Scheduling Algorithm (ELISA) and an algorithm named Perfect Information Algorithm (PIA) reported in the literature [19], for the purpose of continuity.

The algorithms ELISA and PIA belong to QAP whereas RLBVR and QLBVR belong to RAP and QRAP, respectively. We carry out large number of rigorous simulation experiments to capture and analyze the effect of time-varying loads and different lengths of time intervals on the algorithms. As our focus is to analyze and understand the behaviors of the algorithms in terms of their load balancing abilities, minimization of mean response time, in our rigorous simulation experiments, we consider a single-class of jobs for processing. One of our added considerations in this study is to gain an intuition regarding the relative metrics of the different approaches under consideration. We extend our simulations to a large scale multiprocessor system such as Mesh architecture that is of practical use in real-life applications. Our contribution elicits certain important behaviors of the distributed dynamic load balancing algorithms that serve to quantify the performances under different situations. From rigorous experiments, recommendations are drawn to prescribe the suitability of the algorithms under various situations.

This chapter is organized as follows. Chapter 4.1 describes in detail the system model and the classification of the distributed dynamic load balancing algorithms. In Section 4.2, we propose RLBVR and QLBVR, and give a brief introduction of ELISA. In Section 4.3, we present the results of simulations carried out in a 2-processor system to illustrate certain salient features of the algorithms under consideration. In Section 4.4, we extend our work to large scale multiprocessor system, give a brief description of PIA and compare the algorithms of the three policies. We highlight our work in Section 4.5.

4.1 System Model and Classification of Dynamic Load Balancing Algorithms

We first present a general system model in the design of the algorithms. For convenience, we use "node" and "processor" interchangeably in the rest of this chapter. We consider a generic parallel/distributed system shown in Fig. 2.1. The system consists of n *heterogeneous* nodes, which represent host computers having different processing capabilities, interconnected by an underlying arbitrary communication network. Here, we only consider one class of jobs and hence we obtain a simple distributed computer system as shown in Fig. 1.1. We assume that jobs arrive at node i ($i \in N$) according to an ergodic process, such as *inhomogeneous Poisson process* with intensity function $\lambda_i(t)$ [83]. A job arriving at node i may either be processed locally or transferred through the network to another node j ($j \in N$) for remote processing. The service time of a job is a random variable that follows an exponentially distribution with mean $1/\mu_i$, where μ_i denotes the average job service rate of node i and represents the rate (in jobs served per unit time) at which node i operates when busy. The queue discipline of the jobs in each node is *FCFS* and the buffer size is infinite. We denote $\beta_i(t)$ as the rate at which the jobs are processed at node i at time t. Once a job starts to undergo processing in a node, it is allowed to complete processing without interruption and cannot be transferred to another node in the meanwhile.

In this model, we assume that there is a communication delay incurred when a job is transferred from one node to another before the job can be processed in the system and denote $x_{ij}(t)$ as the job flow rate from node i to node j ($j \in V_i$) at time t. Further, we assume that each link (i, j) can transfer the loads at its own transmission capability (otherwise referred to as transmission rate, commonly expressed as bytes/sec). We denote C as the set of transmission capacities of all the links and c_{ij} as the transmission capacity of a link (i, j), $c_{ij} \in C$. There are many communication delay models proposed for data networks in the literature and here

we choose M/M/1 [12, 27].

For load balancing algorithms, the model for a node comprises a scheduler, an infinite buffer to hold the jobs and a processor. The scheduler is to schedule the jobs arriving at the node such that the mean response time of the jobs is a minimum. In the absence of a scheduler in a node, the job flow takes the following sequence of actions: a job enters the buffer, waits in the queue for processing, leaves the queue and gets processed in the processor and then leaves the node (system). However, when a scheduler is present, depending on where a scheduler resides in a node to exercise its control on the job flow, we classify the distributed dynamic load balancing algorithms into three policies:

1. *Queue Adjustment Policy* (QAP): As shown in Fig. 4.1, the scheduler is placed immediately after the queue. Algorithms of this policy [19,22,45] attempt to balance the jobs in the queues of the nodes. When a job arrives at node i, if the queue is empty, the job will be sent to processor directly; Otherwise, the job will have to wait in the queue. The scheduler of node i periodically detects the queue lengths of other nodes that node i is concerned. When an imbalance exists, the scheduler will decide how many jobs in the queue should be transferred and where each of the jobs should be sent to. By queue adjustment, the algorithms could balance the load in the system.

2. *Rate Adjustment Policy* (RAP): As shown in Fig. 4.2, the scheduler is immediately placed

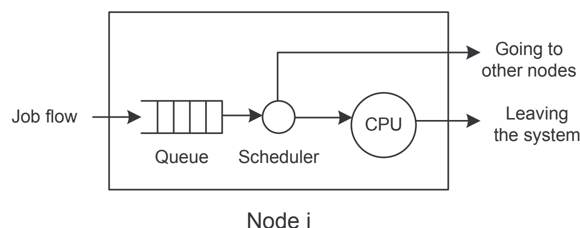

Figure 4.1: Node model for queue adjustment policy.

before the queue. When a job arrives at node i, the scheduler decides where the job should be

sent, whether it is to be sent to the queue of node i or to other nodes under consideration. Once the job has entered the queue, it will be processed by processor and will not be transferred to other nodes. Using this policy, the *static* algorithms [12, 47], can attempt to control the job processing rate on each node in the system and eventually obtain an optimal (or near optimal) solution for load balancing. Because of high the computation overheads, till now no dynamic algorithm in the literature uses this policy. In this chapter, we will propose a dynamic algorithm, which belongs to this policy.

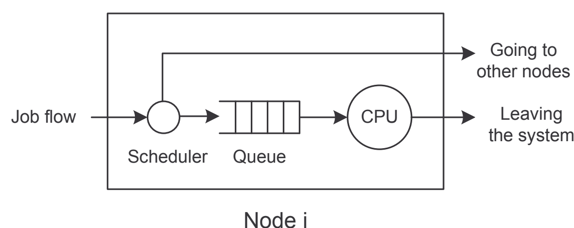

Figure 4.2: Node model for rate adjustment policy.

3. *Combination of Queue and Rate Adjustment Policy* (QRAP): As shown in Fig. 4.3, the scheduler is allowed to adjust the incoming job rate and also allowed to adjust the queue size of node i in some situations. Because we consider a dynamic situation, especially when we use RAP, in some cases, the queue size may exceed a predefined threshold and load imbalance may result. Once this happens, QAP starts to work and guarantees that the jobs in the queues are balanced in the entire system. In this policy, we can consider the rate adjustment as a "coarse" adjustment and the queue adjustment as "fine" adjustment. To the best of our surveys till date, there is no algorithm in the published literature that falls into this class of policies. In this chapter, we will propose an algorithm, which belongs to this policy that considers realizing a dynamic load balancing in distributed networks.

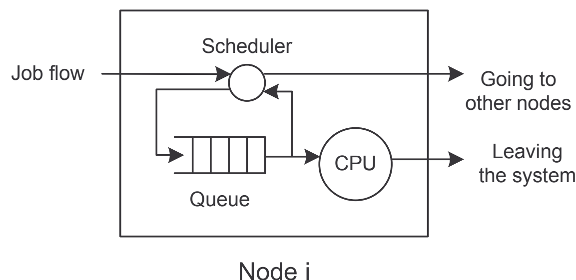

Figure 4.3: Node model for the combination of queue and rate adjustment policy.

4.2 Comparative Study on the Algorithms

In this section, we will introduce the algorithm named ELISA [19] which will be used as a benchmark algorithm, which qualifies under QAP category and we will propose two algorithms based on LBVR, referred to as *Rate based Load Balancing via Virtual Routing* (RLBVR) based on RAP and *Queue based Load Balancing via Virtual Routing* (QLBVR) based on QRAP, respectively. First, we give some notations which will be used throughout this chapter.

In distributed dynamic load balancing algorithms, the nodes in the system exchange their status information at periodic interval of time T_s, which is called the *status exchange interval.* The status information consists of the queue length at the instance of information exchange, an estimate of the job arrival rate and the processing rate. The instant at which this information exchange takes place is called a *status exchange epoch.* Each status exchange interval is further divided into equal subintervals denoted as *estimation intervals* T_e. The points of division are called *estimation epochs.* In order to keep the communication overheads minimal, the exchange of status information is restricted only to the neighboring nodes. In Fig. 4.4, T_n, T_{n-1} denote the status exchange epochs and t_1, t_2 represent the estimation epochs. Normally, the job arrival rate in a distributed computing system is not constant with respect to time and no node in the system is aware of the arrival rates of jobs. Each node estimates the job arrival rate by considering the number of arrivals in a certain fixed interval of time (called a *window*). The size of this window depends on how rapidly the arrival rate varies with time.

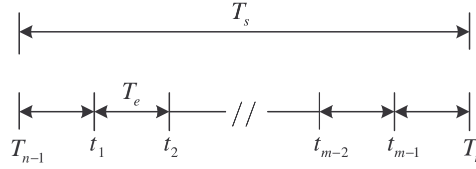

Figure 4.4: Intervals of estimation and status exchange.

Here, we consider the window to be an integer w multiple of T_s, i.e., window size $= wT_s$. Also, $T_s = m \times T_e$ time units from the last status exchange instant and $\hat{\lambda}_i^n$ is denoted as the estimated arrival rate of node i in the n-th T_s. This estimate is made at starting of the interval T_s at time $t = T_{n-1}$ in Fig. 4.4. Now, we introduce the algorithms of the three policies.

4.2.1 ELISA: Estimated Load Information Scheduling algorithm

We describe ELISA [19] in brief for the purpose of continuity. In ELISA, the load scheduling decision is taken as follows: from the estimated queue lengths of the nodes in its neighboring nodes and the accurate knowledge of its own queue length, each node computes the average load on itself and its neighboring nodes. Nodes in the neighboring set, whose estimated queue length is less than the estimated average queue length by more than a threshold θ, form an *active set*. The nodes under consideration transfers jobs to the nodes in the active set until its queue length is not greater than θ and more than the estimated average queue length. The value of θ, which is predefined, is a sensitive parameter and it is of importance to the performance of ELISA. Here, the threshold θ is fixed in such a way that the average response time of the system is a minimum. We present more details on ELISA in Table 4.1.

4.2.2 The proposed algorithm: RLBVR

Although LBVR is a static load balancing algorithm, due to its *super-linear* convergence rate, LBVR can be tuned to handle dynamic situations. Thus we attempt to design a dynamic load balancing algorithm RLBVR based on the working style of LBVR. The main structure

Table 4.1: Main structure of ELISA

Procedure. Transfer. (Computation of transfer probabilities and transfer of jobs)

1. Find average queue length of the neighboring set
2. If queue length of a node is greater than the average queue length (computed in 1), then:

   (a) construct the active set as follows: if a node in the neighboring set has a queue length less than the average queue length, include the node in the active set;

   (b) compute the probability of transferring from the node (source) to each node (destination) in the active set such that the source node load in excess of average queue length is distributed among nodes of active set.
3. Transfer the jobs as per the probabilities computed in 2(b).

Main Algorithm.

At the status exchange epoch, for each node:

1. estimate arrival rate by averaging the number of arrivals over the previous w status exchange intervals.
2. communicate status (queue length and estimate of arrival rate) to all nodes in the neighboring set.
3. call transfer.

At the estimation epoch, for each node:

1. estimate the queue length for each node in the neighboring set.
2. call transfer.

of this algorithm is:

First, we Add a *virtual* node, which is referred to as the *destination* node (node d), into the network system. Connect node d with each node i ($i \in N$) by a virtual direct link (i, d). Let the nodal delay of node i be treated as the communication delay on link (i, d). After these modifications, the process of load balancing can be described in an alternative way. As shown in Fig. 3.4, a three-node system has been transformed into a datagram network, in which node i basically acts as a *router*. The way in which the loads are shared by the nodes can be described as follows. The jobs that arrive at node i according to a Poisson process with an average external job arrival rate of λ_i are routed to destination node d via every node $j \neq i$. The goal of load balancing now can be alternatively stated as a problem which attempts to minimize the mean link delay for each job in the system. Then, each node i in the system only considers the routing paths of (i, d) and $(i, j) \rightarrow (j, d)$, $j \in V_i$. Use *Newton's method* [79] to obtain the optimal job routing rate of each path of node i. Eventually, obtain an optimal job transferring rate x_{ij} and job processing rate β_i, which are equal to job flow rates on paths $(i, j) \rightarrow (j, d)$ and path (i, d) respectively.

Each node i in the system monitors its external job arrival rate and obtains an estimate of the job arrival rate $\hat{\lambda}_i^n$, which will be used as the job arrival rate of node i during the time interval between T_{n-1} and T_n for computing. Before node i starts computing the optimal processing rate and transfer rates, node i broadcasts a request message to its neighboring nodes. When node j ($j \in V_i$) receives this message, it will send x_{ji}^n and β_j^n to node i immediately. If node i does not receive the response message from node j, it will consider node j has been shut down or the edge $< i, j >$ is broken down, and will not send jobs to node j until it receives some message from node j some time later. Once node i obtains all local information, it can start computing to obtain an optimal solution, following which, node i sends some jobs to its neighboring nodes and processes some jobs locally during this time interval. In the following, we will give more details on how to compute the optimal solution

in each node.

Procedure for a node

For node i in the system, we denote $P_i(\beta_i)$ as the mean nodal delay function for a job and $G_{ij}(x_{ij})$ as the mean communication delay function for a job transferred from node i to j. Some generic models of $G_{ij}(x_{ij})$ include the delay of sending a job from node i to node j and the delay of sending the response back from node j to node i. In general, the path taken in each of the above mentioned transfers (job and response transfers) may be different. Note that the profile of functions $P_i(\beta_i)$ and $G_{ij}(x_{ij})$ may be very complicated. In practice, for analytical ease, it is often assumed that the functions $P_i(\beta_i)$ and $G_{ij}(x_{ij})$ are differentiable, increasing and convex functions [5, 27]. Here, we introduce another function F_{ij} to unite the two different delay functions of P_i and G_{ij} as follows:

$$F_{ij}(x_{ij}) = \begin{cases} P_i(x_{ij}), (i,j) \in L \text{ } and \text{ } j = d; \\ G_{ij}(x_{ij}), (i,j) \in L \text{ } and \text{ } j \neq d. \end{cases}$$

Then node i obtains x_{ji}^n and β_j^n for the time interval between T_{n-1} and T_n. When node i starts to compute, it calculates the total job arrival rate γ_i^n, which is:

$$\gamma_i^n = \hat{\lambda}_i^n + \sum_{j \in V_i} x_{ji}^n.$$

Referring to Fig. 3.4 again, we can see that node i has P_i, a set of routing paths. Hence node i must determine the job flow rate x_p^n on each path p, $(p \in P_i)$ and the objective function of node i is

$$Minimize : D(x) = \frac{1}{\gamma_i^n} \sum_{p \in P_i} [\ x_p^n \sum_{\substack{all\ links\ (i,j) \\ on\ path\ p}} F_{ij}(x_{ij}^n)\ ], \tag{4.1}$$

$$Subject\ to: \qquad \sum_{p \in P_i} x_p^n = \gamma_i^n,$$

$$x_p^n \geq 0,\ p \in P_i.$$

We denote d_p as the first derivative length of path p with respect to x_p^n:

$$d_p = \sum_{\substack{all\ links\ (i,j) \\ on\ path\ p}} \frac{\partial[\ x_p^n F_{ij}(x_{ij}^n)\ ]}{\partial x_p^n}.$$

We now show that the flow on path p is indeed identical to the flow on a link (i, j), when$(i, j) \in p$. This intermediate result will be useful for us later.

Lemma 4.1: *The job flow on link $(i, j), j \in V_i$ is equal to the job flow on path p, $p \in P_i$ and $(i, j) \in p$.*

Proof. Recall the definition of path at the beginning of this section and refer to Fig. 3.4. We can observe that there is only one path passing through link (i, j), $j \in V_i$, which is $(i \to j \to d)$. We have:

$$\begin{aligned} x_{ij}^n &= \sum_{\substack{all\ paths\ p \\ containing\ (i,j)}} x_p^n \\ \Rightarrow\ x_{ij}^n &= x_p^n,\ (i, j) \in\ p\ and\ p \in P_i. \end{aligned}$$

Hence the proof. **Q.E.D**

We state the following theorem.

Theorem 4.1. *The set of values of $\bar{x}$ is an optimal solution to problem (4.1)* **only if** *job flow travels along minimum first derivative length (MFDL) paths for each S-D pair. In addition, if F_{ij} are assumed to be convex, then $\bar{x}$ is optimal* **if and only if** *job flow travels along MFDL for each S-D pair.*

Proof. We shall first prove the first part of the above theorem. Let $\bar{x}$ be an optimal path flow vector. Then, if $\bar{x}_p > 0$ for some path p of an S-D pair P_i, we must be able to transfer a small amount $\delta > 0$ from path p to any other path p' of the same S-D pair without decreasing the mean response time; Other wise, the optimality of $\bar{x}$ would be violated. For the first derivative, the change in mean response time from this transfer is:

$$\delta \frac{\partial D(\bar{x})}{\partial x_{p'}} - \delta \frac{\partial D(\bar{x})}{\partial x_p},$$

and since this change must be non-negative, we obtain:

$$\bar{x}_p > 0 \ \Rightarrow \ \frac{\partial D(\bar{x})}{\partial x_{p'}} \geq \frac{\partial D(\bar{x})}{\partial x_p}, \ for \ all \ p' \in P_i. \tag{4.2}$$

In other words, optimal path flow is positive only on paths with *a minimum first derivative length*. Furthermore, at an optimum, the paths along which the input flow γ_i of a S-D pair $(i, d), \ i \in N$ that are different must have equal lengths (and less than or equal to the length of all other paths of S-D pair (i, d)).

Thus, the condition (4.2) is a *necessary* condition for optimality of $\bar{x}$. The above proof is also valid for the *only if* part of the second part of the theorem for convex functions. Now, the above proof can be back tracked to prove the second part of the theorem when we assume that the functions F_{ij} are convex. For example, when the second derivatives $(F_{ij})''$ exist and are positive in the domain of definition of F_{ij}. **Q.E.D**

From Theorem 4.1, we can observe that an optimal solution results only if job flow travels along MFDL paths for each S-D pair. However, by transferring all the flow of each S-D pair to the MFDL path will lead to an oscillatory behavior. Thus, it is more appropriate to transfer only part of the flow from other paths to the MFDL path. We shall determine the amount of these flows from each of the other paths, between an S-D pair, to be transferred to MFDL in each iteration r, to seek an optimal solution.

Assume that the second derivatives of $D(x^n_{ij})$, denoted by $D''(x^n_{ij})$, are positive for all x^n_{ij}. Let $(x^n)^{(r)} = \{(x^n_p)^{(r)}\}$ be the job path flow vector obtained after r iterations in node i, and let $\{(x^n_{ij})^{(r)}\}$ be the corresponding set the job link flow rate. For each S-D pair (i, d), let $\bar{p}_i$ be an MFDL path and L_p be a set of links belonging to either p or to the corresponding MFDL path $\bar{p}_i$, but not in both. From Fig. 3.4, we can observe that there is no link belonging to any two different paths in P_i. The minimization problem (4.1) can be converted to a problem involving only active (strictly positive) constraints by expressing the flows of MFDL paths $\bar{p}_i$ in terms of the other path flows, while eliminating the passive (equality) constraints.

From [27] and [79], we can obtain that a basic iteration takes the form:

$$x_p^{n(r+1)} = max\{0, (x_p^{n(r)} - \alpha^r H_p^{-1}(d_p - d_{\bar{p}_i}))\}, \quad (4.3)$$

$$x_{\bar{p}_i}^{n(r+1)} = \gamma_i^{n+1} - \sum_{p \in P_i,\ p \neq \bar{p}_i^{(r)}} x_p^{n(r)}, \quad (4.4)$$

where $p \in P_i, p \neq \bar{p}_i, i \in N$, d_p and $d_{\bar{p}_i}$ are the first derivative lengths of the paths p and $\bar{p}_i$. H_p is the "second derivative length" given by:

$$\begin{aligned} H_p &= \sum_{(i,j) \in L_p} D''(x^{n(r)}) \\ &= \sum_{(i,j) \in p} \frac{\partial [x_p^{n(r)} F_{ij}(x_{ij}^{n(r)})]''}{\partial^2 x_p^{n(r)}} + \sum_{(i,j) \in \bar{p}} \frac{\partial [x_{\bar{p}}^{n(r)} F_{ij}(x_{ij}^{n(r)})]''}{\partial^2 x_{\bar{p}}^{n(r)}}. \end{aligned}$$

Once we obtain the optimal job flow rates $\bar{x}_p^n$, we obtain the optimal job processing rate $\bar{\beta_i^n}$ and the optimal job transfer rates $\bar{x}_{ij}^n$ immediately, which are:

$$\bar{\beta_i^n} = \bar{x}_p^n,\ link\ (i,\ d) \in p,$$

$$\bar{x_{ij}^n} = \bar{x_p^n},\ link\ (i,\ j) \in p.$$

A pseudo-code of the sub-algorithm for node is also shown in Table 4.2.

