# RESOURCE UNAWARE LOAD DISTRIBUTION STRATEGIES FOR PROCESSING DIVISIBLE LOADS IN NETWORKED COMPUTING ENVIRONMENTS


JIA JINGXI

(*B.Eng, UESTC*)



A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2009

# Acknowledgements

I would like to give my heartfelt thanks to my supervisor, Prof. Bharadwaj Veeravalli, for his guidance, support and encouragement throughout my study. His advices and assistance in and beyond the academic and research has helped me a lot during my stay in NUS.

I would also like to thank my parents as well as my wife. They give me their unconditional love and support.

Finally, I want to thank my friends and my colleagues in CNDS lab for their kind assistance on research and other issues. They make my stay in Singapore enjoyable and memorable. I will definitively miss the joyous discussion during lunch time and the afternoon Kopi-club.

# Contents

**Acknowledgements** **i**

**Summary** **v**

**List of Figures** **vii**

**List of Tables** **ix**

**List of Abbreviations** **x**

**1 Introduction** **1**

1.1 Scheduling Divisible Loads Under Different Communication Models and Network Topologies. . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.1.1 Communication Models . . . . . . . . . . . . . . . . . . . . . . 4

1.1.2 Different Network Topologies . . . . . . . . . . . . . . . . . . . 5

1.2 Scheduling Divisible Loads Under Other Real-life Conditions . . . . . 9

1.3 Scheduling Divisible Loads in The Resource Unaware Context . . . . 15

1.4 Objectives and Organization of The Thesis . . . . . . . . . . . . . . . 17

1.4.1 General Focus, Contributions and Scope . . . . . . . . . . . . 17

**2 Scheduling in Linear Networks** **21**

2.1 Problem Setting and Assumptions . . . . . . . . . . . . . . . . . . . . . 21

2.2 Design of Resource Unaware Scheduling Strategies . . . . . . . . . . . 24

2.2.1 Design and Analysis of Early Start Strategy . . . . . . . . . . 27

2.2.2 Design and Analysis of Wait-and-Compute Strategy . . . . . . 30

2.3 Performance Evaluation and Discussions . . . . . . . . . . . . . . . . 33

**3 Scheduling in Multi-Level tree networks** **46**

3.1 Problem Definition, Assumptions and Remarks . . . . . . . . . . . . 46

3.2 Static Network Parameter (SNP) Case . . . . . . . . . . . . . . . . . 50

3.3 Dynamic Network Parameter (DNP) Case . . . . . . . . . . . . . . . 60

3.4 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 65

3.4.1 Experiment with Static Network Parameter Case using SLD strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

3.4.2 Experiment with Dynamic Network Parameter Case using DLD Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

**4 Issues in Handling Divisible loads on Arbitrary Networks** **82**

4.1 Probing & Reporting Techniques . . . . . . . . . . . . . . . . . . . . 82

4.2 Common Spanning Trees - Performance Evaluation . . . . . . . . . . 85

4.2.1 Problem Formulation and Notations . . . . . . . . . . . . . . 85

4.2.2 Common Spanning Tree Routing Strategies . . . . . . . . . . 86

4.2.3 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . 91

**5 Scheduling Multi-source Divisible Loads in Arbitrary Networks 100**

5.1 General Introduction of the Presented Problem: Scope, Network Model and Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . 100

5.1.1 Network Model and Problem Formulation . . . . . . . . . . . 101

5.2 Static Scheduling Strategy (SSS) . . . . . . . . . . . . . . . . . . . . . 104

5.2.1 Adapting to Resource Unaware Case . . . . . . . . . . . . . . 110

5.3 Dynamic Scheduling Strategy (DSS) . . . . . . . . . . . . . . . . . . . 111

5.4 Analysis of DSS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

5.5 Performance Evaluation and Discussion . . . . . . . . . . . . . . . . . 121

5.5.1 Performance of SSS . . . . . . . . . . . . . . . . . . . . . . . 121

5.5.2 Performance of DSS . . . . . . . . . . . . . . . . . . . . . . . 127

**6 Conclusions and Future Recommendations 133**

**Bibliography 138**

**List of Publications 152**

# Summary

Nowadays, network based computation has attracted more and more attention, as it provides an efficient solution for processing computational intensive tasks/loads. This thesis considers processing one type of the loads - divisible loads, in networked computing environments. We focus on the resource unaware case, where the scheduler does not know the speed information of the network in advance. Networks with different topologies are considered and studied. We also address the problem of scheduling multi-source divisible loads.

We first consider the resource unaware linear networks and multi-level tree networks. A probing technique is applied to detect the link and processor speeds, which are then used by the scheduler to generate a feasible schedule. The characteristic of the network topology is explicitly considered in designing efficient probing based scheduling strategies.

We then argue the usefulness of the probing technique in networks without a regular topology and/or when multiple sources exist. An alternative reporting based technique is suggested. We also study and analyze the performance of the different spanning trees in scheduling divisible load(s) in arbitrary networks.

Finally, the generalized problem of scheduling multi-source divisible loads on arbitrary networks is addressed. Starting from the resource aware case, we proposed

efficient strategies to schedule the multi-source loads in two different cases - when no new loads arrive at the system and when new loads may arrive as time progresses. We also demonstrate that by using a reporting based scheme, our strategies can be easily adapted to the resource unaware case. Queuing model is applied to analyze the systems and rigorous simulation experiments are carried out to validate our algorithms.

# List of Figures

2.1 Linear Daisy Chain Network Architecture with $n$ processors and $(n-1)$ links . . . . . . . . . . 22

2.2 Timing Diagram For Early Start Strategy . . . . . . . . . . 25

2.3 Network model for Example 1 . . . . . . . . . . 34

2.4 Figure of $T_{WCS}(i)$ for Different $\varepsilon$ and $\eta$ when $n = 15$ $r_f = 0.75$ . . . . 43

2.5 Figure of $T_{WCS}(i)$ for Different $\varepsilon$ and $\eta$ when $n = 15$ $r_f = 0.25$ . . . . 44

3.1 General tree network . . . . . . . . . . 47

3.2 Time Diagram For SLD Strategy . . . . . . . . . . 51

3.3 Demonstration of Congestion . . . . . . . . . . 53

3.4 Virtual Tree Construction . . . . . . . . . . 57

3.5 Equivalent Processor for Single-level Tree . . . . . . . . . . 58

3.6 Flow Chart for SLD Strategy . . . . . . . . . . 59

3.7 Flow Chart for DLD Strategy . . . . . . . . . . 66

3.8 Tree model for the experiment . . . . . . . . . . 67

3.9 Virtual Tree and Equivalent Single Level Tree of SNP Case . . . . . . 72

3.10 The variance of $w$ and $z$ with time . . . . . . . . . . 75

3.11 Virtual Tree and Equivalent Single Level Tree of DNP Case . . . . . 80

4.1 An arbitrary graph network and spanning trees (number on the links denote the link weights and the number near the nodes denote the processor weights). (a) An arbitrary graph network $G$ with 8 processing nodes; (b) Minimum spanning tree; (c) Shortest path spanning tree; (d) Fewest hops spanning tree; (e) Robust spanning tree; (f) Minimum equivalent network spanning tree. . . . . . . . . . . . . . . . . . . . . . 90

4.2 Network eccentricity simulation results for $10, 100$, and 200 nodes network with low and high speed links . . . . . . . . . . . . . . . . . . . 93

4.3 Total processing time simulation results for $10, 100$, and 200 nodes with low and high processing speeds in a network with low speed links . . 94

4.4 Total processing time simulation results for $10, 100$, and 200 nodes with low and high processing speeds in a network with high speed links . . 95

5.1 Network models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

5.2 Markov chain of two regions case . . . . . . . . . . . . . . . . . . . . 117

5.3 Experiment Results for the Static Case . . . . . . . . . . . . . . . . . 124

5.4 Total Processing Time of SSS with Different Load Size and Number of Sources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

5.5 The Average Queue length of Loosely-coupled Network and Tightly-coupled Network With Respect to Different $\lambda$ . . . . . . . . . . . . . 130

# List of Tables


2.1 Experimental Results when $r_f = 0.75$ . . . . . . . . . . . . . . . . . . 39

2.2 Experimental Results when $r_f = 0.25$ . . . . . . . . . . . . . . . . . . 39

3.1 PTC and CTC responses from Processors . . . . . . . . . . . . . . . . . 68

3.2 Load Distribution of SNP Case . . . . . . . . . . . . . . . . . . . . . . 71

3.3 Load Distribution of DNP Case . . . . . . . . . . . . . . . . . . . . . . 79

4.1 List of notations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

4.2 Comparison of complexities[1] and performances of various spanning tree algorithms for divisible load scheduling with RAOLD-OS scheduling strategy for arbitrary graphs . . . . . . . . . . . . . . . . . . . . . . . 99

5.1 Regions' Equivalent Computation Capacities for Symmetric Networks 129

5.2 Regions' Equivalent Computation Capacities for the General Case . . 131

5.3 Experimental Results for the General Case . . . . . . . . . . . . . . . 131

# List of Abbreviations

**BFS:** breadth-first search

**CTC:** communication task completion message

**CP:** computation phase

**DLD:** dynamic load distribution

**DLT:** divisible load theory

**DNP:** dynamic network parameter

**DSS:** dynamic scheduling strategy

**ESS:** early start strategy

**EST:** minimum equivalent network spanning tree

**FHT:** fewest hops spanning tree

**GP:** graph partitioning scheme

**MST:** minimum spanning tree

**PL:** probing load

**PP:** probing phase

**PSD:** probing and selective distribution

**PTC:** processing task completion message

**RAOLD-OS:** resource-aware optimal load distribution with optimal sequencing

**RB-SSS:** reporting based static scheduling strategy

**RST:** robust spanning tree

**SDS:** sequential dispatching strategy

**SLD:** static load distribution

**SNP:** static network parameter

**SPT:** shortest path spanning tree

**SSS:** static scheduling strategy

**WCS:** wait-and-compute strategy

# Chapter 1

# Introduction

Network based computation is an active area of current research. Many applications, such as image processing, large matrix production, protein/DNA sequencing, result in large scale computationally intensive tasks. Handling such tasks on a single workstation can be quite time-consuming, and hence people resort to network based computation. Compared to the traditional supercomputer solution, network based computation offers a lower cost/performance ratio for handling large-volume, computational-intensive tasks.

These computational-intensive tasks, depending on the data dependencies among themselves, can be grouped into three different categories: indivisible tasks, modular divisible tasks, and divisible tasks. The divisible tasks, which are normally referred to as divisible loads in the literature, are assumed to have no precedence relationship among the data. Therefore, they can be arbitrarily partitioned into arbitrary size of load fractions, and these load fractions can be processed independently. One can use divisible loads to model many of the real-life tasks emerging from scientific and engineering fields.