Once an optimal solution is obtained by node i, the probability of job transferring from node i to node j ($j \in V_i$) is determined by $p_{ij}^n = \frac{\bar{x}_{ij}^n}{\gamma_i^n}$ and the probability of job processing rate in node i is determined by $p_i^n = \frac{\bar{\beta}_i^n}{\gamma_i^n}$. Obviously, $\sum_{j \in V_i} p_{ij}^n + p_i^n = 1$ holds. In the next T_s, when a job arrives at node i, node i transfers the job to its neighboring node j according to the probability p_{ij}^n or processes the job according to the probability p_i^n.

At the next time instant T_n, node i obtains an estimate of the job arrival rate $\hat{\lambda}_i^{n+1}$ and the system starts to compute the optimal solution again. Thus the computation (and hence, the system) enters into the next time interval T_s and continues recursively.

4.2.3 The proposed algorithm: QLBVR

Now, we propose another dynamic load balancing algorithm referred to as "Queue based Load Balancing via Virtual Routing" (QLBVR), which belongs to QRAP category, namely

Table 4.2: Procedure for Node

Step 1: ***Initialization***

$r = 0$ (r: iteration index).

Determine γ_i^n.

Find a feasible solution x to satisfy $\sum_{p \in P_i} x_p^{n(r)} = \gamma_i^n$.

Step 2: ***Solution Procedure***

Let $r = r + 1$.

For $j = 1$ to $v + 1,\ v = |V_i|$

Calculate the first derivative lengths of the paths p, that is $d_p^{(r)},\ p \in P_i^k$.

End For.

Find the minimum first derivative length (MFDL) paths $\bar{p}_i^{(r)}$ among P_i.

For $j = 1$ to $v,\ v = |V_i|$

To path $p_j,\ p_j \in P_i,\ p_j \neq \bar{p}_i^{(r)}$, calculate the second derivative length: H_{p_j}.

Let $x_{p_j}^{n(r)} = max\{0,\ x_{p_j}^{n(r-1)} - \alpha^{(r-1)} H_{p_j}^{-1} (d_{p_j}^{(r-1)} - d_{\bar{p}_i^{(r-1)}}^{(r-1)})\}$.

Let $x_{ij}^{n(r)} = x_{p_j}^{n(r)},\ link\ (i,j) \in p_j$.

End For.

Let $x_{\bar{p}_i^{(r)}}^{n(r)} = \gamma_i^n - \sum_{p \in P_i,\ p \neq \bar{p}_i^{(r)}} x_p^{n(r)}$.

Step 3: ***Stopping Rule***

If $|D(x^{n(r)}) - D(x^{n(r-1)})|/D(x^{n(r)}) < \varepsilon$, then **stop**, where ε is a desired acceptable tolerance for solution quality; otherwise, go to Step 2.

by combining queue and rate adjustment policies. As mentioned earlier, QLBVR carries out coarse adjustments on job transferring and processing rates and fine adjustments on queue lengths (number of jobs in the queue). These are as described below.

1. Coarse adjustment (on transfer and processing rates): The working style is similar to RLBVR. At every time instant T_n $(n = 0, 1, \ldots)$, nodes in the system use the procedure for node to obtain an optimal solution. In the next time interval T_s, each node adjusts its job transferring rates and job processing rate according to the computed optimal solution.

2. Fine adjustment (on queue lengths): The working style is similar to ELISA. Each status exchange interval T_s is divided into equal subintervals denoted as estimation intervals T_e. At time instant T_e, node i estimates and adjusts the earlier estimate of the queue lengths of its neighboring nodes and has an accurate knowledge of its own queue length. It then calls the transfer procedure in ELISA for load balancing.

Remarks: Note that in distributed dynamic load balancing algorithms, there may exist a non-zero probability that a job would shuttle between processors. There are various ways to alleviate this problem. One of the ways is to allow the job to join at the position where it should have been if the job had arrived at this queue, instead of adding it at the end of the queue. This means that we keep track of the time at which it arrived at the system. This can considerably reduce the probability of the job being transferred once again and can guarantee minimizing the response time of that job.

4.3 Performance Evaluation and Discussions

We have conducted extensive simulation tests to quantify the performance of the three types of load-balancing algorithms discussed above. As mentioned earlier, one of our added considerations in this study is to gain an intuition regarding the relative metrics of the different approaches under consideration. Our simulations focus on two main aspects of dynamic sit-

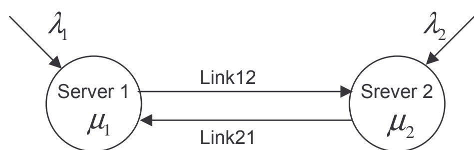

Figure 4.5: Model of a 2-processor system.

uation. One is the effect of the load on the nodes and the other is the effect of the status exchange interval lengths, which is indeed a crucial parameter that reflects the sensitivity of the algorithms. Here we consider the mean response time (MRT) of a job as the main metric in our simulations. First, we give the introduction of our simulation model.

4.3.1 Two-processor system model and some important issues

In order to illustrate the salient features of the algorithms in discussion, first we consider the simplest case when a multiprocessor system is limited to two processors, as shown in Fig. 4.5. But we do not imply any limitation on the size and parameters of the load-balancing algorithms. It is possible that this model could also be considered equivalent to a situation in which an isolated processor communicates to a (single) processor that represent the rest of the network. Another reason why a 2-processor network is used as a test case is that an algorithm that does not work for 2 processors is unlikely to perform well for a multi-processor case. It is relevant to mention that the two processors model considered could also represent a particular kind of multiprocessor model in which an isolated processor communicates to a single processor that represent the rest of the network.

In Fig. 4.5, we assume that jobs arrive at node 1 and node 2 according to Poisson processes with rates λ_1 and λ_2, respectively. We also assume that the average service times of a job at node 1 and node 2 are $1/\mu_1$ and $1/\mu_2$, respectively. From [28], we obtain that the average

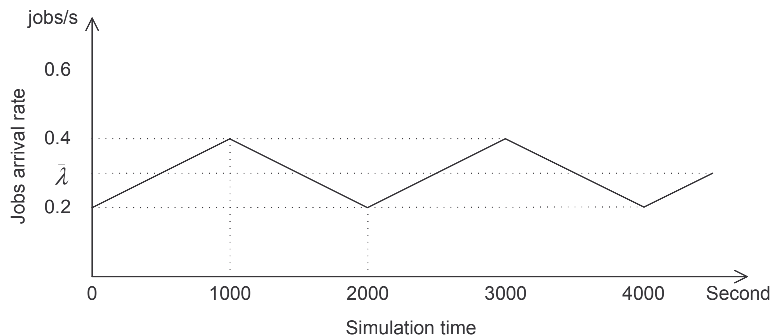

Figure 4.6: Job arrival rate pattern.

nodal delay of a job in node i ($i = 1, 2$) is:

$$S_i = \frac{1}{\mu_i - \lambda_i}, \tag{4.5}$$

where $\mu_1 = 0.25$ jobs/sec and $\mu_2 = 0.5$ jobs/sec, in our simulation tests.

In Fig. 4.5, jobs arrive at node 2 according to a homogeneous Poisson process with $\lambda_2 = 0.12$ jobs/sec, which will be held constant throughout our simulations. Without load balancing, the system utilization of node 2 is $\rho_2 = \frac{\lambda_2}{\mu_2} = \frac{0.12}{0.5} = 0.24$ and the system is at a low utilization level. In order to consider dynamic situations, the job arrival rate at a node must be a time-varying quantity. Thus, the job arrival rate at node 1 is an *inhomogeneous Poisson process* with intensity function $\lambda_1(t)$ [83]. The pattern of $\lambda_1(t)$ is shown in Fig. 4.6. At time 0, $\lambda_1(t)$ starts from the lowest point l_1. As time elapses, $\lambda_1(t)$ increases linearly. At time $t = 1,000$ second, $\lambda_1(t)$ reaches the peak h_1 and afterwards, $\lambda_1(t)$ decreases linearly. Finally, at time $t = 2,000$ second it reaches the lowest point l_1. After this point, $\lambda_1(t)$ increases again. For the sake of simplicity, we fix the difference between h_1 and l_1 to 0.2 jobs/sec. In order to describe the job arrival pattern quantitatively, we denote $\bar{\lambda}_1 = \frac{h_1 + l_1}{2}$ as the load of the dynamic job arrival rate. For example, we use $\bar{\lambda} = 0.3$ to consider the $\lambda(t)$ shown in Fig. 4.6.

For the communication links, we use M/M/1 model [27]. Assume that the transmission capacities of link $(1, 2)$ and link $(2, 1)$ are c_{12} and c_{21}, respectively. Assume that, on an average, A_1 packets are required to describe a job and that, on an average, A_2 packets are

required in sending back a response for a job. Here, we choose $A_1 = A_2 = A = 10$. From [12], we obtain the average communication delay of a job on link (i, j) is:

$$F_{ij}(x_{ij}) = \frac{Ax_{ij}}{c_{ij} - Ax_{ij} - Ax_{ji}} + \frac{Ax_{ji}}{c_{ji} - Ax_{ji} - Ax_{ij}}. \tag{4.6}$$

The first part of (4.6) signifies the communication delay of job flow on link (i, j) and the second part signifies the communication delay of the responses sent back from node j to node i. Here, we choose $c_{12} = c_{21} = 5,000$ (packets/sec).

For QAP and QRAP policies, the threshold θ is an important parameter. For a 2-processor system considered here, if θ is a threshold chosen and q_i is the queue length in node i $(i = 1, 2)$, then jobs are transferred from node 1 to node 2 if

$$q_1 - q_2 > \theta. \tag{4.7}$$

Hence, for a 2-processor system, the decision to transfer is taken if the difference in the queue lengths (estimated or actual) exceeds the specified threshold θ. In our simulations, a value of $\theta = 2$ seems more appropriate as we set both the queue lengths identical initially, as done in [19]. As we mentioned before, the status exchange interval T_s is divided into equal subintervals T_e and $T_s = m \times T_e$. In our simulations, we fix $m = 3$. The *window* used to estimate the job arrival rate is an integer w multiple of T_s. Here, we choose $w = 6$.

In the following experiments, we choose ELISA to present QAP policy in order to test the performances of the algorithms in the three policies with the same estimation method.

4.3.2 Effect of system loading

In this section, we will analyze the performances of the three policies with respect to the load on the system. First, in our simulation, we set $\bar{\lambda}_1 = 0.1$ jobs/sec and then increase the value of $\bar{\lambda}_1$ by 0.005 jobs per step. Also, we set $T_s = 60$ seconds and $T_e = 20$ seconds in our experiments. For each value of $\bar{\lambda}_1$, the simulation time is set to $20,000$ seconds to obtain a statistical mean response time of the jobs arriving at the system. When the average system

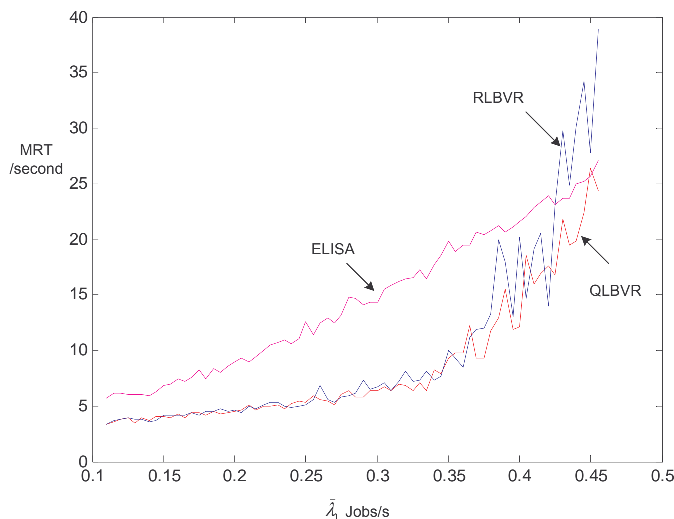

Figure 4.7: Effect of system loading.

utilization $\bar{\rho}$ is greater than 0.95, we terminate the simulation, where $\bar{\rho}$ is defined as the ratio of average total arrival rate to aggregate processing rate of the system: $\bar{\rho} = \frac{\lambda_1^n + \lambda_2^n}{\mu_1 + \mu_2}$.

The results of our experiments are shown in Fig. 4.7. From this figure, we can observe that when the load of the system is light or moderate ($\bar{\lambda}_1 < 0.4$ or the average system utilization $\bar{\rho}$ is below 0.75), RLBVR and QLBVR have a better performance than ELISA. For QAP policy, jobs in the queues can be transferred only at the time instant T_n or t_m. During the time interval T_e, even if the queue of node 1 is empty and the queue of node 2 is very long, the jobs in node 2 have to wait till the next time instant T_n or t_m to be transferred to node 1. Hence, in this situation, it is hard for QAP policy to obtain the optimal load balancing in the system. For RAP policy, each node in the system assigns the job arriving at it according to an optimal solution. Therefore, the probability that the queues in the system are kept balanced is fairly large and RAP policy can obtain a minimum (or near minimum) mean response time of the jobs. For QRAP policy, the coarse adjustment (rate adjustment) plays a crucial role and fine adjustment (queue adjustment) makes little influence on the mean response time of the jobs. This can be proven by the fact that when the system utilization is light or moderate, there is little difference between the MRT of RLBVR and QLBVR.

As shown in Fig. 4.7, when the load of job becomes high ($\bar{\lambda}_1 > 0.4$ or $\bar{\rho} > 0.75$), the MRT of RLBVR rises rapidly with severe fluctuations. With $\bar{\lambda}_1$ increases, when the utilization of the system is more than 90%, the system of RLBVR becomes more and more unstable. Because of RAP's random nature of job transferring, it is possible that in some cases, many jobs have been sent to node i and in the meanwhile, fewer jobs have been transferred to node j. This makes the queue of node i to grow rapidly and the queue of node j deplete quickly and even may become empty. Due to the high load, the utilizations of node 1 and 2 are nearly equal to 1 ($\rho_1 = \frac{\lambda_1^n}{\mu_1} \approx 1, \rho_2 = \frac{\lambda_2^n}{\mu_2} \approx 1$), which means that for node i or node j, its service rate is nearly equal to the job arrival rate. Hence, it is hard for node i to shorten the length of its queue and the MRT of jobs in the system becomes high. From this behavior, we can conclude that the performance of RLBVR is unanticipated when the system load is high. But, clearly this is never a problem for the algorithms of QAP and QRAP policies since balancing the queue lengths of the nodes in the system are effective. As shown in Fig. 4.7, ELISA and QLBVR have a much better performance than RLBVR when the load on the system is high.

From these simulation experiments, we can conclude that when the system load is light or moderate, the algorithms of RAP policy are preferable to obtain a minimum(or near minimum) mean response time of the jobs. If the load of system is high, the algorithms of QAP policy can achieve a better performance. The algorithms of QRAP policy are suitable in the cases when the system load is fluctuating.

4.3.3 Effect of T_s: Length of status exchange interval

The status exchange interval T_s is also an important parameter that decides the performance of dynamic load balancing algorithms. A large T_s indicates lower communication and computation overheads. Here, we will conduct some simulation experiments to analyze the effect of T_s on the three algorithms in the situations that system load is light, moderate and high, respectively. For each situation, we initiate T_s to be 2.1 seconds and in each step, we increase

T_s by 0.9. After 60 experiments, we terminate the simulation. The simulation time for each step is also set to 20,000 seconds.

We set $\bar{\lambda}_1 = 0.11$ jobs/sec to capture the behavior when the load on the system is light. In this case, the system utilization $\bar{\rho} < 0.35$. The simulation results are shown in Fig. 4.8. From this figure, we can point out that among the three algorithms, MRT of ELISA is much higher than those of RLBVR and QLBVR. There is only a little difference between the performances of RLBVR and QLBVR. As one would expect, with the status exchange interval T_s increasing, MRT of ELISA also increases. It is also interesting to observe that when T_s becomes longer, MRT of RLBVR and QLBVR slightly decreases. Because the window time to estimate job arrival rate $\hat{\lambda}$ is fixed (set to $6 \times T_s$), a large T_s value means longer window time for job arrival rate estimation and we can obtain more accurate $\hat{\lambda}$ for the pattern of job arrival rate, as shown in Fig. 4.6. If the job arrival rate changes more rapidly, we can expect that relatively shorter T_s will deliver a better result, when the loading on the system is light.

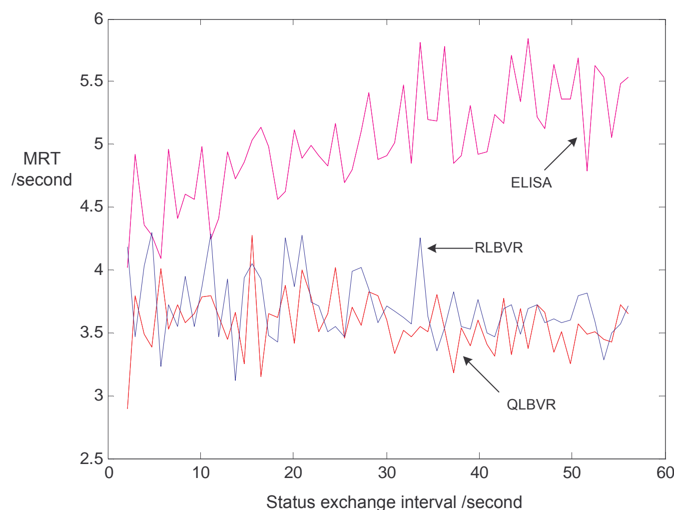

Figure 4.8: Effect of T_s: System loading is light.

We set $\bar{\lambda}_1 = 0.3$ jobs/sec to understand the behavior when the load on the system is moderate and the system utilization $0.35 < \bar{\rho} < 0.75$. The simulation results are shown in Fig. 4.9. From this figure, we can point out that ELISA is more sensitive to T_s than RLBVR

and QLBVR. With T_s getting larger, MRT of ELISA becomes higher quickly than that of RLBVR. Also, MRT of QLBVR is little higher than that of RLBVR. This is because of the fact that for QLBVR, it is possible that node i sends a job to node j due to its coarse adjustment, however node j may be sending another job back to node i due to its fine adjustment. The job shuttles between the nodes and increase the response time of the jobs. A small value of T_s means that more job shuttles between the nodes and the higher MRT of the jobs. From these simulation experiments, we can conclude that when the loading on the system is moderate, RLBVR has a much better performance than ELISA and QLBVR.

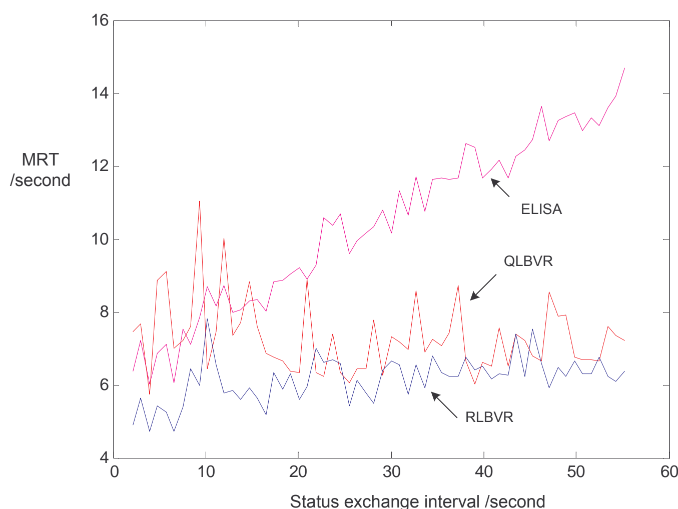

Figure 4.9: Effect of T_s: System loading is moderate.

Now when the load is high, i.e., we set $\bar{\lambda}_1$ to 0.45 jobs/sec. In this case, the system utilization $\bar{\rho}$ is nearly equal to 1. The simulation results are shown in Fig. 4.10. From this figure, we can observe that when the load of the system is high, the best performer is ELISA, the second best one is QLBVR, and the worst performance is by RLBVR. For RLBVR, because of the high load situation, it may be possible that many jobs may "jam" in one node and there is no mechanism to balance the queue sizes of the nodes in the system. Because of the characteristic of RLBVR, MRT of RLBVR is high and fluctuates rapidly with increasing T_s. For QLBVR, when the load is high, the fine adjustment play a crucial role than

coarse adjustment. Due to the shuttling of jobs, MRT of QLBVR is little higher than that of ELISA. As reported in [19], here too we observe that the lengths of T_s have a great effect on the performance of ELISA. We can see when T_s is 60 seconds, MRT of ELISA is nearly three times than that when T_s is 3 seconds. From these experiments, we also can observe that T_s is an important factor which can effect the system performance significantly. Due to the varying patterns of job incoming rates, it is hard to give a confidential interval of T_s here. However, if the job incoming rates change rapidly, we can choose a shorter T_s to capture a better knowledge of the job rates, whereas, if the job incoming rates change slowly, we can choose a longer T_s in order to reduce the communication overheads.