The research of scheduling divisible loads in networked computing environment dates back to the 1988, with the initial works done by two independent groups Cheng

and Robertazzi [1] and Agrawal and Jagadish [2]. A formal mathematical framework was first provided by [3] and the theory was formally referred to as Divisible Load Theory (DLT). DLT proposes elegant solutions, optimal in many cases, to handle large scale divisible loads on different network models. The processors' computation capacities and the links' communication delays are explicitly captured in the problem formulation to seek optimal, or near optimal solutions. The book [4] summarizes the literature until 1996 including the above mentioned formal theoretical framework and formulations. Two recent survey articles [5, 6] highlight the advantages and the reasons to use the DLT.

Since its inception, the DLT paradigm has been applied in many real life applications, where the computation of tasks is less coupled. To name a few, these include edge-detection application of a large-scale satellite image [7], large-scale matrix-vector product [8], large-scale database search problems [9, 10], use of DLT paradigm with clusters of workstations [11, 12], scheduling divisible loads on grid platforms with APST-DV [13], multimedia applications [14, 15, 16], biological sequences aligning [17], and parallel video processing [18]. A recent work [19] exploits parallelizing the discrete wavelet transform computation, which has a highly coupled recursive computational nature, on a bus network. It shows that by carefully scheduling loads among processors, DLT paradigm can also be applied to the applications with highly coupled recursive computational nature to gain a significant speedup. The DLT literature also contains integer approximation algorithms [20] to cater to the granularity requirement.

For all applications, the underlying networked systems which are about to share the loads, may have different infrastructures. In [21], a parallel system can be characterized as the number of processors, interconnection networks (topologies), number of ports per processor and overlap of communication on computation (communication models). Therefore, modeling the network is a very important issue in the DLT domain. Different network models have been proposed to match real life situations and scheduling divisible loads has been studied under different models carefully. On the other hand, many real life constraints such as buffer size, communication start-up costs, bus release time, and so on, have also been incorporated into the problem, and scheduling divisible loads under these constraints have also been carefully addressed in the DLT literature.

The following subsections will review scheduling divisible loads under the different communication and network models, other real-life conditions, the resource unaware context, and finally conclude with the objective and scope of the thesis.

## 1.1 Scheduling Divisible Loads Under Different Communication Models and Network Topologies.

### 1.1.1 Communication Models

In the DLT literature, an important principle that has been proven conclusively in deriving an optimal scheduling, is referred to as optimality principle [4]. It states that, to minimize the total processing time of the load, all processors which are engaged in computation should finish processing simultaneously. To determine the time instant when each processor finishes computing, the load distribution overhead (communication delay) should be considered carefully, as the DLT paradigm explicitly captures the link communication delay into the problem formulation. Therefore, the communication model is an important issue in designing an efficient divisible load scheduling strategy. One crucial assumption which affects how the communication is carried out is whether a processor is equipped with a front-end or not. A front-end is a co-processor that resides on the chip, responsible for the communication task. "With front-end" is commonly assumed in the literature [24, 25, 26, 27, 28]. In this case, each processor is equipped with a front-end, which off-loads the communication task from that processor and hence, computation and communication can be carried out simultaneously. On the other hand, many works [29, 30, 31, 32] have also addressed the "without front-end" case, where the computation and communication cannot be

overlapped. In this case, including all processors into computation may not render the minimum total processing time.

Another important assumption with respect to the communication model is whether a processor has multiple independent ports for transmission. If a processor has only a single port for transmission, simultaneously transmitting or receiving is not allowed. Most works in the literature adopt the single port assumption implicitly or explicitly. On the other hand, many real-life workstations, especially in point-to-point networks, are capable of performing more than one independent communication with other workstations without interference and hence, many works [33, 34, 35, 36] have also considered the multiple ports model. In this case, multiple transmissions or receptions can be carried out concurrently. However, "with front-end" and single port are still common communication models which can be mapped to many systems.

### 1.1.2 Different Network Topologies

Network topology is another important issue that needs to be carefully considered when designing load scheduling strategies. This is because different network topologies have different characteristics that should be exploited by the scheduling strategies. Many network topology models which are commonly used to model the real networks are bus, linear daisy chain, tree, mesh, graph, etc.

Bus is one of the most common topologies found in today's networked systems. Many of the initial studies [37, 38, 39] in the DLT domain consider the problem of scheduling divisible loads in bus networks. In bus networks, processors are inter-

connected by a shared bus, and hence the communication delay between any two processors is identical. Further, any two processors can communicate with each other directly. The closed form solution of the minimum finish time and the optimal load allocation for bus networks is obtained in [40]. Another work [26], for the first time, proved the optimality principle analytically for the case of bus networks.

Unlike bus networks, in linear daisy chain networks, processors are connected one by one sequentially. Any processor within the chain will receive the load from its predecessor and will relay the load to the rest of the chain. In this manner, the load is percolated down the chain. In [30], an "equivalent processor" concept is proposed, and then is used to determine when to distribute the load down the chain in the "without front-end" case. The same concept of processor equivalence is also adopted in [41] to obtain the ultimate performance limits in linear networks in the presence of communication delay. In contrast to this work, [32] presents an asymptotic performance analysis on the effect of communication delay. Closed-form solution of the optimal load allocation for linear networks is obtained in [24].

A more complex network topology mesh, which belongs to the class of point-to-point networks, has also received lots of attention in the literature. A two-dimensional mesh network with a circuit-switched routing scheme, in which the communication delay is virtually independent of the covered distance, is considered first in [42]. This work proposes a scattering method, and analyzes the performance limit in the presence of communication delays. However, a simplifying assumption that all nodes in the same layer are equivalent is adopted in the performance analysis. This assump-

tion may not be practical. A later work [34] relaxes this assumption and studies a two-dimensional toroidal mesh. It proposes a Peters-Syska scattering algorithm which exhibits a better performance than the one proposed in [42]. Three-dimensional mesh networks with the same circuit-switched routing scheme are considered in [43], and a recursive distribution strategy is proposed. However, this work does not obtain the closed-form solution of the load distributions. The closed-form solution for the load shares assigned to each processor in the three-dimensional mesh is first presented in [44]. A more recent work [45] derives the upper bound of the asymptotic speedup that can be achieved in the generalized k-dimensional mesh. Another work [35] by the same author presents two algorithms using a novel pipelined communication technique to schedule divisible loads on linear arrays and derives the closed-form solution of the parallel processing time and asymptotic speedup. It then generalizes the algorithms to the k-dimensional mesh, and these algorithms exhibits good performance by using pipelined communication and interior initial processors.

The tree network is another important topology which can be mapped to many real-life networks. [29] first considers this type of networks, for both “with front-end” and “without front-end” cases. However, this work only presents the recursive relations among the processors, while a rigorous mathematical solution is missing. The closed-form solution is first presented in [46]. In [4], it has been shown that in single-level tree networks, when the scheduling sequence is fixed, including all processors into computation may not render the optimal results. An important rule, referred to as Rule A, is proposed in this work to exclude the unnecessary processors from the computation. On the other hand, when scheduling sequence is not fixed, [31] solves

the problem of how to find the optimal distributing sequence which admits the minimum total processing time. In contrast with the previous work, where homogeneous trees or single level trees are considered, [47] examines arbitrary processor trees and it also takes into account the overhead induced by the result collection process. [48] considers multi-level tree networks, using a multi-port model of communication. A few open problems on tree networks are discussed in [49], and the asymptotic speedup of various network topologies is systematically studied in [50].

In a recent work, J. Yao et al. [51] moves one step further. It considers scheduling divisible loads on networks with an arbitrary graph topology. They proposes a RAOLD-OS strategy, which works in two phases - it first spans a minimum spanning tree (MST) which is rooted at the source and then schedules the divisible loads on this spanning tree. While this work presents the optimal solution for scheduling on a MST for an arbitrary network, it does not address the problem of whether the MST is the optimal spanning tree which admits a minimum total processing time among all the spanning trees for a given network. The reason why a MST is chosen in this work is probably because the MST has the minimum total link cost, and the authors believe that this characteristic may render the minimum total processing time. However, this is not necessarily the case. P. Byrnes et al. [52] has proven that the problem of finding the best/optimal spanning tree for divisible load distribution on a graph is NP-hard by reducing the SUBSET-SUM problem to this problem. Therefore, many heuristic approaches have been proposed to achieve different targets. A local minimum algorithm is proposed in [52]. This algorithm has a greedy nature and works in a step-wise manner, but in each step this algorithm needs to compute the equiv-

alent computation power of a spanning tree. This leads to very large computational complexity. Darin England et al. [53] has proposed a robust spanning tree to achieve the robustness of the load distribution without sacrificing too much performance. However, among the well known spanning trees, such as shortest path spanning tree, shortest hop spanning tree, minimum spanning tree, robust spanning tree, etc, it is not known which spanning tree offers a better trade off between performance and complexity.

## 1.2 Scheduling Divisible Loads Under Other Real-life Conditions

Besides the communication models and topologies, many other real-life conditions or constraints have also been considered when designing the load scheduling strategies. These efforts make the work more close to certain realistic situations.

Buffer size is one of the real-life constraints which may influence the design of a scheduling strategy. In the DLT literature, it is common to assume that the processing time of a certain load is linearly related to the size of this load. This is true only when the load size is less than the size the processor's main memory (RAM). Any larger load chunk will be stored in the virtual memory, and the computation will be more complex and time-consuming because of the scheduling between the main memory and virtual memory. Scheduling divisible loads under the finite buffer size constraint is first addressed in [54]. The underlying topology is a heterogenous single-level tree

(star network). This work proposes an incremental balancing strategy (IBS) to obtain the load distribution. It has been shown in this work that Rule A is detrimental in the case of finite buffer size. However, the optimality of the IBS strategy has not been proven. Also, [54] does not solve the problem of how to obtain the optimum sequence of activating processors with finite buffer size. This optimum sequence problem is solved in a later work [55]. This work considers two different topologies, star and binomial tree, and proposes a method which guarantees finding the optimal load distribution. Scheduling on distributed multi-level tree networks with buffer constraints is addressed in [56]. Unlike the above mentioned works, where the buffer size constraint mainly refers to inadequate memory size, [57] studies the influence of the communication buffer size on the total processing time of the load. In [58], the finite buffer size is considered together with granularity constraints.

Start-up cost is another important factor to consider. In most realistic data communication and computation, overhead delays exist. Depending on the real-life situations, the overheads in communication may appear in different forms, such as protocol processing delay, queuing delay, delays due to unavailability of communication resources, etc. In the computation process, overheads appear in the forms of layered protocol delays, unpacked delays, processor initialization, etc. While these overheads can be neglected in many cases and a linear cost model can be used to model the communication time and processing time, some works [47, 59, 60, 61] have included the overheads into their models (affine cost model) as a constant start-up cost. In [47], the overheads in query processing and image processing are considered. In [59], overheads are addressed for different network topologies - linear chain, bus,

tree and hypercube, and recursive equations in different cases are presented. This work, however, only considers overheads in communication process. A more general work [60], studies both overheads in communication and computation process. Closed form solutions are derived, for the first time, in this work, and the effect of the start-up cost are discussed. [62] has proven NP-Completeness of scheduling divisible loads on heterogenous star networks with affine cost model, and [57, 61] considers the start-up cost together with the finite buffer constraint.