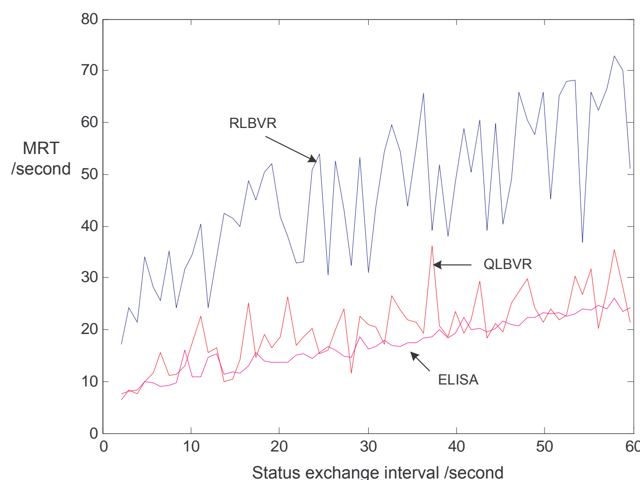

Figure 4.10: Effect of T_s: system loading is high.

4.4 Extensions to Large Scale Multiprocessors System

Now we shall consider a large scale multiprocessor system to capture and understand the behavior of our algorithms. This is an important study as it reflects a real-life scenario. In our study, we considered a mesh architecture. Mesh-connected multiprocessor is one of the most suitable architectures for multiprocessor systems of small or moderate size. As

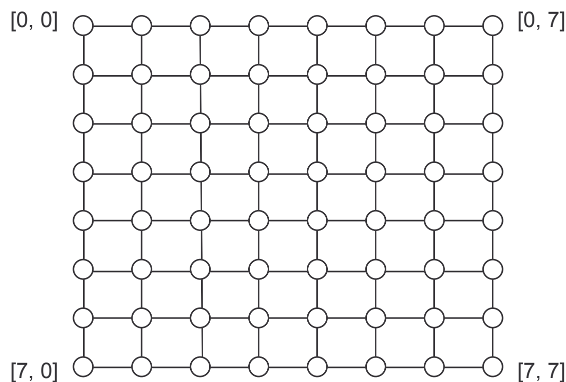

Figure 4.11: A mesh-connected multiprocessor $M[8, 8]$ system.

mentioned earlier, based on the mesh topology, many prototype and commercial systems delete the sentence. such as Dash [84], Paragon [85], and Intel Touch-stone Delta have been built and these architectures are able to handle data intensive computations. Hence, in this section, we extend our simulation experiments to a mesh multiprocessor system, as shown in Fig. 4.11. We will compare the performances of the three policies under different system workload in a mesh-connected multiprocessor system.

A two-dimensional mesh, denoted by $M[X, Y]$, comprises $X \times Y$ processors arranged in a $X \times Y$ two-dimensional grid, where each node represents a processor and each edge denotes a two-way communication link. The processor located in row x and column y is identified by coordinate (x, y). In these simulations, we use the same nodal model and link model described in Chapter 4.3. The processing capability of the processors in the mesh $M[X, Y]$ can be represented by the "processing rate" array denoted as $\mu[X, Y]$, in which element $\mu(x, y)$ has the value of jobs processed per second. In our simulations, we set $X = Y = 8$ (a 64-processor system) and the job processing rate of each node in the system is randomly generated with uniform distribution between 4 and 10, as shown in (4.8). For convenience, we still use node i, node j to denote the node in the system. The capacities of links in the system are also randomly generated with a uniform distribution between $50,000$ and $150,000$ packets per second.

$$\mu = \begin{bmatrix} 7 & 5 & 7 & 6 & 10 & 5 & 9 & 8 \\ 9 & 8 & 5 & 4 & 6 & 5 & 5 & 10 \\ 7 & 6 & 8 & 8 & 7 & 4 & 5 & 6 \\ 4 & 6 & 9 & 10 & 5 & 5 & 4 & 6 \\ 4 & 5 & 10 & 5 & 4 & 5 & 8 & 8 \\ 5 & 7 & 6 & 5 & 5 & 10 & 7 & 10 \\ 5 & 5 & 6 & 4 & 5 & 6 & 10 & 5 \\ 4 & 7 & 6 & 10 & 10 & 8 & 7 & 9 \end{bmatrix}. \quad (4.8)$$

Again, we use (4.6) as the function of the average communication delay of a job on the links. In this section, we choose *Perfect Information Algorithm* (PIA) to present QAP, instead of ELISA. PIA is similar to ELISA, but it is assumed that at the transfer epochs, each node has perfect information about the state of every other neighboring node. In [19], it has been proven that PIA has better performances than ELISA at the expense of more communication overheads and PIA can present the best performance of QAP algorithms when the MRT is the only metric. Let PIA(1) to denote the PIA algorithm whose status exchange interval is 1 second, and PIA(3) and PIA(10) to denote the PIA algorithms whose status exchange interval are 3 and 10 seconds, respectively. Again, we use RLBVR(30) to present RAP, which is RLBVR algorithm with $T_s = 30$ seconds. We use QLBVR(30) to present QRAP which is QLBVR algorithm with $T_s = 30$ seconds. In QLBVR(30) algorithm, every node in the system balances its queue with its neighboring nodes once in each status exchange interval by the mechanism similar to ELISA. In the following, we will conduct some simulations for all the 5 algorithms under 2 types of jobs loads: (a) arrival of loads are static or slowly varying; (b) arrival of loads is varying rapidly.

4.4.1 Static or slowly varying system loading

Here, we assume that jobs arrive at node (i, j) according to a homogeneous Poisson process with $\lambda(i, j)$ jobs/sec. Under a given set of $\lambda[X, Y]$, the system utilization is:

$$\rho = \frac{\sum_{i=0}^{7} \sum_{j=0}^{7} \lambda(x, y)}{\sum_{i=0}^{7} \sum_{j=0}^{7} \mu(x, y)}. \tag{4.9}$$

We carry out a series of simulations for the $M[8, 8]$ mesh-connected multiprocessor system with the 5 algorithms described above, under different system utilization parameter ρ. The simulation conditions are kept identical for all the 5 algorithms for a given ρ. For each simulation run, the simulation time is set to $20,000$ seconds, during which, the first $5,000$ seconds are considered as "warm time". After the warm time, we trace the jobs arriving at each node and record their arriving time, processing time and leaving time. We average $10,000$ jobs' response time as the mean response time for this simulation. In the first simulation, the job arriving rates $\lambda[X, Y]$ are randomly generated. Then, each node (x, y) increases its $\lambda(x, y)$ by 0.25 jobs per second in each step. When the system utilization exceeds 0.9, we terminate the simulations.

Fig. 4.12 and Fig. 4.13 show the simulation results of the mean response time of jobs for the 5 algorithms. We observe that when the system utilization is light or moderate (less than $\rho = 0.75$), PIA (QAP policy) results in much higher mean response time than QLBVR(30) (QRAP policy) and RLBVR(30) (RAP policy). Also, a small T_s can improve the performance of PIA only when the system utilization becomes moderate (greater than $\rho = 0.3$), because when the system utilization is light, the queue length of each node is very short and the queue adjustment makes little effect on the MRT of the jobs no matter how large the value of T_s is set. Due to the slow varying nature of the job pattern, QLBVR(30) and RLBVR(30) exhibit much better performances with a relatively longer $T_s = 30$ and there is little difference between the MRTs of the two algorithms. When the system utilization becomes high (greater than $\rho = 0.75$), PIA(1) has the best performance of all the 5 algorithms. QLBVR(30) renders

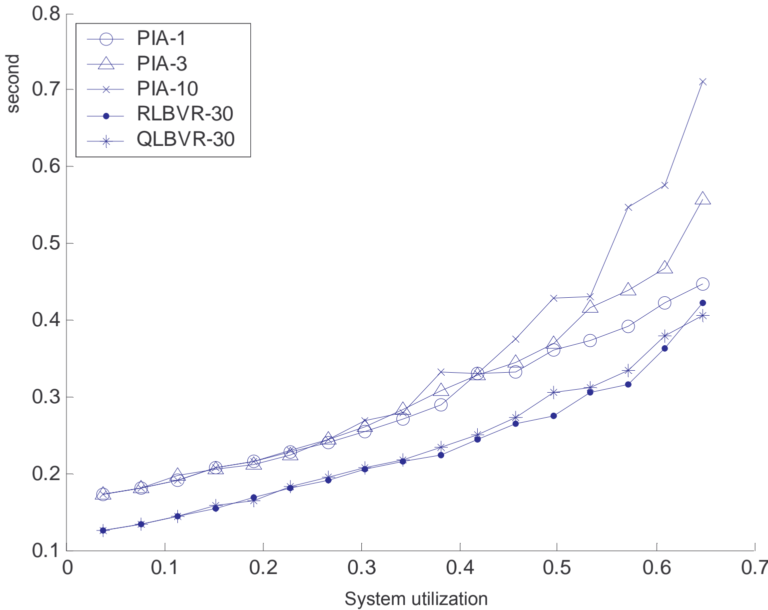

Figure 4.12: Mean response time of jobs for 5 different algorithms under different system utilization: System utilization is light or moderate ($\rho < 0.75$).

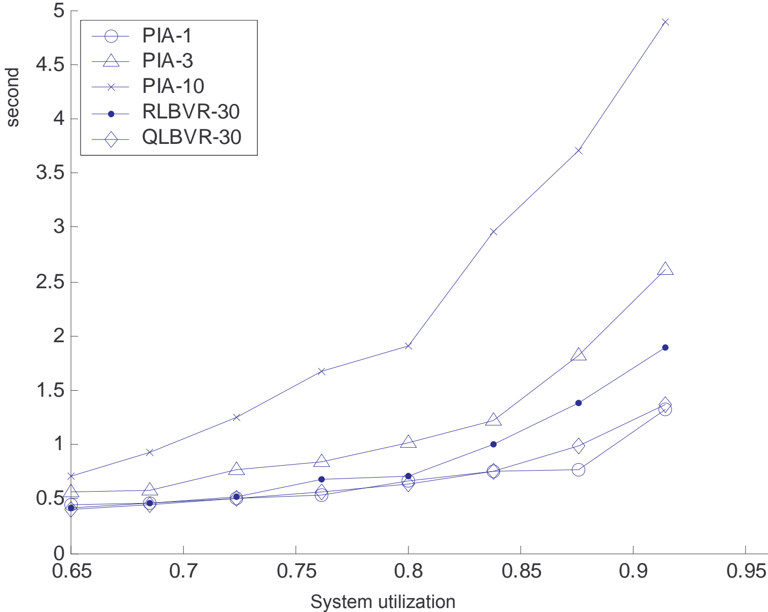

Figure 4.13: Mean response time of jobs for 5 different algorithms under different system utilization: System utilization is high ($\rho > 0.75$).

the second best performance, whose MRT is little higher than that of PIA(1). The worst performer is PIA(10). However, if we consider the communication overheads, we recommend QRAP instead of QAP, because QLBVR(30) has nearly the same performance as PIA(1), with much longer T_s than PIA(1) (which indicates much lower communication overheads).

4.4.2 Experiments when arrival of loads is varying rapidly

We construct another job pattern to simulate the situation when the job arriving rate of each node changes rapidly. We assume that for each node, the time interval for a batch of jobs is 10 seconds, during which jobs arrive at the node according to a homogeneous Poisson process. In each 10 seconds time interval, the probability that the arrival rate will be same as in its previous interval is 0.1, otherwise, the job arriving rates are generated randomly with uniform distribution between λ_{min} and λ_{max}. We assume that, in each node, the maximum job arriving rate cannot exceed its maximum job processing rate. For node (x, y), we use r_{min} to denote the ratio $\frac{\lambda_{min}}{\mu(x,y)}$ and r_{max} to denote the ratio $\frac{\lambda_{max}}{\mu(x,y)}$. Hence, we can use $[r_{min}, r_{max}]_{(x,y)}$ to denote this job pattern for node (x, y) in the system. For example, the job pattern shown in Fig. 4.14 can be denoted as $[0.3, 0.7]_{(7,7)}$ for node $(7, 7)$ whose job processing rate is 9 jobs per sec. Again, let $\bar{\rho}$ be the average system utilization for our simulations, which is:

$$\bar{\rho} = \frac{number\ of\ jobs\ arriving\ /\ simulation\ time}{\sum_{i=0}^{7}\sum_{j=0}^{7}\mu(x,y)}. \tag{4.10}$$

We conducted 4 kinds of simulation tests indicated as S.N. (simulation number), shown in the following.

A: System utilization is light. In this case, the job pattern of each node (x, y) in the system is $[0.1, 0.4]_{(x,y)}$ and $E(\bar{\rho}) = \frac{0.1+0.4}{2} = 0.25$.

B: System utilization is moderate. In this case, the job pattern is $[0.3, 0.7]_{(x,y)}$ for node (x, y) and $E(\bar{\rho}) = \frac{0.3+0.7}{2} = 0.5$.

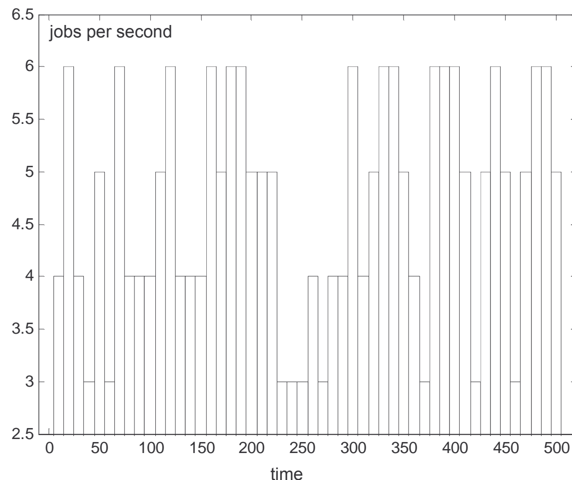

Figure 4.14: An example of the job pattern.

C: System utilization is high. In this case, the job pattern is $[0.7, 1]_{(x,y)}$ for each node (x, y) in the system and $E(\bar{\rho}) = \frac{0.7+1}{2} = 0.85$.

D: We randomly choose $\frac{1}{3}$ of the nodes as lightly-loaded nodes ($[0.1, 0.4]$), $\frac{1}{3}$ of nodes as moderately-loaded nodes ($[0.3, 0.7]$) and the rest $\frac{1}{3}$ of nodes as highly-loaded nodes ($[0.7, 1]$) and $E(\bar{\rho}) = \frac{0.25+0.5+0.85}{3} = 0.533$.

In each kind of simulations, for all the 5 algorithms, we generate a set of job arriving rate sequences to ensure that the simulation conditions remain identical. The jobs recording method is same as that described in Chapter 4.1. The simulation results of jobs' MRT are shown in Table 4.3.

From these simulations, we can observe that when $\bar{\rho}$ is light (A), the best performers are RLBVR(30) and QLBVR(30) and the performances of PIA(1), PIA(3) and PIA(10) are nearly the same. However, when $\bar{\rho}$ is moderate (B), the best performer is still RLBVR(30) and the second best one is PIA(1). The performances of the rest algorithms are nearly the same. When $\bar{\rho}$ is high (C), PIA(1) is much better than RLBVR(30) and QLBVR(30). But if T_s increases, the performance of PIA also decreases. We can observe that the difference between PIA(10) and QLBVR(30) is very little (within 10%). In D, the best ones are RLBVR(30)

and QLBVR(30) and the worst one is PIA(10).

Table 4.3: MRT of 5 algorithms in the 8 × 8 mesh multiprocessor system (sec.)

S.N.	$\bar{\rho}$	PIA(1)	PIA(3)	PIA(10)	RLBVR(30)	QLBVR(30)
A	0.253	0.2045	0.2083	0.2080	0.1789	0.1865
B	0.500	0.3067	0.3119	0.3203	0.2962	0.3282
C	0.851	0.4800	0.6243	0.8342	1.0266	0.9063
D	0.541	0.3420	0.4051	0.4511	0.3266	0.3353

4.5 Concluding Remarks

In this chapter, we classified the distributed dynamic load balancing algorithms into three policies: QAP policy: queue adjustment policy, RAP policy: rate adjustment policy, and QRAP policy: combination of queue and rate adjustment policy. We proposed two algorithms named Rate based Load Balancing via Virtual Routing (RLBVR) and Queue based Load Balancing via Virtual Routing (QLBVR), which belong to RAP and QRAP, respectively. These two algorithms are based on LBVR and hence the computation overheads are small. We have used Estimated Load Information Scheduling Algorithm (ELISA) [19] to present QAP policy, the main idea of which is to carry our estimation of load by reducing the frequency of status exchange, thereby reduces the communication overheads.

We constructed a dynamic job arrival rate pattern and carried out rigorous simulation experiments to compare the performance of the three algorithms under different system loads, with different status exchange intervals T_s. With our rigorous experiments, we have shown that when the system loads are light or moderate, algorithms of RAP policy are preferable with longer T_s. When the system loads are fairly high, QAP policy and QRAP policy have better performances than RAP policy. In this situation, QAP and QRAP policies can obtain

smaller mean response time of jobs with shorter T_s. We also demonstrated that QRAP has relatively good performance in most of the situations and it is suitable when the system loads are fluctuating. We have also extended our rigorous simulation tests for a large scale multiprocessor system assuming a mesh-architecture. In this case, we used Perfect Information Algorithm (PIA) to study the QAP policy, which delivered the best performance of QAP with respect to MRT metric [19]. From our experiments, we have clearly identified the relative metrics of the performance of the proposed algorithms and we are able to recommend the use of suitable algorithms for different loading situations. Our system model and experimental study can be directly extended to large size networks such as multidimensional hypercubes networks to test their performances. Finally, in this chapter, we have rigorously demonstrated the performance of the algorithms for a single class of jobs.

In our future works, we can divide the jobs in the system into several classes and assign each class of jobs its own priority. It would be interesting to consider multi-class jobs system as well and analyze the performance of these algorithms. We only consider infinite buffer size in this chapter. We can study the performance of the algorithms with limited buffer size. Job's slow-down [81], which is the ratio of the response time to the processing time of a job, is an important fairness index for distributed systems. We also can do more research work on this metric for the proposed algorithms.

Chapter 5

Extensions to Divisible Loads Scheduling on Arbitrary Networks

We have extensively studied the indivisible load balancing problems on arbitrary network configurations in both *static* and *dynamic* situations. We also have proven that the distributed strategies via virtual routing are efficient to solve the load balancing problems. Now, we extend our research work on divisible load theory (DLT) [14, 15, 63]. In this chapter, we consider the problem of scheduling divisible load originating from single or multiple sites on heterogenous arbitrary networks. Our objective is to design efficient distributed load distribution strategies so as to minimize the total processing time of all the loads given to the distributed computer system for processing. In 1998, Barlas [51] presented an important result concerning an optimal sequencing in a tree network and it is one of the important studies that demonstrates the performance of the load distribution algorithm when result collection phase is included in the problem formulation. In [52], a multi-level tree is considered and it is assumed that the load distribution takes place concurrently from a source processor to all its immediate child processors. This is one of the earliest attempts in using a *multi-port* model of communication, in which all the incident links on a processor are concurrently used. The advantage of the multi-port model was also demonstrated in a two-dimensional mesh network [53]. In [54],

load partitioning of intensive computations of large matrix-vector products in a multi-cast bus network was investigated.

Scheduling the input loads which comprise of both divisible and indivisible loads was also considered in the literature and heuristic algorithms were proposed in [76]. In [90], the authors introduced simultaneous use of links to expedite the communication and the concept of *fractal hypercube* on the basis of *processor isomorphism* to obtain the optimal solution with fewer processors. Most recent studies focus on system dependent constraints such as, scheduling under finite buffer capacity constraints and processor release times [56], estimating the processor and link speeds by some methods of probing and estimating [57], scheduling divisible loads on Mesh multiprocessor architectures [58], to quote a few. Also, the applicability of DLT concepts to schedule and process loads generated from large scale physics experiments (RHIC at BNL, USA) on computational grids are investigated in [59]. Some other studies in the domain of DLT include scheduling divisible loads with arbitrary processor release times in bus networks [86], use of affine delay models for communication and computation for scheduling divisible loads [61], etc.

Finally, use of affine delay models for communication and computation components for scheduling divisible loads are extensively studied in [60, 61], etc.

All the above studies focussed their attention in designing load distribution strategies under the assumption that all loads originate at one processor. For the solution approaches, the algorithms proposed in the literature were centralized. That is, according to the network topology and the node where the loads originate, one processor in the system will determine the strategy of scheduling. In this chapter, we will consider a distributed load scheduling problem with divisible loads originating from single or multiple processors on arbitrary networks. This scenario happens commonly in realistic situations, such as applications in distributed real-time systems.

The organization of the chapter is as follows. In Chapter 5.1, we present a mathematical

model, the problem definition and we formulate the problem. In Chapter 5.2, we propose our strategy and derive conditions for obtaining an optimal solution. We prove several important properties used in the design of the proposed algorithm and prove its convergence and complexity. In Chapter 5.3, we present a detailed illustrative example to show the complete workings of our proposed algorithm for ease of understanding. We shall also present numerical studies to quantify the performance in the cases that when divisible loads originate from single and multiple sites. We highlight our contributions and discuss on possible future extensions in Section 5.4.

5.1 Mathematical Model and Problem Formulation

In this section, we shall formally introduce the problem we are tackling and propose a mathematical model used in our analysis. We will also introduce the required terminology, notations and definitions that will be used throughout this chapter. In the following, we shall first describe the characteristics of the system, the resource constraints, and some assumptions that are considered in the problem context.