Fault tolerance is also addressed in the literature. In [63], the effect of fault tolerance on the processing time of an $N$ processor bus network is studied. Correction methods are proposed to handle the unprocessed data by the faulty processors. A more recent work [53] addresses the fault tolerance problem in networks with an arbitrary topology. Unlike [63], where the main contribution is designing strategies to handle the error, [53] proposes a robust spanning tree (RST) which shows a fault tolerant characteristic in nature. The RST is constructed to be neither too "fat" (shallow) nor too "skinny" (deep), and it is shown in [53] that in such a way, RST can strike a balance between time performance and robustness to the data loss caused by node or link failure.

Other works which address the practical concerns can be found in [64, 65, 66, 67, 68, 69]. The research in [64] relaxes a common assumption that all processors are available at the time when the load scheduling starts. It proposes an efficient algorithm to take into account the processor release time in bus networks, and [65] extends the previous work to linear networks. In [66], the processor release time

is considered together with the finite buffer constraint. Instead of minimizing the total processing time of load(s), the research in [67] considers monetary cost as an alternatively objective function. The work [68] considers minimizing both monetary cost and total processing time. Energy use Optimization is addressed in [69], and [70] discusses the combinatorics in the divisible load scheduling.

Further, multi-round algorithms have been proposed to reduce the total processing time of a divisible load by improving the overlap of communication and computation. The initial studies are done in [71, 72], for linear networks and tree networks respectively. In [72], a multi-installment strategy, which starts with small chunks and increases chunk size throughout the load distribution, is proposed and the closed form solution for homogenous systems is derived. This work also discusses the trade-off between the number of processors and the number of installments in absence of overheads. Other multi-round algorithms can be found in [73, 74]. These works, in general, all adopt the linear cost model (i.e., do not consider the start-up cost) and validate their finds through simulation and experiments. The first quantitative result for a multi-round algorithm is presented in [75], which proves the asymptotic optimality of the proposed algorithm. However, [75] also sticks to the linear cost model. This model cannot be used to derive the optimal number of installments, since the impractical infinite large number of installments will be the answer. In [76], communication overheads are considered under the multi-installment setting. A later work [77] considers both overheads in communication and computation processes, and obtains the closed-form solution for homogeneous systems with the start-up cost. A new algorithm, Uniform Multi-Round (UMR), which caters for both homogeneous and het-

erogeneous systems, is proposed. Under the affine cost model, [77] also demonstrates how to compute a near optimal number of installments. Multi-round algorithms have also been proposed to account for performance prediction errors in [78, 79].

Another important issue addressed in the literature is the multi-job scheduling problem. In reality, a system may have multiple divisible loads to process, instead of only one load, and this naturally results in a multi-job scheduling problem. The multi-job and multi-round problems are similar, to some extent. In the latter case, a single divisible load is artificially divided into several installments, which can be regarded as "several loads" because of the load's divisible nature.

Depending on whether the multiple loads originate in a single processor or multiple processors, the multi-job scheduling problem can be categorized into single source problem and multi-source problem. The single source problem is first addressed in [80]. In this work, only one load is considered for distribution at a time and a single-installment technique is used to distribute each load. The strategy is designed to minimize the idle times of processors and to optimize the processing time of all loads. Unlike in [80], [81] proposes a multi-installment multi-job strategy and derives the conditions under which an optimal solution employing multiple installments would exist. Both works consider networks with bus topology. Scheduling multiple loads under linear networks is studied in [82].

In single source problem, the system receives load(s) from a single workstation. However, in many real-life applications, such as in the Grid systems, users can submit the processing loads at different locations. This leads to multiple load origins/sources

in the computing networks. In this scenario, designing an efficient scheduling strategy is much more difficult than in the single source case, since multiple sources must cooperate with each other to share the resources. Because of the complexity, the multi-source scheduling problem has received much less attention in the DLT literature. M. Moges et al. [83, 84] addresses the multi-source scheduling problem on a tree network via linear programming and closed form solutions respectively. Another work by T. Lammie et al. [85] studies the two sources scheduling problem on linear networks. T.G. Robertazzi et al. consolidates the previous results in [86], and L. Xiaolin et al. [87] considers the multi-source problem on single level tree networks. However, the limitation of those works is that they focus on networks with regular topologies, such as linear networks or trees, and in most cases only two load origins (sources) is considered. The generalized case, scheduling multi-source divisible loads on an arbitrary network has not been rigorously addressed.

One may notice that, a similar but different problem of scheduling multi-flows on arbitrary networks has been attempted by using the multi-commodity flow model [88, 89, 90, 91]. However, multi-commodity flow modeling and divisible load scheduling paradigm have different concerns. In multi-commodity flow problems, commodities flow from a set of known sources to a set of known sinks via an underlying network and a major concern is to seek a maximal flow. Therefore, determining routes that provides maximal flow between sources and sinks is a key concern. However, in the DLT domain, every node is a potential sink and the connotation of "sink" as a special kind of node is not found in the DLT problem formulation. Thus, a load fraction is allowed to be processed anywhere in the system. Also, DLT provides a discrete, fine

grained control of the system, such as timing control (i.e., when a processor should send a load fraction to another processor, based on delays), while this is not the main concern with the multi-commodity flow problem.

## 1.3 Scheduling Divisible Loads in The Resource Unaware Context

Almost all works reviewed so far bear the same fundamental assumption that the processor computation speed and the link communication delay are constant and known a priori to the scheduler which facilitates to generate an optimal, if not, a feasible schedule. This may not be the case in real life network based computing. Only one of the earlier works [22] digresses from this assumption. This work considers a time-varying nature of processor computation speeds and link communication delays in the form of a probabilistic model which is then used for optimal load scheduling in an average sense. However, in [22], the time varying nature of the processor computation speeds and link communication delays is still assumed to be known in advance.

On the other hand, in a recent work, D.Ghose et al. [23] investigates scheduling divisible loads in a "resource unaware environment", where the speed parameters are unknown in advance. In this case, before dispatching the load, the source processor where the initial load resides, should first detect the respective speeds of the link and the processor in the network. This is not a trivial task, since it would not be efficient to

spend too much time in estimating the speeds, while a relatively precise estimation of the link and the processor speeds is needed by the scheduler. D.Ghose et al. proposes a probing technique in their work to estimate the link and the processor speeds. In this technique, the source processor will send out a portion of the load, referred to as probing load, to other processors. These processors will process the fraction of probing load they receive, and they will send back the time stamps of when they start and finish transmission and when they finish processing to the source processor via short messages. Based on these feedbacks, the source processor is able to estimate the link and the processor speeds. This technique works efficiently in the sense that as the source processor "probes" the network (i.e., obtain the speed estimation of the link and the processor), a portion the real work has been done at the same time.

However, the scheduling algorithms proposed in [23] mainly cater to bus networks, where the source processor can directly send the probing load to any other processor. In networks where the probing load must be relayed from the source processor to other processors, such as linear networks or tree networks, a multiple ports assumption must hold for those algorithms to work properly. Further, while a probing technique is useful for networks with regular topologies, it may not be suitable for networks with arbitrary topology or the case where multiple sources exist. In such an environment, it is quite difficult to determine how to conduct the probing, as it is not easy to control the probing in an arbitrary topology and multiple sources may interfere with each other. Large overhead could be induced by probing, and it may suppress the gains.

## 1.4 Objectives and Organization of The Thesis

From the above review, we can see that there are a few gaps in the literature:

- In the resource unaware scheduling context, the existing strategies [23], which are based on a probing technique, are mainly designed for bus networks. It may not perform well for networks with other topologies, such as linear networks and tree networks. Further, the probing technique may not be useful for the case where the network bears an arbitrary graph topology, or/and multiple sources exist.
- For the problem of scheduling divisible loads in arbitrary networks, the performance of different spanning trees has not been systematically studied. It is not known which spanning tree offers the best trade-off between performance and complexity.
- Scheduling multi-source divisible loads on arbitrary networks has not been rigorously addressed.

### 1.4.1 General Focus, Contributions and Scope

The general focus of this thesis is to investigate the problem of scheduling divisible loads in resource unaware environment for more general cases. While achieving this objective, this thesis also addresses the problem of which spanning tree should be chosen for scheduling divisible load on arbitrary networks, and the problem of scheduling multi-source divisible loads on arbitrary networks. Specifically, we design and evalu-

ate resource unaware strategies for linear and multi-level tree networks. We compare the performance of different spanning tree routing strategies for scheduling divisible loads on arbitrary networks. Our findings suggest that, instead of the MST used in [51], the shortest path spanning tree (SPT) offers a better trade-off between complexity and performance. Further, to address the problem of multi-source scheduling on arbitrary networks, we propose a novel graph partition scheme (GP) to tackle the resource sharing issue. We then design and evaluate two strategies using the GP to schedule multi-source divisible loads on arbitrary networks.

The scope of the thesis is to design efficient strategies for scheduling divisible loads in different cases. We study the strategies analytically and also carry out rigorous simulation studies to validate these strategies under different network parameters. Implementation, however, is out of the scope of the thesis. The present study could enhance our understanding of scheduling divisible loads in resource unaware environments and also scheduling multi-source divisible loads. Further, the strategies proposed can be used to address real-life application when the network's speed parameters are unknown in advance and/or there are multiple sources .

The organization of this thesis is as follows. In Chapter 2, we extend the previous work [23] to the linear daisy chain networks. In this chapter and also the rest of the thesis, we adopt the "single port" and "with front-end" assumption, which is also most commonly assumed in the DLT literature. Under this assumption, two strategies, based on probing technique, are proposed to cater for the specific topology

of linear networks. Further, both strategies exhibit more control on the probing. This solves the potential overloading problem which may be caused by the strategies proposed in [23].

In Chapter 3, we address the problem of scheduling divisible loads in resource unaware multi-level tree networks. We first consider a static case, where the link and processor speeds are unknown in advance, but are constant. Therefore, a one time probe is sufficient to estimate the speed parameters. Then, we consider the dynamic case, where the link and processor speeds are unknown and may fluctuate. In this case, dynamic probing should be conducted to keep track of the varying speed parameters. Two strategies, also based on probing technique, are proposed to dispatching divisible loads under the above two cases, respectively. Further, communication congestion problem, which exists in the "single port" communication model, is explicitly considered in designing the scheduling strategies.

In Chapter 4, we discuss two important issues in scheduling a divisible load in an arbitrary network. Firstly, we argue the effectiveness of the probing technique in arbitrary networks and/or under multiple sources case. An alternative method is suggested. Secondly, we systematically study the performance of the different spanning trees by rigorous simulations. Which spanning tree should be chosen is suggested under different objectives.