5.1.1 Description of system, assumptions and notations

We consider the problem of scheduling and processing divisible loads on an arbitrary computer network system as shown in Fig. 5.1. The system consists of n heterogeneous nodes, which represent host computers, interconnected by a generally configured communication or intercommunication network.

We assume that all nodes in the system have front-end. We only adopt uni-port model, which means each node can take care of transferring the load to its neighboring nodes one-at-a-time. The assumption is reasonable, because the buffer size of node is limited and it has not enough load to send to all of its neighboring nodes simultaneously. If node i has no load

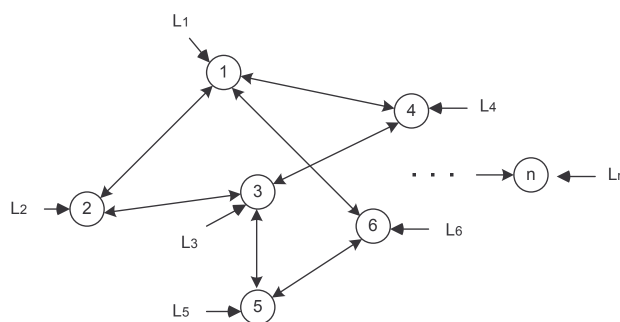

Figure 5.1: An arbitrary computer network system with multiple loads.

to be processed initially, node i cannot start processing until it receives all the loads. This assumption is also reasonable, because when the node is receiving the loads, it doesn't know which part of the loads should be processed locally and which part should be transferred to which neighboring node [14, 15, 59]. When node i receives all the loads, it can start sending the loads to its neighboring nodes one by one and processing at the same time. Note that in [14], the authors assumed that each node can receive loads from only one neighboring node at a time. In our model, we relax this constrain and assume that each node is provided with such a facility, which is capable of receiving (gathering) all the load fractions simultaneously over all the incident links on it.

We consider the divisible loads originating from single or multiple nodes in the network and denote L_i as the amount of loads originating at node i for processing, where $L_i \geq 0$ holds. Let l_i denote the amount of loads assigned to node i according to a scheduling strategy and x_{ij} denote the amount of loads should be transferred from node i to node j, $j \in V_i$. Hence, the following conservation equations hold:

$$\gamma_i = L_i + \sum_{j \in V_i} x_{ji} = l_i + \sum_{j \in V_i} x_{ij}, \tag{5.1}$$

where γ_i is the total amount of loads at node i. In general, there is a node delay incurred when l_i jobs are processed at node i and a communication delay incurred when x_{ij} jobs are transferred on link (i, j). In DLT, the above two kinds of delay functions are assumed to be

linear, increasing functions [14–16, 70]. Without loss of generality, we shall denote $P_i(l_i) = E_i \times l_i$ as the node delay function for node i and $G_{ij}(x_{ij}) = C_{ij} \times x_{ij}$ as the communication delay function on link (i, j), where, E_i is the time it takes to compute a unit load by node i and C_{ij} is the time it takes to transfer a unit load on link (i, j), respectively.

Our objective is to obtain an optimal load distribution such that the processing time of the entire loads is a minimum. For each node i, the processing time T_i is dependent on l_i and t_i^{max}, where t_i^{max} is the time instance at which the last job is received by node i. Obviously, if no job is sent to node i by its neighboring nodes, $t_i^{max} = 0$ holds. Now, we can obtain:

$$T_i = t_i^{max} + E_i \times l_i. \tag{5.2}$$

We denote $t_i^{max}(j)$ as the time instance at which the last job coming from node j is received by node i. According to the definition of t_i^{max}, we obtain the following equation:

$$t_i^{max} = max\ \{t_i^{max}(j),\ \mid \forall j \in V_i\}. \tag{5.3}$$

We can observe that t_i^{max} may be determined by node j. Let t_{i-j}^{max} denote the time instance at which the last job coming from V_i, *except* node j, is received by node i.

As we mentioned before, when node i receives all the loads, it can start processing and transferring some loads to one of its neighboring nodes simultaneously according to the scheduling. In the DLT literature [14, 15], it has been shown that the processing time can be improved if we follow a sequence in which link speeds decrease (referred to as, optimal sequence). Thus, we can observe that it is important for node i to obtain such an optimal sequence, following which node i should send loads to its neighboring nodes one by one. We will discuss the optimal sequence in Chapter 5.2 in detail. Now, we use S_i to re-index the nodes of V_i, the global number of the neighboring nodes, as $S_i = \{1, 2, \ldots, v_i\},\ v_i = |V_i|$, for ease of mathematical representation. Thus, we can conceive an inverse function that maps the nodes of S_i to V_i and we denote such an inverse function of $S_i(j)$ as $S_i^{-1}(k)$. Thus, $S_i(j) = k$ and $S_i^{-1}(k) = j$ imply $k \in S_i$ and $j \in V_i$, respectively. For example, referring to Fig. 5.1, node

1 has three neighboring nodes and $V_1 = \{2, 4, 6\}$. For any sequence of node 1, $S_1 = \{1, 2, 3\}$. If node 1 send load to node 2 first, then node 6, and send load to node 4 last. We can obtain that $S_1^{-1}(1) = 2,\ S_1^{-1}(2) = 6,\ S_1^{-1}(3) = 4$. If we know node 2 is in V_1, we also can obtain the position of node 2 in S_1 by $S_1(2) = 1$. Hence, if node j, $j \in V_i$, sends loads to node i, the following equations hold:

$$t_i^{max}(j) = t_j^{max} + \sum_{k=1}^{S_j(i)} C_{jm} \times x_{jm},\ m = S_j^{-1}(k). \tag{5.4}$$

From (5.3), (5.4) and from the definition of t_{i-j}^{max}, we obtain:

$$\begin{aligned} t_i^{max} &= max\ \{t_i^{max}(j),\ t_{i-j}^{max}\} \\ &= max\ \{t_j^{max} + \sum_{k=1}^{S_j(i)} C_{jm} \times x_{jm}, \mid \forall j \in V_i,\ x_{ji} > 0,\ m = S_j^{-1}(k)\}. \end{aligned}$$

Note that the processing time of the system is influenced by l_i and the transfer amount x_{ij}. Now we shall present a formal definition of the problem addressed in this Chapter.

5.1.2 Problem formulation

Now, we shall formally define the problem that we want to address. In essence, we formulate the problem as a real-valued optimization problem with the objective of minimizing the processing time of the system according to (5.2). We state the following:

$$Minimize: \quad D(l, x) = max\ \{T_i = t_i^{max} + E_i \times l_i,\ \mid \forall i \in N\}, \tag{5.5}$$

$$where,\ t_i^{max} = max\ \{t_j^{max} + \sum_{k=1}^{S_j(i)} C_{jm} \times x_{jm}, \mid \forall j \in V_i,\ x_{ji} > 0,\ m = S_j^{-1}(k)\}.$$

$$Subject\ to: \quad L_i + \sum_{j \in V_i} x_{ji} = l_i + \sum_{j \in V_i} x_{ij}, \tag{5.6}$$

$$\sum_{i \in N} L_i = \sum_{i \in N} l_i, \tag{5.7}$$

$$l_i \geq 0,\ i \in N, \tag{5.8}$$

$$x_{ij} \geq 0,\ (i, j) \in C. \tag{5.9}$$

The equation (5.6) defines load balancing constraints, each of which equates the amount of loads in and out of node i. To the best of our knowledge, it is the first time in DLT literature that the scheduling problem is formulated as an optimal function. In order to obtain an optimal solution to the problem defined by (5.5), each node i can determine l_i and $x_{ij},\ j \in V_i$, according to the total amount jobs γ_i. For node i, another important parameter t_i^{max} (independent on node i) is only determined by the incoming amount loads $x_{ji},\ j \in V_i$. Hence, the load scheduling strategy should determine the values of l, x, and the optimal sequence of each node sending jobs to its neighboring nodes, where, $l = [l_1, l_2, \ldots, l_n]$ and $x = [x_{ij}],\ (i,j) \in C$.

We shall now introduce the main idea we used to attack the problem. In [10], the authors proposed a distributed Load Balancing algorithm via virtual routing. In [87], the divisible load scheduling problem on single-level tree topology was analyzed. Note that, if we use a distributed method to solve the problem addressed in this chapter, each node will only consider itself and its neighboring nodes. If node i has loads to send to its neighboring nodes, we can obtain a reduced single-level tree (star) graph with node i as the root processor. For example, referring to Fig. 5.1, if node 1 has loads to send to node 2, 4, 6, we can obtain a sub-graph as shown in Fig. 5.2. Then, node 1 can calculate l_1, and $x_{12},\ x_{14},\ x_{16}$ by virtual routing method proposed in [10], which has been proven to be an efficient distributed search method. All the nodes in the system construct their own single-level tree and then calculate the l and x according to the optimal sequences one by one. After some iterations, an optimal solution can be found, if it exists. Thus, all main procedures of our proposed strategy has been so far described.

In the following section, we will present our solution approach, discuss the optimal sequence and prove the convergence of our strategy.

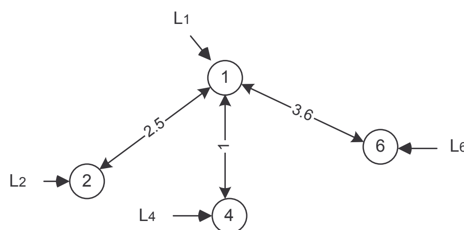

Figure 5.2: Single-level tree topology.

5.2 Proposed Strategy for Optimal Solution

In this section, we will propose our solution approach the problem defined in (5.5). We shall first present the sub-algorithm for each node to search for an optimal solution in each iteration and then discuss the optimal sequence, based on which we construct our scheduling strategy. After that, we will present the scheduling strategy for divisible load originating from single site first and discuss the difference between divisible load originating from single and multiple sites, for ease of simplicity. After that, we will present the scheduling strategy for divisible loads originating from multiple sites. At the end of this section, we will prove the convergence of our strategy and deduce the complexity of the proposed algorithm.

5.2.1 Sub-algorithm for a node and optimal sequence

As proved in [14, 15], a set of l and x is an optimal solution to problem (5.5) *if and only if* all the nodes in the system stop processing at the same time. For our distributed scheduling strategy, each node considers its neighboring nodes and constructs a star sub-graph. Each node will try to obtain a local optimal solution among the sub-graph. As a first step, we present our sub-algorithm of a node to make all the nodes in the sub-graph stop processing at the same time.

Our sub-algorithm of node is based on the virtual routing method proposed in [10]. In our sub-algorithm, each node i in the system will attempt to obtain a local optimal solution

to make node i and node j, $j \in V_i$, to stop processing at the same time. At first, we assume that $L_i > 0$ and $L_j = 0$, and besides node i, no other node will send jobs to node j. Now we obtain the sub-problem of problem (5.5) for an optimal sequence $\bar{S}_i$ as follows:

$$\begin{aligned} Minimize: \ D_i(l_i, x_i) &= max\{T_i, \ T_j, \ | \ \forall j \in V_i\}. \\ T_i &= E_i \times l_i, \\ T_j &= \sum_{k=1}^{S_i(j)} C_{im} \times x_{im} + E_j \times l_j, | \ \forall j \in V_i, \ x_{im} > 0, \ m = \bar{S}_i^{-1}(k). \end{aligned} \tag{5.10}$$

$$Subject\ to: \quad L_i = l_i + \sum_{j \in V_i} l_j, \quad with \ \ (5.6), \ \ (5.8), \ \ (5.9).$$

The sub-algorithm will then search for the solution of l_i, l_j $(= x_{ij})$ to make all the nodes among the sub-graph stop processing at the same time. Now we add a *virtual* node, which is referred to as the *destination* node (node d), into our network system. Further, we connect node d with node i and j, $j \in V_i$, by direct *virtual* links (i, d) and (j, d). We use functions $G_{id}(x_{id}) = E_i \times x_{id}$ and $G_{jd}(x_{jd}) = E_j \times l_j$ as the delay functions of links (i, d) and (j, d), respectively. Note that node i has a set of routing paths P_i, through which node i can send jobs to node d. Let p_i^0 denote the routing path of $i \rightarrow d$ directly and p_i^k denote the routing path of $i \rightarrow j \rightarrow d$, where $j = \bar{S}_i^{-1}(k)$. We denote x_i^k as the amount jobs that are transferred on path p_i^k and denote d_i^k as the time delays on paths p_i^k. Obviously, $x_i^0 = l_i$, $x_i^k = x_{ij}$, and $d_i^0 = T_i$, $d_i^k = T_j$ hold, where $k = 1, 2, \ldots, v_i$, $j = \bar{S}_i^{-1}(k)$. After this transformation, we can describe the process of scheduling problem in another way. To obtain an optimal solution to the sub-problem (5.10), node i shall determine the amount jobs which are transferred on each path $p_i^k \in P_i$ and make the time delays of all the routing paths to be the same, which is:

$$\bar{d}_i^0 = \bar{d}_i^1 = \ldots = \bar{d}_i^{v_i}. \tag{5.11}$$

Let d_i^{min} denote the Minimum Delay Routing Path (MDRP) of node i, which is min $\{d_i^k, \ | \ k = 0, 1, \ldots, v_i\}$. Clearly, an optimal solution of problem (5.10) results only if amount loads travel along MDRP paths in P_i. The amount of loads is strictly suboptimal *only if* there is a positive

amount loads that travel on a non-MDRP path. This suggests that a suboptimal solution can be improved by transferring some amount loads to an MDRP path from other paths in P_i (evident from (5.11) above). However, by transferring all the loads to the MDRP path will lead to an oscillatory behavior. Thus, it would be better to transfer only part of the loads from other paths via MDRP path. Below, we shall determine the amount loads from each of the non-MDRP paths in P_i to be transferred via MDRP to seek an optimal solution. From [10, 27], we obtain that the iteration takes the form:

$$x_i^{k(r+1)} = max\{0,\ x_i^{k(r)} - \alpha^{(r)} \frac{d_i^{k(r)} - d_i^{min(r)}}{H_i^k}\},\ k = 0, \ldots, v_i,\ k \neq min, \quad (5.12)$$

$$x_i^{min(r+1)} = \gamma_i^{(r+1)} - \sum_{k=0, k\neq min}^{v_i} (x_i^{k(r+1)}), \quad (5.13)$$

where, $\alpha^{(r)}$ is a positive scalar step-size in r-th iteration, according to some rule [62], min presents the MDRP path in set $d_i^{(r)}$ and

$$H_i^k = \frac{\partial d_i^{k(r)}}{\partial x_i^{k(r)}} + \frac{\partial d_i^{min(r)}}{\partial x_i^{min(r)}}.$$

We use the linear section search method in this chapter. One can also choose other linear section search such as Golden Section Search or find a closed-form equation [14] to solve the problem (5.10) for a given sequence. However, these methods may be good for the single-level network, however may be not suitable for the arbitrary topology discussed in this chapter due to slow convergence rate. It is beyond the scope of this thesis to discuss these.

Example 5.1**:** For ease of understanding, we consider an example as shown in Fig. 5.2, in which $L_1 = 100$ units loads and $L_2 = L_4 = L_6 = 0$ and node 1 tries to send some jobs to its neighboring nodes. The processor speed parameters are $E_1 = 2.0$, $E_2 = 4.0$, $E_4 = 1.0$, $E_6 = 1.5$ and the respective link speed parameters are marked in the graph. At first we choose a random sequence for node 1, as sending loads to node 2 first, and then node 4, 6 one by one. We obtain $S_1(2) = 1$, $S_1(4) = 2$ and $S_1(6) = 3$. After adding the virtual destination node d and the virtual links $(1, d)$ and (j, d), $j \in V_1$, we obtain Fig. 5.3, where P indicate the routing paths of node 1. In order to obtain (5.11), we first set an initial feasible solution

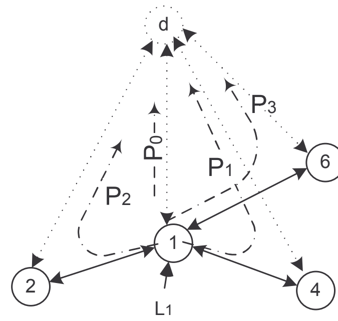

Figure 5.3: Single-level tree with virtual node and virtual links.

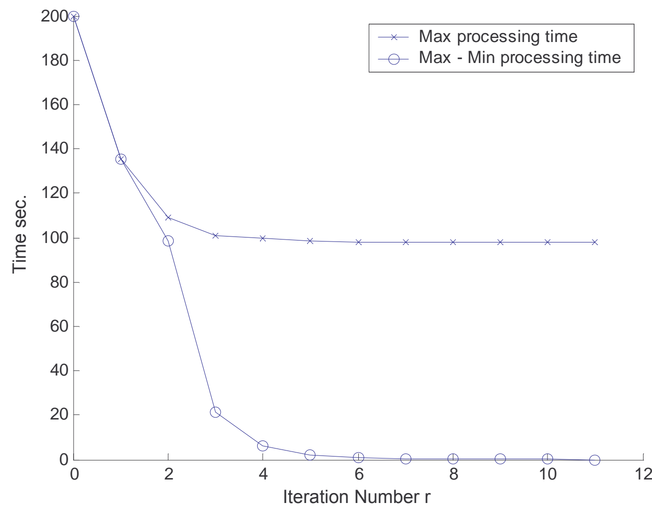

Figure 5.4: Results of Example 5.1: d_1^{max} and $d_1^{max} - d_1^{min}$ in each iteration r.

that is $x_1^{(1)} = [x_1^{0(0)}, x_1^{1(0)}, x_1^{2(0)}, x_1^{3(0)}] = [100, 0, 0, 0]$, $r = 0$ firstly. Then use (5.12) and (5.13) to search for the optimal solution. We obtain the results as shown in Fig. 5.4, in which max processing time is d_1^{max} and max−min processing time is the difference between the maximum and minimum processing time among the paths of node 1. After 11 iterations, we can observe that the time delay on each path becomes the same that is $\bar{d}_1^0 = \bar{d}_1^1 = \bar{d}_1^2 = \bar{d}_1^3 = 97.86$ seconds, which means all the nodes in this sub-graph will stop processing at time 97.86 seconds. The solution for this example is $l_1 = 48.93$, $x_{12} = 15.06$, $x_{14} = 30.11$ and $x_{16} = 5.90$ units loads.

Theorem 5.1 (Optimal Sequence). *In a single-level tree network, in order to achieve minimum processing time, the sequence of load distribution by the root node i should follow the order in which the link speeds ($\frac{1}{C_{ij}}, j \in V_i$) decrease* [15, 87].

Theorem 5.1 and equation (5.11) together provide a necessary and sufficient condition for solving the problem (5.10). According to this condition, the loads should be distributed first through the fastest link, then through the second fastest link, and so on until the slowest link is assigned the last load fraction and all the nodes should stop processing loads at the same time. Thus, by adopting such a sequence of load distribution according to the speeds of the links and finding a solution to make all the nodes stop processing at the same time, an optimal processing time will be achieved.

Now we consider Example 5.1 to prove the optimal sequence. We use brute-force method to find a sequence following which the processing time is minimum. Because node 1 has three neighboring nodes, the number of possible sequences that should be searched is $3! = 6$. For each sequence, we use the search method (5.12) and (5.13) to find the solution that all the nodes will stop processing loads together. The search results are shown in Table 5.1, in which we can observe that the minimum processing time happened in number 4, the sequence of which is $S_1(4) = 1, S_1(2) = 2$ and $S_1(6) = 3$. This sequence just follows the order in which the link speeds $\frac{1}{C_{14}}, \frac{1}{C_{12}}, \frac{1}{C_{16}}$ decrease, and Theorem 5.1 holds. Following the optimal sequence, we

Table 5.1: Brute-force search results for optimal sequence of Example 5.1

Number	sequence	Max processing time (Sec.)
1	$2 \longrightarrow 4 \longrightarrow 6$	97.86
2	$2 \longrightarrow 6 \longrightarrow 4$	115.60
3	$4 \longrightarrow 6 \longrightarrow 2$	89.23
4	$4 \longrightarrow 2 \longrightarrow 6$	87.93*
5	$6 \longrightarrow 2 \longrightarrow 4$	120.22
6	$6 \longrightarrow 4 \longrightarrow 2$	115.50

obtain the optimal processing time of this example, which is 87.93 seconds as indicated by a star in Table 5.1.

Thus, we have provided a *necessary and sufficient* condition for the optimal solution of the sub-problem (5.10) and a search method (5.12) and (5.13) which is a building block of the proposed scheduling strategy. In the following section, we will present our scheduling strategy for divisible jobs originating from single and multiple sites on arbitrary networks.

5.2.2 Scheduling strategy for an arbitrary topology

In this section, using the sub-algorithm for each node in the system described above, we shall derive our scheduling strategy systematically. Though the basic idea of the solutions for the scheduling problem of divisible jobs originating from single and multiple sites is the same, there are some difference between the two cases. Hence, we will present the strategy for loads originating from single site first and then discuss the main difference of the two cases. Lastly, we will propose the strategy for loads originating from multiple sites on arbitrary networks.