In Chapter 5, we consider the most general problem - scheduling multi-source divisible loads in arbitrary networks. Starting from resource aware environments, two different cases - when each source has only one load and when each source has

an independent load inflow, are considered. A novel graph partitioning scheme is proposed to partition the network, and this scheme is used by two strategies, one catering for each case, to dispatch the multiple divisible loads. It also shows that the strategies proposed can be adapted to the resource unaware case. Queuing theory is applied to analyze the dynamic nature of the system and experiments are carried out to validate the usefulness and effectiveness of the present strategies. Certain interesting observations revealed by the experiments are carefully discussed.

Finally, in Chapter 6, we conclude this thesis and put forward some future recommendations in the context of this problem.

# Chapter 2

# Scheduling in Linear Networks

## 2.1 Problem Setting and Assumptions

In this chapter, we consider scheduling divisible loads on linear works. A linear network with processing nodes and communication links is shown in Figure 2.1. Each node or processor is equipped with a front-end processor which off-loads the communication responsibilities of that processor. This enables computation and communication to be carried out simultaneously. However, each processor is assumed to have only a single port for transmission, which means simultaneously transmitting or receiving is not allowed. Without loss of generality, each node is assumed to have adequate buffers to hold and process the data.

The total load to be scheduled and processed is initially stored on the root processor $P_1$. In this setting, we assume that the computing speeds of the nodes (except the root processor, where our scheduler that computes the required load distribution resides) and communication delays of the links are not known in advance. Further we neglect any start-up overheads and time to compute a (an optimal) load distribution.

Thus the objective is to minimize the total processing time of the entire load (time to complete processing from $t = 0$) under the above assumptions. We follow the work

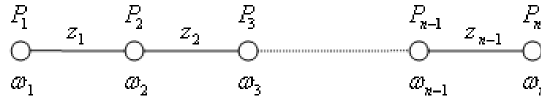


Figure 2.1: Linear Daisy Chain Network Architecture with $n$ processors and $(n-1)$ links

presented in [23]. However, the strategies presented in that work are predominantly useful for bus-like architectures wherein only one link exists to interconnect processors. Further in the linear network each probing-load (PL) has to percolate down the chain via $k$ links to reach processor $P_{k+1}$. Thus apart from seeking a load distribution that minimizes the processing time, additional issues such as the number of processors to be used in the chain[1], whether or not the same PL can be used owing to delays, etc, will play a vital role in influencing the overall performance. An illustrative example in Section 2.3 demonstrates the fact that the strategy *Probing and Selective Distribution* (PSD) [23][2] performs worse, if not, unsuitable for linear networks. This naturally motivates to design strategies that consider all the above issues that are critical and imperative to a linear network architecture.

Below we define some notations and terminology that will be used throughout this chapter.

(a). $L$ : the total load to be distributed and processed

[1]Otherwise waiting indefinitely for response from processors farther away owing to slow links, if any, may defeat the purpose.

[2]PSD strategy is the best performer for bus networks as shown in [23].

(b). $L_i$ : the load to be distributed and processed in the $i^{th}$ computation phase.

(c). $\alpha_j^i$ : the fraction of the load $L_i$ dispatched to the $j^{th}$ processor during the $i^{th}$ computation phase.

(d). $z_i$ : Ratio of the time taken to transmit a certain amount of data through the $i^{th}$ link to the time taken by a standard link.

(e). $w_i$ : Ratio of the time taken to compute a certain amount of data by the $i^{th}$ processor to the time taken by a standard processor[3]. $w_1 = K$, where $K$ is a constant.

(f). $T_{cm}$ : Communication intensity constant. It equals the time taken by a standard link to transmit a unit of the load. Thus, if $\alpha$ is the load to be carried by a link with communication speed parameter $z$, the communication delay incurred due to that link is given by $\alpha \cdot z \cdot T_{cm}$.

(g). $T_{cp}$ : Computation intensity constant. It equals the time taken by a standard node to compute a unit of the load. Thus, if $\alpha$ is the load assigned to a processor $P_i$ with computation speed parameter $w_i$, then the computation time incurred by $P_i$ is given by $\alpha \cdot w_i \cdot T_{cp}$.

(h). $\eta$ : Fraction of the total load used in the probing phase as a PL. Thus the size of the PL can be denoted as $\eta \cdot L$.

[3]A standard link (processor) can be any link (processor) that is referenced in the system.

## 2.2 Design of Resource Unaware Scheduling Strategies

Now we shall describe two efficient strategies that achieves our objective of minimizing the overall processing time of the entire load under unknown computation and communication speed parameters. Our strategies work in a two phase approach - a probing phase (PP) followed by a computation phase (CP). We further partition CP into several sub-phases. It may be noted that a PP and CP can overlap in time. Usually, probing phase will last several computation phases, as described in our strategies below. Before describing the design of strategies, first we present steps that are common to both the strategies now.

At the beginning of a probing phase, the root processor ($P_1$) will send a PL to its adjacent processor $P_2$. Because computation and communication can be overlapped, the root can start its computation when it transmits a PL to $P_2$. After receiving this PL, $P_2$ will start immediately to compute PL, while at the same time it sends a copy of PL to $P_3$. This process continues with every processor, allowing the PL to percolate down the chain. As a response to the processing of PL, each processor will record the communication completion time when it finishes receiving PL. It will send back this time to the root processor together with the processing completion time when it finishes computing PL through a processing task completion message (PTC). Since the messages are very small in size, their transmission time are negligible. Further, notice that as the communication completion time and processing completion time