From Theorem 5.1, we can observe that for node i, the optimal sequence is independent of l_j, the amount of loads to be processed at node j, and E_j, the processing capability of node j,

where $j \in V_i$. For a given l_j, the time instance at which node j starts processing may be later than the time when node j receives the jobs coming from node i, if we increase the processing speed ($\frac{1}{E_j}$) to a higher value. Hence, we can draw a conclusion that the optimal sequence of node i is also independent of t_j^{max}, when node j receives all the jobs from its neighboring nodes. We obtain the following lemma:

Lemma 5.1. *For a node i in the system, in case $t_j^{max} \geq t_j^{max}(i),\ j \in V_i$, an optimal sequence for node i exists.*

Now, we shall make little modification of sub-problem (5.10) as following:

$$Minimize: \quad D_i(l_i, x_i) = max\{T_i,\ T_j,\ |\ \forall j \in V_i\}, \tag{5.14}$$
$$where, \quad T_i = t_i^{max} + E_i \times l_i, \quad T_j = t_j^{max} + E_j \times l_j,$$

with the same constraints in (5.5). According to Lemma 5.1, we can observe that the optimal sequence and optimal condition (5.11) still hold. Hence, (5.12) and (5.13) can also be used to seek an optimal solution for the sub-problem (5.14).

We consider Fig. 5.1 to discuss the scheduling strategy for divisible loads originating from single site. In this example, we assume $L_1 = 100$ and $L_2 = L_3 = \ldots = L_n = 0$. Each node i in the system will try to find an optimal solution for the single-level tree sub-graph, the root processor of which is node i itself. In each iteration, the nodes calculate one by one, according to their total amount jobs γ_i defined by (5.1), till all the nodes have finished calculating. In the first iteration, node 1 sends some loads to its neighboring nodes $2, 4, 6$. Then, the neighboring nodes of node 1 also send some loads to their own neighboring nodes except node 1, and so on. Eventually, after some iterations, all the nodes in the system will have some loads to process and these loads are originating from node 1. For some nodes in the system, they may receive some loads from two or more neighboring nodes (maybe simultaneously or orderly). However, Lemma 5.1 guarantees that the optimal sequence still holds in this case. Hence, each node can obtain a local optimal solution by (5.12) and (5.13), following its own optimal

sequence. Below we continue with this example to discuss the search method in detail.

Similar to Fig. 5.3, by adding a virtual node d and virtual links (i,d) and (j,d), we can obtain a set of routing paths of node i in Fig. 5.1. The optimal sequence of each node i, $\bar{S}_i$, can be pre-defined before the calculation. In r-th iteration, node i has its own total amount of loads $\gamma_i^{(r)} = L_i + \sum_{j\in V_i} x_{ji}^{(r)}$, $(L_i = 0,\ i \neq 1)$. In order to avoid job shuttles within the system, node i will not calculate load amount to be transferred on the link (i,j), if $x_{ji}^{(r)} > 0$. Node i calculates $x_i^{k(r)}$ on each routing path one by one and skips the path $(i \to j \to d)$ if $x_{ji}^{(r)} > 0$. Obviously, $t_{j-i}^{max(r)}$, $t_i^{max(r)}$, and $x_{ji}^{(r)}$, $x_{mn}^{(r)}$, for $m \neq i, n \neq i, (m,n) \in C$, are independent of node i in r-th iteration. Before node i starts calculating $l_i^{(r+1)}$ and $x_{ij}^{(r+1)}$, $(x_{ji}^{(r)} = 0)$, node i can obtain $t_i^{max(r)}$ and $t_{j-i}^{max(r)}$ by following equations:

$$\begin{aligned} t_i^{max(r)} &= max\ \{t_j^{max(r)} + \sum_{k=1}^{\bar{S}_j(i)} C_{jm} \times x_{jm}^{(r)}, \mid \forall j \in V_i,\ x_{ji}^{(r)} > 0,\ m = \bar{S}_j^{-1}(k)\}, \\ t_{j-i}^{max(r)} &= max\{t_n^{max(r)} + \sum_{k=1}^{\bar{S}_n(j)} C_{nm} \times x_{nm}^{(r)}, \mid \forall n \in V_j,\ n \neq i,\ x_{nj}^{(r)} > 0, \\ & m = \bar{S}_n^{-1}(k)\}. \end{aligned}$$

However, node i can determine $t_j^{max(r)}(i)$ as:

$$t_j^{max(r)}(i) = t_i^{max(r)} + \sum_{k=1}^{\bar{S}_i(j)} C_{im} \times x_{im}^{(r)},\ j \in V_i,\ m = \bar{S}_i^{-1}(k),$$

by the transfer vector $x_i^{(r)}$ in r-th iteration. Thus, we can obtain the time instance at which node j finishes receiving loads from its neighboring nodes, which is:

$$t_j^{max(r)} = max\ \{t_j^{max(r)}(i),\ t_{j-i}^{max(r)}\}. \tag{5.15}$$

From (5.15), we can obtain the time delay on each path in P_i of node i in r-th iteration:

$$\begin{aligned} d_i^{0(r)} &= t_i^{max(r)} + E_i \times x_i^{0(r)}, \\ d_i^{k(r)} &= t_j^{max(r)} + E_j \times l_j^{(r)},\ k = 1, 2, \ldots, v_i,\ j = \bar{S}_i^{-1}(k), \end{aligned} \tag{5.16}$$

where, $l_j^{(r)} = L_j + \sum_{m\in V_j} x_{mj}^{(r)} - \sum_{m\in V_j} x_{jm}^{(r)}$. We can also obtain the first derivative of the time

delay on each path as shown in the following:

$$\frac{\partial d_i^{0(r)}}{\partial x_i^{0(r)}} = E_i,$$

$$\frac{\partial d_i^{k(r)}}{\partial x_i^{k(r)}} = \begin{cases} E_j, & if\ t_j^{max(r)}(i) < t_{j-i}^{max(r)}, \\ E_j + C_{ij}, & if\ t_j^{max(r)}(i) \geq t_{j-i}^{max(r)}, \end{cases} \tag{5.17}$$

where, $k = 1, 2, \ldots, v_i,\ j = \bar{S}_i^{-1}(k)$.

Thus we have obtained all the variants needed to carry out next calculation by using (5.12) and (5.13). As we mentioned before, the step-size $\alpha^{(r)}$ in (5.12) is some positive scalar which may be chosen by a variety of methods available in the literature [62]. We choose $\alpha^{(r)}$ to be a constant from the range [0.5, 1] for all α, which is shown to work well in our method. The distributed working style of the proposed scheduling strategy is as follows.

Any one of the processors in the system works as a *core* processor whose role is to trigger computations among the set of processors. At initial time, the core processor starts to compute an optimal solution for its sub-graph and also informs the rest of the nodes to compute one by one in a pre-defined sequence. The objective of the core processor is to avoid the time asynchrony among the processors in the system. When node i finishes computing and obtains a solution it broadcasts the solution to its neighboring nodes. After some iterations, when the stopping rule is satisfied, all the nodes stop computing and obtain an optimal solution of the system, if it exists. It may be noted that if the core processor fails in some situation, any other node can assume the role of a core processor.

Though our strategy has the distributed characteristic, it can be also implemented in a centralized method. If a processor has all the information about the topology of the system and about the processors and links, such as $E_i,\ C_{ij}$, etc., it can simulate the working style of each node and search for an optimal solution of the system. Once the key node obtains the optimal solution, it can informs the rest nodes to transfer the loads according to the optimal scheduling. Thus, our scheduling strategy is flexible for both distributed situation, when a node in the system has knowledge only about its local topology, and centralized situation,

when one key node has the information about the entire topology and other information about the system.

With the distributed working style, we present an important theorem, which essentially highlights some important properties of our scheduling strategy.

Theorem 5.2 (Cycle Free): *An optimal solution to problem (5.5) is cycle free, i.e., there exists no load transfer such that* $x_{i_1 i_2} > 0$, $x_{i_2 i_3} > 0, \ldots, x_{i_m i_1} > 0$, *where* $i_1, i_2, \ldots, i_m \in N$.

Proof. We assume that in r-th iteration, i_1 has some loads to send to its neighboring nodes. Node i uses (5.12) and (5.13) to search an optimal solution for its sub-graph. If $x^{(r)}_{i_1 i_2} > 0$, the path $(i_1 \rightarrow i_2 \rightarrow d)$ must be the MDRP in P_{i_0} and $d^{0(r-1)}_{i_1} > d^{0(r-1)}_{i_2}$ holds that is, $T^{(r-1)}_{i_1} > T^{(r-1)}_{i_2}$. Similarly, if $x^{(r)}_{i_2 i_3} > 0$, we can also obtain $d^{0(r-1)}_{i_2} > d^{0(r-1)}_{i_3}$. Recursively, we observe that the following inequalities hold:

$$d^{0(r-1)}_{i_1} > d^{0(r-1)}_{i_2} > \ldots > d^{0(r-1)}_{i_m}.$$

For node i_m, if it should send some loads to node i_1, the inequality $d^{0(r-1)}_{i_m} > d^{0(r-1)}_{i_1}$ must hold, which is a clear contradiction. Hence, in each iteration, it is impossible that a load cycle $x^{(r)}_{i_1 i_2} > 0$, $x^{(r)}_{i_2 i_3} > 0, \cdots, x^{(r)}_{i_m i_1} > 0$ exists. Our optimal solution is approached by the iterations and it can be deduced that Theorem 5.2 holds. **Q.E.D**

Though each node i can calculate the local optimal solution according to $\gamma^{(r)}_i$ in r-th iteration, where $\gamma^{(r)}_i = L_i + \sum_{j \in V_i} x^{(r)}_{ji}$ and node i also can handle the situation when $L_i > 0$, up to now we only assume that one of the nodes in the system has some loads to be scheduled initially. Because of the assumption that each node can start processing loads only after it receives all loads coming from its neighboring nodes, we may meet an interesting problem when we use the search method for the divisible loads originating from two or more sites.

In some cases, some node may not receive or send loads to its neighboring nodes at all and the optimal solution will never be achieved. For example, we only consider two nodes in a system, i and j, which are connected by links (i, j) and (j, i), and $L_i > 0$, $L_j > 0$. Node

i may have other neighboring nodes and node j has only one neighboring node. Initially we assume $E_i \times L_i > E_j \times L_j$, and for the optimal solution, some loads shall be transferred from node i to j. However node j cannot start processing jobs until it receives all loads from node i. For node i, it has many routing paths including $(i \to d)$ and $(i \to j \to d)$ and the time delays of the two paths should be $d_i^0 = E_i \times l_i$ and $d_i^m = t_j^{max} + E_j \times (L_j + x_{ij})$, where $t_j^{max} = \sum_{k=1}^{m}(C_{in} \times x_{in}), n = S_i^{(-1)}(k), m = S_i(j)$. Due to t_j^{max}, the time instance at which the last job arrives at node j, and L_j, d_i^m may be greater than d_i^0. Hence node i will not send loads to node j. For node j, needless to mention that it will not send loads to node i, because of $d_j^0 < d_j^1$, where path 1 is $(j \to i \to d)$.

We refer to this problem as "Trap" problem. In order to solve "Trap" problem, we shall make little modification of the assumption that each node cannot start processing until it receives all loads from its neighboring nodes. We assume that, for node i, if $L_i > 0$, it can start processing its loads at $t = 0$. When the first load arrives at node i, it will stop processing and start to receive loads. Node i can resume the processing after the last job arrives at node i. According to this modification, we must determine the time instance at which the first job arrives at each node. After the last job is received by each node, we obtain the following:

$$\begin{aligned} t_i^{min} &= min\ \{t_j^{max} + \sum_{k=1}^{\bar{S}_j(i)-1} C_{jm} \times x_{jm}, \mid \forall j \in V_i,\ x_{ji} > 0,\ m = \bar{S}_j^{-1}(k)\}, \qquad (5.18) \\ t_{i-j}^{min} &= min\{t_m^{max} + \sum_{k=1}^{\bar{S}_m(i)-1} C_{mn} \times x_{mn}, \mid \forall m \in V_i, m \neq j, x_{mi} > 0, \\ & n = \bar{S}_m^{-1}(k)\}. \end{aligned}$$

When node i resumes processing, it has some unprocessed loads, which is given by:

$$L_i^* = \ max\{0,\ L_i - \frac{t_i^{min}}{E_i}\}.$$

Then, in each iteration, the total amount of loads $\gamma_i^{(r)}$ should be changed to:

$$\gamma_i^{(r)} = L_i^{*(r)} + \sum_{j \in V_i} x_{ji}^{(r)},\ L_i^{*(r)} = max\{0,\ L_i - \frac{t_i^{min(r)}}{E_i}\}. \qquad (5.19)$$

Table 5.2: Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 1)

Step 1: ***Initialization***

$r = 0$ (r: iteration index)

For $i = 1$ to n, $n = |N|$

1. Determine P_i, according to the optimal sequency S_i.
2. Calculate the total amount loads at node i, which is $\gamma_i^{(0)} = L_i + \sum_{j \in V_i} x_{ji}^{(0)}$.
3. Find a feasible set $x_i^{(0)}$ as an initial feasible solution, which satisfies

   $\sum_{k=0}^{v_i} x_i^{k(0)} = \gamma_i^{(0)}$. Here, we choose $x_i^{(0)} = \gamma_i^{(0)}$, $x_i^{k(0)} = 0,\ k = 1, \ldots, v_i$.

End For

After this modification, each node i can calculate the time delay of all the routing paths according to $L^{*(r)}$ and the "Trap" problem has been solved. Thus, we have proposed our scheduling strategy for divisible loads originating from single and multiple site. A pseudo-code of our proposed algorithm is also shown in Table 5.2, Table 5.3, Table 5.4, which are three steps of this algorithm, respectively.

Note that, for each node searching for a local solution, there is little difference between the sub-algorithm proposed in Chapter 3.1 and the search method used in our strategy. In [27,62], it is pointed out that to obtain a faster convergence rate, it is better to reach a solution between the original starting point and the local optimal point in each iteration instead of reaching the local optimal point directly. Hence, in our strategy, in each system iteration r, every node calculates the amount loads on every feasible routing path only once, instead of several times as shown in Chapter 5.2.1.

Table 5.3: Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 2)

Step 2: ***Solution Procedure***

Let $r = r + 1$

For $i = 1$ to $n,\ n = |N|$, do the following for node i.

1. If $L_i > 0$, calculate $t_i^{min(r)}$ by (5.18) and $\gamma_i^{(r)}$ by (5.19).

   Else, $\gamma_i^{(r)} = \sum_{j \in V_i} x_{ji}^{(r)}$.

2. Determine $t_i^{max(r)}$ according to the incoming loads $x_{ji}^{(r-1)}$, and $d_i^{0(r)}$.

3. For $k = 1$ to v_i, do the following for each neighboring node $j = S_i^{-1}(k)$.

   If $x_{ji}^{(r-1)} > 0$, do the next loop.

   Calculate $t_j^{max(r)}$, $t_{j-i}^{max(r)}$, and $t_j^{min(r)}$, $t_{j-i}^{min(r)}$ if $L_j > 0$.

   Calculate d_i^k and $\frac{\partial d_i^k}{\partial x_{ij}^{(r-1)}}$, by (5.16) and (5.17), respectively.

   Find MDRP $d_i^{min} = \ min\{d_i^k\}$.

   End For.

4. For $k = 0$ to v_i, do the following for each routing path.

   Let $x_i^{k(r)} = max\{0, x_i^{k(r-1)} - \alpha^{(r)}(d_i^k - d_i^{min})/H_i^k\},\ k \neq min$, and

   $x_i^{min(r)} = \gamma_i^{(r)} - \sum_{k=0, k \neq min}^{v_i}(x_i^{k(r)})$.

   End For.

End For.

Table 5.4: Proposed Scheduling Strategy for Loads Originating from Multiple Sites (Step 3)

Step 3: ***Stopping Rule***
If $\|T^{max(r)} - T^{min(r)})\| < \varepsilon$, then **stop**, where, $T^{max(r)} = max\{T_i^{(r)}, \mid \forall i \in N\}$, $T^{min(r)} = min\{T_i^{(r)}, \mid \forall i \in N\}$, and ε is a desired acceptable tolerance for solution quality; otherwise, go to Step 2.

5.2.3 Convergence and complexity of the algorithm

In this section, we prove the convergence of the proposed algorithm and present the derivation of its complexity. Note that the algorithm executes the sub-algorithm iteratively for node i, $i = 1, 2, \ldots, n$. That is, the sub-algorithm is a building block of our proposed algorithm. Let the sub-algorithm for node i denote as an *operator* $\mathcal{Z}_i : FS \mapsto FS$ where FS is the set of all feasible solutions. In turn, denote the proposed algorithm by an operator $\mathcal{Z}$, which is a composition of operators $\mathcal{Z}_i$, $i = 1, 2, \ldots, n$.

$$\mathcal{Z} = \mathcal{Z}_1 \circ \mathcal{Z}_2 \circ \ldots \circ \mathcal{Z}_n.$$

As we discussed in Chapter 3.2, for the sub-problem (5.14), we obtain the following lemma:

Lemma 5.2. *The operator $\mathcal{Z}_i$ generates an optimal solution to sub-problem (5.14), $i = 1, 2, \ldots, n$.*

Dafermos and Sparrow [88] have derived some elegant results showing some general conditions for the convergence of an algorithm. Here, we apply their results to prove the convergence of the proposed algorithm. Note that l is determined if x is given. Hence we use $D(x)$ instead of $D(l, x)$ for the sake of convenience. From Theorem 2.1 and Theorem 2.2 in [88], we have the following lemma:

Lemma 5.3. *The set of feasible solutions to problem (5.5) FS is closed and convex. Further, if the operator $\mathcal{Z}$ has the following properties, then the proposed algorithm gives an optimal*

solution to problem (5.5).

1) $\mathcal{Z}x = x$ *for some* $x \in FS$ *implies that* x *satisfies the condition that all the nodes will stop processing at the same time.*

2) $\mathcal{Z}$ *is a continuous mapping from* FS *to* FS.

3) $D(x) \geq D(\mathcal{Z}x)$ *for all* $x \in FS$.

4) $D(x) = D(\mathcal{Z}x)$ *for some* $x \in FS$ *implies that* $\mathcal{Z}x = x$.

The fact that the set of feasible solutions space is closed and convex is evident from the linearity of the problem and can be argued as follows. Our sub-algorithm, for node i constructs a single-level tree. In each iteration r, x_{mj}, $m \neq i$ and L_j, $j \in V_i$ is independent of node i and we can consider them as constants. Hence the set of feasible solutions to sub-problem (5.14) is also closed and convex. Because $\mathcal{Z} = \mathcal{Z}_1 \circ \mathcal{Z}_2 \circ \ldots \circ \mathcal{Z}_n$, if the solutions to each $\mathcal{Z}_i$ is closed and convex, we can realize that the set of feasible solutions to problem (5.5) is also closed and convex.

Theorem 5.3. *The proposed algorithm* $\mathcal{Z}$ *generates a sequence* $\{x^{(r)}\}$ *which converges to an optimal solution to the optimization problem defined by (5.5).*

Proof. The proof of the theorem can be completed if the proposed algorithm $\mathcal{Z}$ satisfies the properties required in Lemma 5.3. Above, we proved that the feasible solutions to problem (5.5) is closed and convex. Now, we prove that $\mathcal{Z}$ has the four properties given in Lemma 5.3.

1). $\mathcal{Z}x = x$ implies $\mathcal{Z}_1 x_1 \circ \mathcal{Z}_2 x_2 \circ \cdots \circ \mathcal{Z}_n x_n = \{x_1, x_2, \ldots, x_n\}$. We can deduce that $\mathcal{Z}_i x_i = x_i$ for all $i \in N$, which implies that $T_1 = T_2 = \ldots = T_n$. Hence, Property 1 in Lemma 5.3 holds.

2). In [15], the authors have given the solutions for single-level tree from which it is easy to see that $\mathcal{Z}_i$ is a continuous mapping from FS to FS. Then, from the definition of $\mathcal{Z}$ we can obtain that $\mathcal{Z}$ is also a continuous mapping from FS to FS.

3). According to Lemma 2, we have $D(x_i) \geq D(\mathcal{Z}_i x_i)$. Notice that $\mathcal{Z}$ is a composition of operators $\mathcal{Z}_i$. Thus, we have $D(x) \geq D(\mathcal{Z}_1 x_1 \circ \mathcal{Z}_2 x_2 \circ \cdots \circ \mathcal{Z}_n x_n) \Rightarrow D(x) \geq D(\mathcal{Z}x)$. That is, Property 3 in Lemma 5.3 holds.

4). If $D(x) = D(\mathcal{Z}x)$, we obtain $T_1 = T_2 = \ldots = T_n$. For each node $i \in N$, if $T_i = T_j,\ j \in V_i$, we can obtain $\mathcal{Z}_i x_i = x_i$. Then, we have $\mathcal{Z}x = x$.

Hence, the theorem holds. **Q.E.D**

We shall now quantify the complexity of the algorithm proposed in this chapter. As a first step of our algorithm, we sort the neighboring nodes of node i in the system according to optimal sequence. We use Quicksort algorithm [89] which takes us $O(v_i \lg v_i)$ steps for node i, where v_i is the number of node i's neighboring nodes ($v_i = |V_i|$). Thus, it consumes $O(max\{v_i \lg v_i,\ |\ \forall i \in N\} \times n)$ steps for the system to obtain the optimal sequence for each node. Considering $v_i \leq (n-1),\ n = |N|$, we can simplify this complexity to $O(n^2 \lg n)$. Note that, once the optimal sequence is determined, all the calculations can follow this sequence. Further, in each iteration r, in our calculations, we shall compute the amount loads on each possible routing path of node i which takes us $O(v_i + 1)$ steps. The complexity to calculate in one iteration should be $O((v_1+1)+\ldots+(v_n+1)) \approx O(n^2)$. If the number of iterations is R, the complexity of the computation can be $O(Rn^2)$ steps, because our algorithm can converge to an optimal solution after some iterations R. We can observe that R is important to determine the complexity of our algorithm. In the following, we shall estimate the complexity of R.

To estimate the complexity of R, we consider a system with n nodes. Obviously, the worst case is the linear daisy chain network with the load originating from node 1, as shown in Fig. 5.5(a). Before we start our algorithm, we can observe that $T_1^{(0)} = L_1 \cdot E_1$ and $T_2^{(0)} = T_3^{(0)} = \ldots = T_n^{(0)} = 0$, as shown in Fig. 5.5(b). In each iteration of our algorithm, we let node 1 start computation first. When node 1 finishes, node 2 starts, and so on till node n finishes its computation. Now, we denote $\triangle_i^{(r)} = T_{i-1}^{(r)} - T_i^{(r)}$ as the time difference between

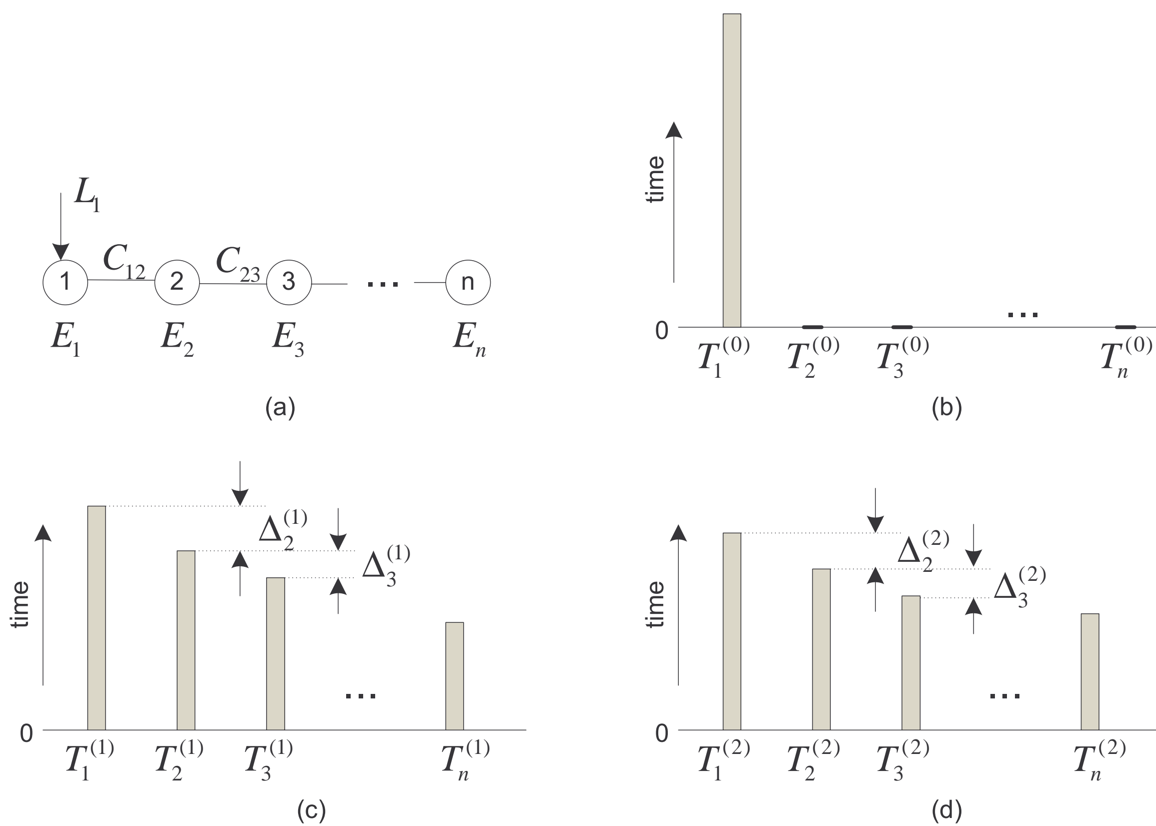

Figure 5.5: The iteration procedure of the proposed algorithm.

node $i-1$ and node i in iteration r, where $1 < i \leq n$ and $\triangle_i^{(r)} \geq 0$ always holds. When $r = 0$, we can obtain $\triangle_2^{(0)} = E_1 \cdot L_1$ and $\triangle_i^{(r)} = 0,\ 2 < i \leq n$. After first iteration, we can obtain the following equations:

$$\begin{aligned}
\triangle_2^{(1)} &= \frac{E_1}{C_{12}+E_1+E_2} \cdot \frac{E_2}{C_{23}+E_2+E_3} \cdot E_2 \cdot L_1 \\
&= \frac{E_2}{C_{12}+E_1+E_2} \cdot \frac{E_2}{C_{23}+E_2+E_3} \cdot \triangle_2^{(0)}; \\
\triangle_3^{(1)} &= \frac{E_1}{C_{12}+E_1+E_2} \cdot \frac{E_2}{C_{23}+E_2+E_3} \cdot \frac{E_3}{C_{34}+E_3+E_4} \cdot E_3 \cdot L_1 \\
&= \frac{E_3}{C_{34}+E_3+E_4} \cdot \frac{E_3}{E_2} \cdot \triangle_2^{(1)}; \\
&\cdots \\
\triangle_{n-1}^{(1)} &= \prod_{i=1}^{n-1} \frac{E_i}{C_{i(i+1)}+E_i+E_{i+1}} \cdot E_{n-1} \cdot L_1 \\
&= \prod_{i=3}^{n-1} \frac{E_i}{C_{i(i+1)}+E_i+E_{i+1}} \cdot \frac{E_{n-1}}{E_2} \cdot \triangle_2^{(1)}; \\
\triangle_n^{(1)} &= 0.
\end{aligned}$$

We define $\triangle^r = T_1^{(r)} - T_n^{(r)}$ as the $T^{max(r)} - T^{min(r)}$, which can be deduced as:

$$\triangle^{(r)} = \sum_{i=2}^{n-1} \triangle_i^{(r)} = [1 + \sum_{i=3}^{n-1}(\prod_{j=3}^{i} \frac{E_j}{C_{j(j+1)}+E_j+E_{j+1}} \cdot \frac{E_i}{E_2})] \cdot \triangle_2^{(r)} = \phi \cdot \triangle_2^{(r)}, \tag{5.20}$$

where ϕ is a fixed number according to the network settings. In iteration r, we can deduce that $\triangle_2^{(r)}$ is:

$$\begin{aligned}
\triangle_2^{(r)} &= (\frac{E_2}{C_{12}+E_1+E_2} \cdot \triangle_2^{(r-1)} + \triangle_3^{(r-1)}) \cdot \frac{E_2}{C_{23}+E_2+E_3} \\
&= (\frac{E_2}{C_{12}+E_1+E_2} \cdot \frac{E_2}{C_{23}+E_2+E_3} + \frac{E_3}{C_{23}+E_2+E_3} \cdot \frac{E_3}{C_{34}+E_3+E_4}) \cdot \triangle_2^{(r-1)} \\
&= \eta \cdot \triangle_2^{(r-1)}.
\end{aligned} \tag{5.21}$$

Obviously, $0 < \eta < 1$ holds. By (5.20) and (5.21), we obtain the following equation:

$$\begin{aligned}
\triangle^{(r)} = \phi \cdot \triangle_2^{(r)} &< \lim_{n\to\infty}[1 + \sum_{i=3}^{n-1}(\prod_{j=3}^{i} \frac{E_j}{C_{j(j+1)}+E_j+E_{j+1}} \cdot \frac{E_i}{E_2})] \cdot \triangle_2^{(r)} = \Phi \cdot \triangle_2^{(r)} \\
\Rightarrow \triangle^{(r)} &< \Phi \cdot \triangle_2^{(r)} \leq \Phi \cdot \eta^{r-1} \cdot E_1 \cdot L_1.
\end{aligned} \tag{5.22}$$

From (5.22), we can see R is independent of n. Hence, if we determine the value of ε, we can estimate the iteration number R by:

$$R > \frac{\ln\varepsilon - \ln\Phi - \ln E_1 - lnL_1}{\ln\eta} + 1. \tag{5.23}$$

Hence, the total complexity of our algorithm is $O(n^2 \lg n) + O(Rn^2) \approx O(n^2 \lg n)$.

5.3 Simulation Results and Some Discussions

Our proposed algorithm retains three important features of practical interest: Firstly, the algorithm is simple to realize and can be implemented in a distributed fashion. The second one is in its style of working by avoiding the need for generating a timing diagram explicitly for any complex networks having an arbitrary network topology. The last one is its capability of handling divisible loads originating from both single and multiple sites. To the best of our knowledge, the studies attempted in [14] are closest to our work. In [14], the authors proposed a centralized algorithm named "*Resource-Aware Optimal Load Distribution Algorithm*" (RAOLD) to solve the scheduling problem of divisible load originating from single site on arbitrary graphs. It will be interesting to compare the performance of RAOLD and our algorithm and discuss the main difference of the two algorithms in the case that divisible loads originating from single site on arbitrary networks. It is the first time in the literature that an algorithm is proposed to solve the scheduling problem of divisible loads originating from multiple sites on arbitrary networks. We shall study the performance of the proposed algorithm for several situations by means of extensive simulations. We will highlight and discuss all the important issues observed from our simulation results.

In the rest of this section, we will show an illustrative example which demonstrates the details of the workings of our proposed algorithm and then compare the performance of RAOLD algorithm with ours. Further, we will consider divisible loads originating from multiple sites on several types of networks such as *sparse, medium-dense*, and *dense* so as to capture the

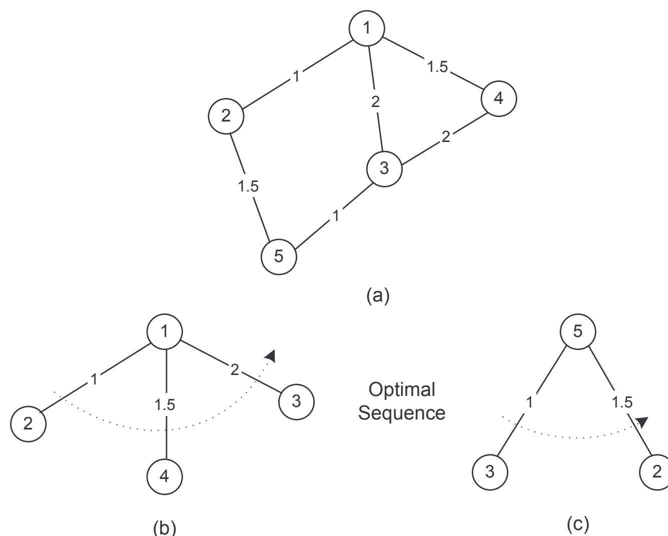

Figure 5.6: Experiment 5.1: (a) An arbitrary network; (b) Optimal sequence of node 1; (c) Optimal sequence of node 5.

performance of our algorithm.

5.3.1 Demonstration: An example of optimal scheduling

Following example clarifies and illustrates the workings of our proposed algorithm in seeking optimal solution, if it exists.

Experiment 5.1: Consider a distributed computer system that consists of five host computers (nodes), as illustrated in Fig. 5.6(a). There are $L = 100$ units divisible loads originating from node 1. The processor speed parameters are randomly generated and they are $E_1 = 2,\ E_2 = 2.5,\ E_3 = 2,\ E_4 = 1.5,\ E_5 = 1$ seconds for unit load and the respective link speed parameters are also randomly chosen and are marked in the graph. In this experiment, node 1 will try to send some loads to the rest nodes to obtain a minimum processing time.

As a first step of our proposed algorithm, we determine an optimal sequence $\bar{S}_i$ defined in Theorem 1 for node i, $i = 1, 2, \ldots, 5$. For example, the optimal sequence of node 1 and node 5 are shown in Fig. 5.6(b) and Fig. 5.6(c), respectively. Then, we determine the

routing path of each node i according to the optimal sequence $\bar{S}_i$ one by one with the virtual node d and the virtual links (i, d) and (j, d), $j \in V_i$. Each node i obtains a set of routing paths $P_i = [p_i^0,\ p_i^1, \ldots, p_i^{v_i}]$, $v_i = |V_i|$. For example, node 2 has three routing paths, which are $p_2^0 = (2 \rightarrow d)$, $p_2^1 = (2 \rightarrow 1 \rightarrow d)$ and $p_2^2 = (2 \rightarrow 5 \rightarrow d)$. From system setting, we can obtain that $\gamma_1^{(0)} = L = 100$ and $\gamma_2^{(0)} = \gamma_3^{(0)} = \gamma_4^{(0)} = \gamma_5^{(0)} = 0$. After this, we set a feasible solution in the first iteration, which is $x_1^{(0)} = [x_1^{0(0)}, \ldots, x_1^{3(0)}] = [100, 0, 0, 0]$, $x_2^{(0)} = [x_2^{0(0)}, \ldots, x_2^{2(0)}] = [0, 0, 0]$, and so on. This completes the *Initialization* step in our proposed algorithm.

Then, in the *Solution Procedure* (second) step, each node i will calculate the amount of loads on each path in P_i according to the equations discussed in Chapter 3.1 and 3.2. Note that in each iteration r, node i will skip the calculation on path $(i \rightarrow j \rightarrow d)$, if $x_{ji}^{(r)} > 0$, $j \in V_i$. When the r-th iteration is completed, the algorithm will determine $T^{max(r)}$ and $T^{min(r)}$ (the maximum processing time and the minimum processing time among all the processors, respectively). If the difference between $T^{max(r)}$ and $T^{min(r)}$ is within an acceptable tolerance ε, we stop the *Solution Procedure* and obtain a near-optimal solution [1] . Otherwise, continue the procedure. In all experiments in this chapter, we set $\varepsilon = 0.01$ seconds.

In Fig. 5.7, we show the maximum processing time and the difference between the maximum and minimum processing time of each iteration r for Experiment 5.1. From this figure, we can observe that in this experiment the maximum processing time converges to 83.41 seconds, and the difference between the maximum and minimum processing time converges to 0. After 15 iterations, we obtain an optimal solution for this system, which is $\bar{l}_1 = 41.70$, $\bar{l}_2 = 18.53$, $\bar{l}_3 = 5.81$, $\bar{l}_4 = 15.42$, $\bar{l}_5 = 18.53$. The timing diagram of the solution is also shown in Fig. 5.8, from which we can observe that at $t = 60.28$, node 3 receives

[1] Note that the optimal solution is obtained when all nodes stop computing at the same time instant, as shown in the literature [15] and in (5.11) above. However, in our iterative procedure, it is natural that we strike a near-optimal solution due to the stopping rule with $\varepsilon > 0$.

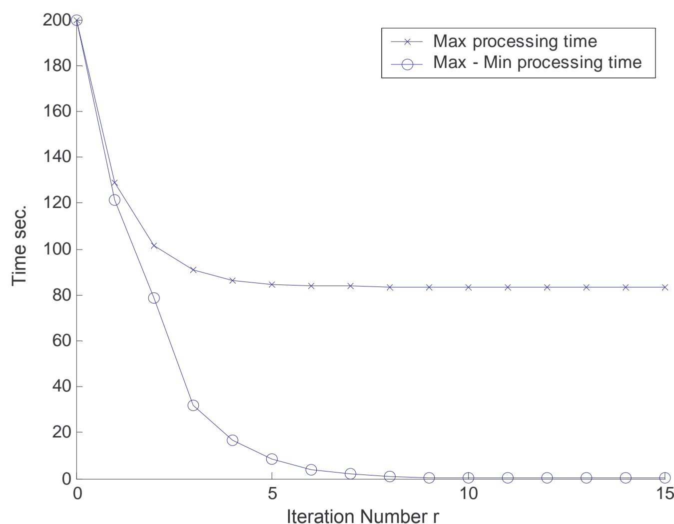

Figure 5.7: Results of Experiment 5.1: $T^{max(r)}$ and $T^{max(r)} - T^{min(r)}$ in each iteration r.

loads from node 1 and node 4 simultaneously and the procedure lasts 0.12 seconds. By using the proposed algorithm, it is shown that the optimal processing time of the system is 83.41 seconds, which is much shorter than the processing time 200 seconds in the case of no loads scheduling, where only node 1 processes its own loads.

5.3.2 Performance comparison of algorithms

Notice that if divisible loads originating from single site, the RAOLD algorithm proposed in [14] can also be applied directly to solve the scheduling problem. We shall first give a brief introduction of RAOLD algorithm and then compare the proposed algorithm with RAOLD algorithm in terms of processing time of the system.

Algorithm **RAOLD** [14]: In this work, the authors transformed the scheduling problem into a multi-level unbalanced tree network. For a given arbitrary network, RAOLD algorithm first generates a *minimum cost spanning tree* (MST) by using Kruskal's or Prim's algorithms [27,89]. The root node of the MST is the node at which a divisible load is assumed to originate. Then, RAOLD algorithm applies **Rule A** in [15] systematically in a *bottom-up* fashion to the MST, starting from the last level to the root, to obtain an optimal reduced tree,

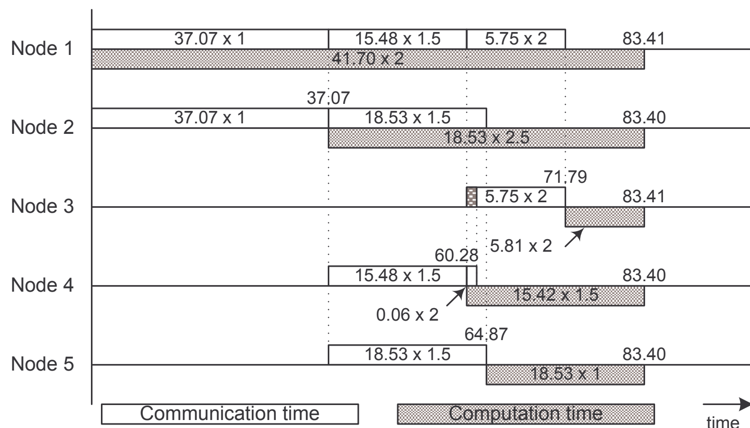

Figure 5.8: Results of Experiment 5.1: Timing diagram.

which is a single-level tree. Following the optimal sequence defined in Theorem 1, RAOLD algorithm can deliver an optimal time performance. Interested readers can refer to [14] for more details. Now we conduct the following experiment to compare the performance of the proposed algorithm with RAOLD algorithm.

Experiment 5.2: Consider an distributed computer system that consists of ten host processors (nodes), as illustrated in Fig. 5.9. In this experiment, the processor speed parameters are $E_1 = 4.93$, $E_2 = 0.77$, $E_3 = 0.55$, $E_4 = 0.03$, $E_5 = 9.22$, $E_6 = 7.53$, $E_7 = 4.68$, $E_8 = 9.01$, $E_9 = 9.05$ and $E_{10} = 9.90$ seconds for unit load. The respective link speed parameters are marked in this graph. The processor speed and the link speed parameters are chosen using a uniform probability distribution in the range $[0.01,\ 10]$. We assume $L = 100$. Now, as per our proposed algorithm, we construct ten sub-experiments, in which the divisible loads are assumed to originate at node 1, 2, ..., 10, respectively. The experimental results are shown in Table 5.5.

From Table 5.5, we can observe that our proposed algorithm performs better than RAOLD algorithm in terms of the processing time of the system. When the divisible load originating from node 10, our proposed algorithm has the most gain over RAOLD algorithm, which is 19.9%. In Fig. 5.10, we present the timing diagram of the proposed algorithm when the

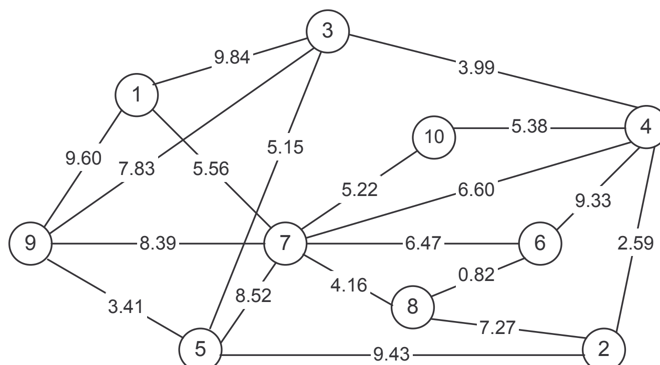

Figure 5.9: Experiment 5.2: An arbitrary network with 10 nodes.

Table 5.5: Computational results for load distribution with different load origination

Root node (i)	RAOLD (sec.)	Proposed Algorithm (sec.)	Gain
1	3.048×10^2	2.797×10^2	8.2%
2	5.951×10^1	5.951×10^1	0.0%
3	4.454×10^1	4.451×10^1	0.1%
4	2.969×10^0	2.968×10^0	0.0%
5	3.139×10^2	2.966×10^2	5.5%
6	1.671×10^2	1.514×10^2	9.4%
7	2.340×10^2	2.319×10^2	0.9%
8	1.234×10^2	1.214×10^2	1.6%
9	3.913×10^2	3.253×10^2	16.9%
10	4.300×10^2	3.443×10^2	19.9%

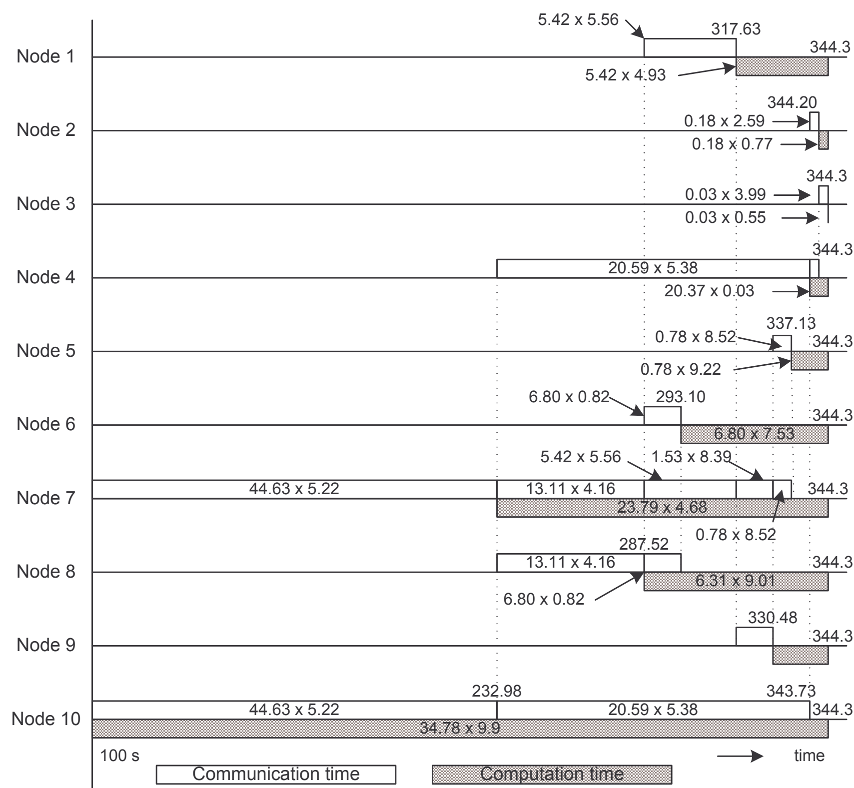

Figure 5.10: Results of Experiment 5.2: Timing diagram of the proposed algorithm when divisible loads originating from node 10.

divisible loads originating from node 10.

It would be of natural interest to investigate the mechanics behind the gain achieved by our algorithm over RAOLD. The reason is as follows. RAOLD algorithm is based on the minimum cost spanning tree generated from the original arbitrary topology. MST is only determined by the cost of links in the network. However, the number of hops that the loads shall be retransferred also significantly influences the processing time of the system. In some cases, if we choose the minimum hops instead of minimum cost of links, we can improve the system performance significantly. Now we consider an example to illustrate this fact.

Example 5.2: A simple system with three nodes is shown in Fig. 5.11(a), with $L_1 = 100$, $L_2 = L_3 = 0$ and the processor speed and link speed parameters are also marked in this graph.

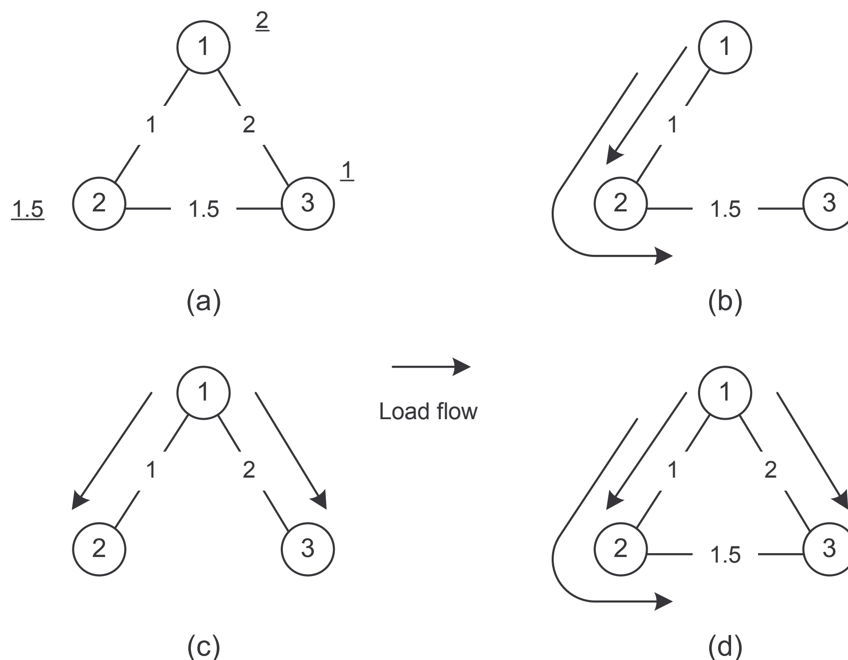

Figure 5.11: Example 5.2, load flow: (a) A simple example with three nodes; (b) MST with root node 1; (c) The single-level tree; (d) Our proposed algorithm.

To solve this problem, RAOLD algorithm first generates MST as shown in Fig. 5.11(b) and obtains an optimal solution which is $\bar{l}_1 = 49.21$, $\bar{l}_2 = 31.75$, $\bar{l}_3 = 19.05$ and the processing time is 98.43 seconds. For RAOLD algorithm, the loads sent to node 3 must go via node 2 and take two hops. However, if we construct a single-level tree as shown in Fig. 5.11(c), following an optimal sequence, we can obtain an optimal solution which is $\bar{l}_1 = 45.45$, $\bar{l}_2 = 36.37$, $\bar{l}_3 = 18.18$ and the processing time is 90.91 seconds. Although the Fig. 5.11(c) is not the MST of this system, node 1 can send loads to node 3 directly and takes only one hop and its processing time is much shorter than that of RAOLD algorithm. Our proposed algorithm obtains the shortest processing time which is 90.90 seconds. For the proposed algorithm, loads can be transferred on all the three links in the system, as shown in 5.11(d). From this example, we can observe that the processing time of the system is determined by both the cost of links and the number of hops. Our algorithm is shown to take advantage of both the cost of links and the number of hops, and hence performs much better than RAOLD which only considers

the cost of links.

5.3.3 Divisible loads originating from multiple sites

In this section, we shall conduct some experiments on divisible loads originating from multiple sites on arbitrary networks. Without loss of generality, we set the number of nodes in the system to nine. By varying the link probability as 25%, 50% and 80%, we generate sparse, medium-dense, and dense networks, respectively [14]. That is, the connectivity is varied from 25% to 80% to capture different types of graphs, which are shown in Fig. 5.12.

Experiment 5.3: Consider an arbitrary network system with nine nodes. The process speed parameters of the nodes are randomly generated following a uniform probability distribution as, $E_1 = 2.0$, $E_2 = 3.0$, $E_3 = 1.5$, $E_4 = 1.2$, $E_5 = 1.5$, $E_6 = 1.0$, $E_7 = 2.0$, $E_8 = 1.0$, and $E_9 = 1.0$. We assume that the divisible loads originate at node 3, 6 and 9 and we set $L_3 = 40$, $L_6 = 60$ and $L_9 = 100$ unit load throughout this experiment. We randomly generate a sparse network, a medium dense network and a dense network, as shown in Fig. 5.12(a)-(c), respectively. In each network, the link speed parameters are randomly chosen according to a uniform distribution in the range $[0.5, 10]$ and the values are marked in the graph. We conduct three sub-experiments to test the performance of our proposed algorithm for these networks and we show the results in Table 5.6, which presents the optimal solution $\bar{l}_i$ in unit load for each node i.

From Table 5.6, we observe that the system processing time decreases by increasing the degree of connectivity. However, as in [14], we cannot guarantee the performance improvement when degree of connectivity increases. When the degree of connectivity increases, in some cases, due to the changes of link delays, the optimal sequence of each node may be changed and thus, the optimal solution of the processing time may become even greater. In Fig. 5.13, we show that the timing diagram of the system when the network connectivity is medium dense. Through these experiments, we can draw a conclusion that our proposed algorithm

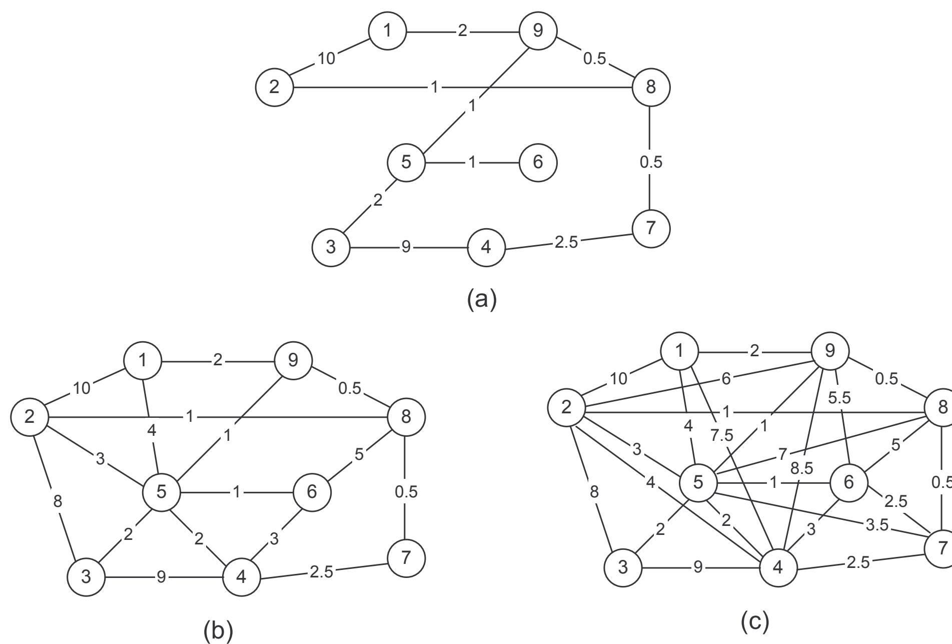

Figure 5.12: Experiment 3, a 9 nodes system: (a) A sparse connection; (b) A medium dense connection; (c) A dense connection.

Table 5.6: Computational results of Experiment 5.3 (unit load)

Node i	Sparse Network	Medium dense Network	Dense Network
1	6.42	4.84	5.30
2	7.69	5.27	1.18
3	33.34	32.09	32.07
4	3.65	5.08	8.55
5	17.12	12.92	14.2
6	50.01	48.13	48.11
7	0.97	13.66	12.9
8	30.78	29.87	29.56
9	50.01	48.14	48.11
Processing time	50.01 sec.	48.14 sec.	48.11 sec.

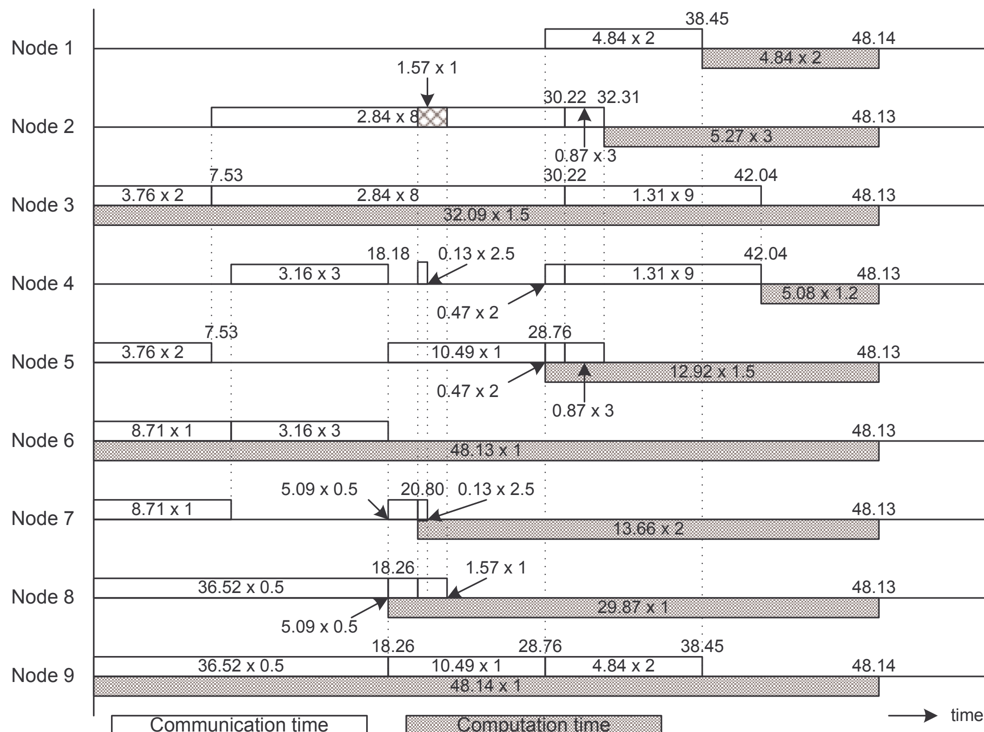

Figure 5.13: The timing diagram of the 9-node system when the connection is medium dense.

can also obtain the optimal solutions efficiently in the cases when divisible loads originate from multiple sites.

Note that, in some situations, our proposed algorithm may sink into some local optimal solutions (local minima) and the global optimal solution may be not achieved. In order to avoid this problem, we can adjust the value of $\alpha^{(r)}$ (the step size). As we mentioned before, we set $\alpha^{(r)}$ in the range of $[0.5,\ 1]$, which works well in our experiments.

5.4 Concluding Remarks

In this chapter, we have studied an optimal scheduling problem of divisible loads originating from single or multiple sites in a distributed computer system with arbitrary network configurations. We have presented a generic mathematical model for this problem and formulated it as a real-valued constrained optimization problem. We decomposed the problem

into sub-problems which could be solved by each processor in the system. We have derived the *necessary and sufficient* conditions for the optimal solution of the sub-problems and we have proven that after some iterations, our solution can converge to a global optimal solution of the system. On the basis of this study, we have proposed a *distributed* algorithm to solve the general divisible loads scheduling problem. We have adopted a load balancing approach using the concept of virtual routing in the design of our distributed algorithm. We analyzed the difference between the cases that divisible loads originate from single and multiple sites in our proposed algorithm and proposed a simple model to solve the "Trap" problem in the case of multiple sites. In case the load originates only from a single site, we compared the proposed algorithm with recently proposed RAOLD algorithm and analyzed the main difference between the two algorithms. It is shown that the proposed algorithm performs much better than RAOLD algorithm in terms of processing time. When divisible loads originated from multiple sites, we constructed several networks with different degree of connectivity to test the performance of the proposed algorithm. From the extensive simulations, we have proven that our proposed algorithm can efficiently solve the scheduling problem of divisible loads originating from single and multiple sites on arbitrary networks.

The approach followed in this chapter is a novel contribution to the domain of DLT. We designed a distributed scheduling strategy to achieve the optimal processing time of all loads in the system. It is the first time that a distributed algorithm is attempted and proposed in the DLT literature. The main advantage of this approach comes from the fact that the need for generating a timing diagram to develop recursive equations is altogether circumvented. Thus, when one needs to schedule multiple divisible loads originating from several sites, it is rather impossible to capture the load distribution process by a single timing diagram. The main difficulty lies in this approach would be to explicitly schedule several timing components such as computation and communication from one site to other sites, identifying which processor-link pairs are redundant [15], etc. In this chapter, we have taken a radically different

approach to address this complex problem by carefully formulating as a generalized minimization problem, thus avoiding the need for a timing diagram representation. Extensions to this work may consider applying the algorithm on large scale networks such as Intel Paragon system and also using non-linear functions for delay components, taking more constraints such as the release time of the processors in the system.

Chapter 6

Conclusions and Future Work

The analysis and design of distributed indivisible load balancing and divisible load scheduling strategies for arbitrary network configurations are considered in this thesis. The load balancing problems for indivisible loads are long known as an NP-complete problems and remain challenging tasks in the research field. In the divisible load theory (DLT) literature, the divisible load distribution strategies are all centralized and the researchers attempt to obtain the closed-form solutions for each network configuration with different constraints, such as bus, linear daisy chain, multilevel tree networks, with or without front-end, etc. However, in some cases, when arbitrary networks are considered, it is very hard to obtain the closed-form solutions for load balancing/scheduling problems. Furthermore, in a distributed computer system, if there is no computer which has all information about the system, centralized strategies cannot be carried out and the load balancing/scheduling can only be achieved by distributed strategies. In this thesis, we proposed novel distributed strategies via virtual routing approach for both indivisible load balancing and divisible load scheduling problems on arbitrary distributed computer networks. The comparison between the traditional equal load balancing/scheduling strategies and our proposed strategies via virtual routing demonstrated that our proposed strategies are much better in the quality of solutions and the time performance.

For the *static* load balancing problem, we considered several constraints such as, job flow rate, communication delays due to the underlying network, processing delays at the processors in the problem formulation. We formulated the problem of load balancing as a constrained non-linear minimization problem to minimize the mean response time of the jobs that arrive in a distributed system for processing, and as a solution approach, we have transformed the problem into an equivalent problem of routing under the above set of constraints. An algorithm named "Load Balancing via Virtual Routing" (LBVR) was proposed, the convergence rate of which was proven to be 2, which is *super-linear*. The approach proposed is conclusively to yield an optimal solution, when the delay functions defined in Chapter 3.1 are assumed to be convex functions. When the delay functions were non-convex functions, a *necessary* condition was derived to obtain an optimal solution to the problem. The rate of convergence to the optimal solution obtained by LBVR algorithm when compared with that of LK algorithm [12] was tested and the results are consistent with the theoretical findings. We have also shown that LBVR algorithm guarantees that no job is left unprocessed once it enters the system for processing and the solution approach is cycle-free (Lemma 3.3). Although the studies attempted in [12, 38] are the ones that are considered close to our modelling, there exist some key differences. In [38], while the formulation was generic, only one class of jobs was considered. In [12] multi-class of jobs were considered under a more-or-less similar formulation. However, in the latter case, the underlying node model was different, in the sense that, each node i treats its neighboring nodes differently according to the transfer vector of class-k jobs. In our model, we relax this constraint and consider a generic situation in which the entire set of class-k jobs arriving at one node (from other nodes and also externally at the node), are considered as a whole while balancing the entire load.

For the *dynamic* load balancing problem, we classified the dynamic load balancing algorithms into three policies: QAP policy: queue adjustment policy, RAP policy: rate adjustment policy, and QRAP policy: combination of queue and rate adjustment policy. We proposed

two algorithms named Rate based Load Balancing via Virtual Routing (RLBVR) and Queue based Load Balancing via Virtual Routing (QLBVR), which belong to RAP and QRAP, respectively. These two algorithms are based on LBVR and hence the computation overheads are small. We have used Estimated Load Information Scheduling Algorithm (ELISA) and Perfect Information Algorithm (PIA) [19] to present QAP policy. We constructed several dynamic job arrival rate patterns and carried out rigorous simulation experiments to compare the performance of the algorithms under different system loads, with different status exchange intervals T_s. Through the rigorous experiments, we have shown that when the system loads are low or moderate, algorithms of RAP policy are preferable with longer T_s. When the system loads are fairly high, QAP policy and QRAP policy performance much better than RAP policy. In this situation, QAP and QRAP policies can obtain a smaller mean response time of jobs with shorter T_s. We also demonstrated that QRAP performances relatively well in most of the situations and it is suitable when the system loads are fluctuating. From our experiments, we have clearly identified the relative metrics of the performance of the proposed algorithms and we were able to recommend the use of suitable algorithms for different loading situations. Our system model and experimental study can be directly extended to large size networks such as multidimensional hypercubes networks to test their performance.

We extended an approach for scheduling divisible loads originating from single or multiple sites on arbitrary networks. We first proposed a generalized mathematical model and formulated the scheduling problem as an optimization problem with an objective to minimize the processing time of the loads. A number of theoretical results on the solution of the optimization problem were derived. On the basis of these results, we proposed an efficient algorithm for scheduling divisible loads using the concept of load balancing via *virtual routing* for an arbitrary network configuration. The proposed algorithm has major three attractive features. Firstly, the algorithm is simple to realize and can be implemented in a *distributed* fashion. The second one is in its style of working by avoiding the need for generating a timing

diagram explicitly for any complex networks having an arbitrary network topology. This is one of the strengths of this algorithm, as the proposed solution approach can be applied even to large scale cluster/grid computing networks. The last one is its capability of handling divisible loads originating from both single and multiple sites. Illustrative examples and several rigorous numerical experiments were presented to capture the performance of the algorithm. When divisible loads originated from single node, we compared the proposed algorithm with a recently proposed RAOLD algorithm which is based on minimum cost spanning tree. When divisible loads originated from multiple sites, we testified the performance on sparse, medium and densely connected networks. From rigorous simulations, we proved that the proposed algorithm is efficient to solve the scheduling problems on arbitrary networks. This is the first time in the DLT literature that such an approach employing load balancing via virtual routing is attempted.

There are several possible immediate future extensions for the work in this thesis. In our research work, we assumed that each node in the system has infinite buffer size. Actually, in the distributed systems, the buffer size of each node is finite. For example, in some cases, more jobs will be sent to the nodes with higher processing capabilities according to some optimal solutions. However, if the buffer of such node is limited and the node cannot hold all the jobs, some of the incoming jobs will be dropped and then, error occurs. Hence, designing efficient resource-aware strategies are crucial for distributed load balancing. Such constraints can be considered in both static and dynamic situations.

As far as processing arbitrary divisible loads is concerned, the communication delay and processing delay are assumed to be linear functions. Most of the algorithms and strategies proposed in DLT cannot handle the situations where these delay functions are assumed to be nonlinear with some kinds of start-up costs [90]. When these nonlinear functions are used in DLT, there are several open questions, such as, “does **Rule A** (optimal sequence) still hold?”, “what is the speedup?”, etc. Furthermore, in the divisible scheduling problems, we assume

that all the nodes and links in the network are free before we can start to obtain an optimal solution. However, in the real distributed systems, when a divisible load arrives, some of the nodes in the system may be occupied by other divisible load under processing. In this case, we must consider the release time of each node and add some constraints in our problem formulation. It would be interesting to propose some strategies to solve such problems on arbitrary networks.

Bibliography

[1] M.M. Eshaghian, *Heterogeneous Computing,* Artech House, 1996.

[2] D. Gerogiannis and S.C. Orphanoudakis, "Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks," *IEEE Trans. Parallel and Distributed Systems*, no. 4, pp. 994-1013, 1993.

[3] K.Y. Tieng, F. Ophir, and L.M. Robert, "Parallel Computation in Biological Sequence Analysis," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 283-294, March 1998.

[4] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill Book Co., 1993.

[5] A.N. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *Journal of the Association for Computing Machinery*, vol. 32, no. 2, pp. 445-465, April 1985.

[6] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Eng.*, vol. 12, no. 5, pp. 662-675, May 1986.

[7] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Balancing," *Performace Evaluation*, vol. 6, pp. 53-68, 1986.

[8] B. Shirazi, A.R. Hurson, and K. Kavi,"Scheduling and Load Balancing in Parallel and Distributed Systems," *IEEE Computer Society Press,* Los Alamitos, CA, USA, 1995.

[9] D. Grosu, A.T. Chronopoulos, and M.Y. Leung, "Load Balancing in Distributed Systems: An Approach Using Cooperative Games," *IPDPS 2002* Fort Lauderdale, Florida, USA, April 2002.

[10] Z. Zeng and V. Bharadwaj, "Design and Analysis of a Non-Preemptive Decentralized Load Balancing Algorithm for Multi-Class Jobs in Distributed Networks," *Computer Communications*, 27, pp. 679-694, 2004.

[11] H. Kameda, J. Li, C. Kim, and Y. Zhang, *Optimal Load Balancing in Distributed Computer Systems*, London: Springer-Verlag, 1996.

[12] J. Li and H. Kameda, "Load Balancing Problems for Multi-class Jobs in Distributed/Parallel Computer Systems," *IEEE Trans. Computers*, vol. 47, no. 3, pp. 322-332, March 1998.

[13] Z. Zeng and V. Bharadwaj, "A Static Load Balancing Algorithm via Virtual Routing," In the Proceedings of the International Conference on *Parallel and Distributed Computing and Systems,* Marina del Rey, CA, USA, November, 2003.

[14] J. Yao and V. Bharadwaj, "Design and Performance Analysis of Divisible Load Scheduling Strategies on Arbitrary Graphs," To appear in *Cluster Computing*, 2004.

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

[16] V. Bharadwaj, D. Ghose, and T.G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed System," *Special Issue on Divisible Load Scheduling in Cluster Computing,* Kluwer Academic Publishers, vol. 6, no. 1, pp. 7-18, January 2003.

[17] Y.C. Cheng and T.G. Robertazzi, "Distributed Computation with Communication Delays," *IEEE Trans. Aerospace and Electronic Systems,* 24, pp. 700-712, 1988.

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

[19] L. Anand, D. Ghose, and V. Mani, "ELISA: An Estimated Load Information Scheduling Algorithm for Distributed Computing System," *Computers and Mathematics with Applications*, 37 (1999), pp. 57-85.

[20] D. Evans and W. Butt, "Dynamic Load Balancing Using Task-Transfer Probabilities," *Parallel Computing*, vol. 19, pp. 279-301, 1993.

[21] C. Walshaw and M. Berzins, "Dynamic Load-Blancing for PDE Solvers on Adaptive Unstructured Meshes," *Concurrency: Practice and Experience*, vol. 7, pp. 17-28, 1995

[22] J. Watts and S. Taylor, "A Practical Approach to Dynamic Load Balancing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 235-248, March 1998.

[23] Y. Zhang, H. Kameda, and K. Shimizu, "Adaptive Bidding Load Balancing Algorithms in Heterogeneous Distributed Systems," *Proc. IEEE Second Int'l Workshop Modeling, Analysis, and Simulation of Computer and Telecomm. Systems*, pp. 250-254, Durbam, N.C., January 1994.

[24] D. Grosu and A.T. Chronopoulos, "A Game-Theoretic Model and Algorithm for Load Balancing in Distributed Systems," *Proc. 16th International Parallel & Distributed Symp.*, Ft. Lauderdale, Florida, USA, April 2002.

[25] J. Li and H. Kameda, "Optimal static load balancing of multi-class jobs in a distributed computer system," *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on* , pp. 562-569, 1990.

[26] T.G. Robertazzi, "Ten Reasons to Use Divisible Load Theory," *Computer*, vol. 36, no. 5, pp. 63-68, May 2003.

[27] D. Bertsekas and R. Gallager, *Data Networks*, Prentice-Hall, Inc., 1992.

[28] N. U. Prabhu, *Foundations of Queuing Theory*, Kluwer Academic Publishers, 1997.

[29] N.G. Shivaratri, P. Krueger, and M. Singhal,"Load Distributing for Locally Distributed Systems," *Computer,* vol. 25, no. 12, pp. 33-44, December 1992.

[30] S.P. Dandamudi, "The Effect of Scheduling Discipline on Sender-Initiated and Receiver-Initiated Adaptive Load Sharing in Homogeneous Distribted System," *Proc. Int. Conf. Distributed Computing System,* Vancouver, 1995.

[31] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Engneering,* vol. 12, no. 5, pp. 662-675, May 1986.

[32] C. Cunha, P. Kacsuk, and S.C. Winter, *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments,* Huntington, N.Y.: Nova Science Publishers, 2001.

[33] A.N. Tantawi and D. Towsley, "A General Model for Optimal Static Load Balancing in Star Network Configurations," *Proc. Performance'84,* E. Gelenbe, Ed., pp. 277-291. North-holland: Elsevier Science Publishers B. V., 1985.

[34] D. Ghose and H.J. Kim, "Load Patitioning and Trade-off Study for Large Matrix-Vector Computations in Multicast Bus Networks with Communiction Delays," it Journal of Parallel and Distributed Computing, vol. 54, 1998.

[35] C. Kim, and H. Kameda, "An Algorithm for Optimal Static Load Balancing in Distributed Computer Systems," *IEEE Trans. Computers*, vol. 41, no. 3, pp. 381-384, Mar. 1992.

[36] J. Li and H. Kameda, "Optimal Static Load Balancing in Tree Network Configurations with Two-Way Traffic," *Computer Networks and ISDN Systems,* vol. 25, no. 12, pp. 1335-1348, 1993.

[37] J. Li and H. Kameda, "A Decomposition Algorithm for Optimal Static Load Balancing in Tree Hierarchy Network Configurations," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 5, pp. 540-548, May 1994.

[38] K.W. Ross and D.D. Yao, " Optimal Load Balancing and Scheduling in a Distributed Computer System," *Journal of the Association for Computing Machinery*, vol 38, no. 3, pp. 676-690, July 1991.

[39] C. Kim and H. Kameda, "Optimal Static Load Balancing of Multi-Class Jobs in a Distributed Computer System," *IEEE Trans. IEICE*, vol. 73, no. 7, pp. 1207-1214, July 1990.

[40] Y. Zhang, K. Hakozaki, H. Kameda, and K. Shimizu, "A Performance Comparison of Adaptive and Static Load Balancing in Heterogeneous Distributed Systems," *Proc. IEEE 28th Ann. Simulation Symp.*, pp. 332-340, Phoenix, Ariz., April 1995.

[41] E. Choi, "Performance Test and Analysis for an Adaptive Load Balancing Mechanism on Distributed Server Cluster Systems," *Future Generation Computer Systems*, vol. 20, pp. 237-247, 2004.

[42] B.G. Lawson, E. Smirni, and D. Puiu, "Self-adapting Backfilling Schedulers for Parallel Systems," *International Conference on Parallel Processing, 2002*, Vancouver, B. C., August 2002.

[43] O. Akay and K. Erciyes, "A dynamic load balancing model for a distributed system," *Mathematical and Computational Applications*, vol. 8 (1-3), pp. 353-360, 2003.

[44] M. Mitzenmacher, "How Useful Is Old Information?," *IEEE Trans. Parallel and Distributed Systems*, vol. 11, no. 1, pp. 6-20, January 2000.

[45] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster," *IEEE Trans. Parallel and Distributed Systems*, vol. 11, no. 7, pp. 760-768, July 2000.

[46] A.E. Kostin, I. Aybay, and G. Oz, "A Randomized Contention-Based Load-balancing Protocol for a Distributed Multiserver Queuing System," *IEEE Trans. Parallel and Distributed Systems*, vol. 11, no. 12, pp. 1252-1272, December 2000.

[47] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094-1104, October 2001.

[48] L. Fratta, M. Gerla, and L. Kleinrock, "The Flow Deviation Network Design," *Networks*, vol. 3, pp. 97-133, 1973.

[49] J. Sohn and T.G. Robertazzi, "Optimal Divisible Job Load Sharing on Bus Networks," *IEEE Trans. Aerospace and Electronic Systems,* 1, 1996.

[50] J. Blazewicz, M. Drozdowski, and M. Markiewicz, "Divisible Task Scheduling-Concept and Verification," *Parallel Computing,* Elsevier Science, vol. 25, pp. 87-98, January 1999.

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

[52] T.H. Jui, H.J. Kim, and T.G. Robertazzi, "Scalable Scheduling in Parallel Processors," *Conference on Information Sciences and Systems,* Princeton University, March 20-22, 2002.

[53] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram, "Scheduling a Divisible Task in a 2-Dimensional Mesh," *Discrete Applied Mathematics,* 94(1-3), pp. 35-50, June 1999.

[54] S.K. Chan, V. Bharadwaj, and D. Ghose, "Large Matrix-vector Products on Distributed Bus Networks with Communication Delays using the Divisible Load Paradigm: Performance Analysis and simulation", *Mathematics and Computers in Simulation*, 58, pp. 71-79, 2001.

[55] K. Li, "Parallel Processing of Divisible Loads on Partitionable Static Interconnection Networks", Special Issue on Divisible Load Scheduling in *Cluster Computing*, Kluwer Academic Publishers, vol. 6, no. 1, pp. 47-56, January 2003.

[56] V. Bharadwaj and G. Barlas, "Efficient Scheduling Strategies for Processing Multiple Divisible Loads on Bus Networks," *Journal of Parallel and Distributed Computing*, vol. 62, no. 1, pp. 132-151, January 2002.

[57] D. Ghose, H. J. Kim, and H. T. Kim, "Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources," Technical Report KNU/CI/MSL/001/2003, 2003.

[58] W. Glazek, "A Multistage Load Distribution Strategy for Three-Dimensional Meshes," Special Issue on Divisible Load Scheduling in *Cluster Computing*, Kluwer Academic Publishers, vol. 6, no. 1, pp. 31-40, January 2003.

[59] H.M. Wong, D. Yu, V. Bharadwaj, and T.G. Robertazzi, "Data Intensive Grid Scheduling: Multiple Sources with Capacity Constraints," In the Proceedings of the International Conference on *International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, November 2003.

[60] M. Drozdowski, *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems,* Wydawnictwa Politechniki Poznanskiej, (In English) Book no. 321, Poznan, Poland, 1997.

[61] V. Bharadwaj, X.L. Li, and C. C. Ko, "On the Influence of Start-up costs in Scheduling Divisible Loads on Bus Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 12, pp. 1288-1305, December 2000.

[62] D.P. Bertsekas, *Nonlinear Programming*, Belmont, Mass.: Athena Scientific, c1995.

[63] V. Bharadwaj and W.H. Min, "Scheduling Divsible 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, March 2004.

[64] F. Seredynski, "Discovery with Genetic Algorithm Scheduling Strategies for Cellular Automata," in A. E. Eiben, T. Back, M. Schoenauer and H.-P. Schwefel (Eds.), *Parallel Problem Solving from Nature - PPSN V*, Lecture Notes in Computer Science 1498, Springer, pp. 643-652, 1998.

[65] F. Seredynski and A.Y. Zomay, "Sequential and Parallel Cellular Automata-Based Scheduling Algorithms," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 10, pp. 1009-1023, Octorber 2002.

[66] A. Colorni, M. Dorigo, and V. Maniezzo, "Distributed Optimization by Ant Colonies," *Proceedings of ECAL91-European Conference on Artificial Life*, Paris, France, 1991.

[67] J. Hao and J. Pannier, "Simulated Annealing and Tabu Search for Constraint Solving," *Fifth Intl. Symposium on Artifcial Intelligence and Mathematics (AIM'98)*, Fort Lauderdale Florida, USA, January 4-6, 1998.

[68] G. Manimaran and C.S.R. Murthy, "An Efficient Dynamic Scheduling Algorithm For Multiprocessor Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems*, vol 9, no. 3, pp. 312-319, March 1998.

[69] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd ed., Addison-Wesley Publisher, 2000.

[70] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram, "Scheduling a Divisible Task in a 2-Demensional Mesh," *Discrete Applied Math.,* vol. 94, no. 1-3, pp. 35-50, 1999.

[71] P.D. Spraggis, D. Towsley, and C.G. Cassandras, "Optimality of Static Routing Policies in Queuing Systems with Blocking," *Decision and Control, 1991., Proceedings of the 30th IEEE Conference*, vol. 1, pp. 809-814, 1991.

[72] M.D. Grammatikakis, J.S. Jwo, M. Kraetzl, and S.H. Wang, "Dynamic and static packet routing on symmetric communication networks," *GLOBECOM '94. Communications: The Global Bridge., IEEE*, vol 3, pp. 1591-1595, 1994.

[73] H. Tokumaru, K. Kinoshita, N. Yamai, and K. Murakami, "A fast and efficient dynamic routing for inter-agent communications," *Info-tech and Info-net, 2001. Proceedings. ICII 2001 - Beijing*, vol 4, pp. 73 -78, 2001.

[74] R. Jain, *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, 1991.

[75] R.K. Ahuja, T.L. Mananti, and J. B. Orlin, *Network Flows*, Prentice-Hall, Inc., 1993.

[76] A.M. Al-Saideen and S. Bataineh, "A Heuristic Algorithm for Scheduling Multi-Classes of Tasks in Multiprocessor Systems," *International Journal of Computers and Applications,* January 1999.

[77] D.J. Wilde and C.S. Beightler, *Foundations of Optimization*, Englewood Cliffs N.J.: Prentice Hall, 1967.

[78] S. Dafermos, "The Traffic Assignment Problem for Multiclass-user Transportation networks," *Transportation Sci.* 6, pp. 73-87, 1972.

[79] M. Avriel, *Nonlinear Programming Analysis and Methods*, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1997.

[80] E. Altman and H. Kameda, "Equilibria for Multiclass Routing in Muli-Agent Networks," *Proceeding of the 40th IEEE Conference on Decision and Control*, Orlando, USA, December 2001.

[81] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 3, pp. 236-247, March 2003.

[82] M.J. Zaki, W. Li, and S. Parthasarathy, "Customized Dynamic Load Balancing for a Network of Workstations," *Journal of Parallel and Distributed Computing*, 43, pp. 156-162, 1997.

[83] F.E. Beichelt and L.P. Fatti, *Stochastic Processes and Their Application*, Taylor & Francis, 2002.

[84] D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford DASH Multiprocessor," *IEEE Trans. Computers*, vol. 25, no. 3, pp. 63-79, 1992.

[85] http://www.top500.org/ORSC/1997/paragon.html.

[86] V. Bharadwaj, H.F. Li, and T. Radhakrishnan, “Scheduling Divisible Loads in Bus Networks with Arbitrary Processor Release Times,” *Computer Math. Applic.,* vol. 32, no. 7, 1996.

[87] X. Li, V. Bharadwaj, and C.C. Ko, “Scheduling Divisible Tasks on Heterogeneous Single-level Tree Networks with Finite-size Buffers,” *IEEE Trans. Aerospace and Electronic Systems*, vol. 37, pp. 1298-1308, January 2001.

[88] S.C. Datermos and F.T. Sparrow, “The Traffic Asignment Problem for a General Network,” *Transportation Science,* vol. 6, pp. 73-87, 1969.

[89] C.Y. Cheng, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* The MIT Press, 1994.

[90] D.A.L. Piriyakumar and C.S.R. Murthy, “Distributed Computation for a Hypercube Network of Sensor-Driven Processors with Communication Delays Including Setup Time,” *IEEE Trans. Systems Man and Cybernetics-Part A: Systems and Humans,* vol. 28, no. 2, pp. 245-251, March 1998.

Author's Publications

[1] Z. Zeng and V. Bharadwaj, "Design and Analysis of a Non-Preemptive Decentralized Load Balancing Algorithm for Multi-Class Jobs in Distributed Networks," *Computer Communications*, 27, pp. 679-694, 2004.

[2] Z. Zeng and V. Bharadwaj, "Distributed Scheduling Strategy for Divisible Loads on Arbitrarily Configured Distributed Networks using Load Balancing via Virtual Routing," *Submitted after 2nd revision, IEEE Trans. Parallel and Distributed Systems*, 2004.

[3] Z. Zeng and V. Bharadwaj, "Design and Performance Evaluation of Distributed Dynamic Load Balancing Algorithms in Heterogeneous Distributed Computer Systems," *Submitted to Cluster Computing*, 2004.

[4] Z. Zeng and V. Bharadwaj, "Divisible Load Scheduling on Arbitrary Distributed Networks via Virtual Routing Approach," *The Tenth International Conference on Parallel and Distributed Systems (ICPADS 04)*, Newport Beach, California, July 7-9, 2004.

[5] Z. Zeng and V. Bharadwaj, "Rate-Based and Queue-Based Dynamic Load Balancing Algorithms in Distributed Systems," *The Tenth International Conference on Parallel and Distributed Systems (ICPADS 04)*, Newport Beach, California, July 7-9, 2004.

[6] Z. Zeng and V. Bharadwaj, "A Static Load Balancing Algorithm via Virtual Routing," In the Proceedings of the International Conference on *Parallel and Distributed Computing and Systems,* Marina del Rey, CA, USA, pp. 244-249, November 3-5, 2003.

Appendix: Queue Length Estimation

We model the arrivals and departures at node i as independent Poisson processes with means λ_i and μ_i, respectively. When node i is to estimate the load on processor j as $n_j(T_{01})$ at a time $T_{01} = T_0 + T_e$, and the actual load on j at T_0 is $n_j(T_0)$. Then

$$n_j(T_{01}) = n_j(T_0) + m_j(T_e), \tag{A.1}$$

where $m_j(T_e)$ is the actual number of jobs added to the queue at processor j in the time interval T_e. It is obvious that $m_j(T_e)$ could be negative, zero, or positive, depending on whether the number of arrivals is less, equal, or more, than the departures, respectively. But $n_j(T_{01})$ cannot take negative values. So the smallest value that $m_j(T_e)$ can take is $-n_j(T_0)$. Let us denote the estimate of $n_j(T_{01})$ as $\hat{n}_j(T_{01})$. Then

$$\hat{n}_j(T_{01}) = n_j(T_0) + \sum_{m_j(T_e)=-n_j(T_0)}^{\infty} m_j(T_e) \times p(m_j(T_e)), \tag{A.2}$$

where $p(m_j(T_e))$ is the probability that $m_j(T_e)$ number of jobs that have been added to the queue in the interval T_e, and is given by

$$\begin{aligned} p(m_j(T_e)) &= \sum_{k=m_j(T_e)}^{\infty} p(k \textit{ arrivals in } T_e) \times p([k - m_j(T_e)] \textit{ departures in } T_e) \\ &= \sum_{k=m_j(T_e)}^{\infty} \frac{e^{-\lambda_j T_e}(\lambda_j T_e)^k}{k!} \times \frac{e^{-\mu_j T_e}(\mu_j T_e)^{k-m_j(T_e)}}{(k - m_j(T_e))!}. \end{aligned} \tag{A.3}$$

Depending on the accuracy required, computations of $p(m_j(T_e))$ and $\hat{n}_j(T_{01})$ can be terminated after computing a sufficiently large number of terms in (A.2) and (A.3). Note that in (A.3), instead of using the actual arrival rate λ_j, the estimated arrival rate $\hat{\lambda}_j$ is used during computation.