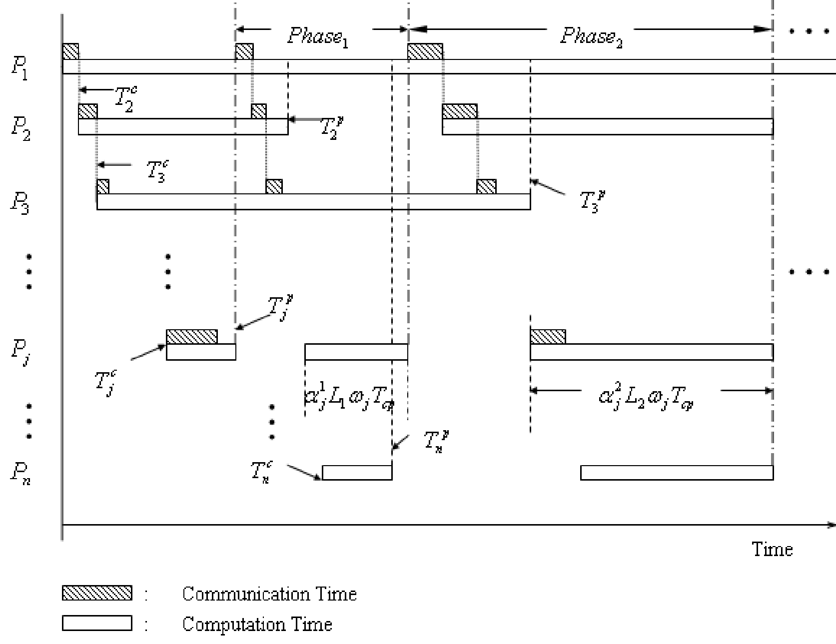


Figure 2.2: Timing Diagram For Early Start Strategy

are sent back through a single message, the number of messages are reduced to only a half compared to the work in [23]. We use $T_i^c$ and $T_i^p$ to denote the communication completion time and processing completion time of $P_i$, respectively. This process is shown in the timing diagram Figure 2.2.

From the timing diagram, we have,

$$z_i = \frac{T_{i+1}^c - T_i^c}{\eta L \cdot T_{cm}}, \qquad i = 1, 2, \ldots n-1, \qquad T_1^c = 0 \tag{2.1}$$

$$w_i = \frac{T_i^p - T_i^c}{\eta L \cdot T_{cp}}, \qquad i = 2, \ldots n, \qquad w_1 = K \tag{2.2}$$

It may be noted that from the set $\{T_i^c, i = 1, ..., n\}$, $T_j^c < T_{j+1}^c, \forall j = 1, ..., n-1$, and the set $\{T_i^p, i = 2, ..., n\}$, $T_j^p > T_j^c, \forall j = 2, ..., n$, but arrival of PTCs could be arbitrary in time. The processor with large subscript may return PTC early, if its processing speed is fast enough. Furthermore, it is impossible to predict when the last

PTC will arrive. Thus, in order to circumvent large waiting delay times owing to late responses from slow processors, it is wiser to engage processors that have returned their PTCs early. Thus computation of the load can be initiated on those processors that have rendered early responses. The implications of this idea are discussed in Section 2.3 and its impact will be demonstrated through our simulation studies.

Therefore, in our strategies, the scheduler divides the total load into two portions, the first portion is PL, and the remaining part is then further divided into several parts $(L_1, L_2, ..., L_m)$ and processing time of each part is referred to as one computation phase. We will also propose on the choice of the number of parts $(m)$ later. After the first several PTCs has been received, we will apply divisible load scheduling paradigm on the first part of load for those participating processors and the root. That is, we make use of the $k \leq n-1$ processors that respond earlier and the root processor to compute the first part of the load. These processors will receive an amount of load according to DLT paradigm and following an optimality principle [4] they stop computing at the same instant. This is denoted as the first computation phase (or simply $phase_1$). During $phase_1$, it is possible that other processors may respond to the root their processing of PL via PTCs one after another. Now, to accommodate these processors, we employ divisible load paradigm again for all the detected processors at the end of $phase_1$. Then $phase_2$ computation starts that includes the older processors (from $phase_1$) and some of the newly participating processors, if any. This recursive way of working continues until all the processors have been detected or the entire load has been taken up for processing by the currently active processors. Thus, it may be possible that the entire load can be completed with fewer set of processors without

waiting for all the processors to respond.

Since in a heterogeneous linear network, in general, a fastest processor need not be closer to the root processor and hence one may not expect that the first arriving PTC would come from this fastest processor. This is mainly due to the presence of communication links. Thus, it is more meaningful to define an *effective speed* of a processor to represent than its actual computation capability in this set-up. Consequently, we define, the effective computation speed parameter of a processor $P_j$ as,

$$\beta_j = w_j T_{cp} + \sum_{k=1}^{j-1} z_k T_{cm} \tag{2.3}$$

Thus, hereafter, when we refer to a processor as a fastest processor, we refer to its effective speed. Notice that the smaller the effective computation speed parameter ($\beta$), the faster the effective speed.

Based on how many processors are engaged in $phase_1$, our strategies can be classified into two types - Early Start Strategy (ESS) and Wait-and-Compute Strategy (WCS). Thus ESS initially starts with two processors in $phase_1$ and progresses recursively as explained above whereas, more than two processors can participate in $phase_1$ under WCS. We shall now present our analysis of these two strategies below.

### 2.2.1 Design and Analysis of Early Start Strategy

ESS only waits for the fastest processor returning its PTC, and then applies divisible load scheduling paradigm. However, the fastest processor may not be $P_2$, hence, in most cases, (2.1) may not be applicable. Actually, $z_i$ is unknown if $i \neq 2$, but load

distribution only considers $\sum_{i=1}^{j-1} z_i$, assuming that $P_j$ is the fastest node. This can be calculated as,

$$\sum_{i=1}^{j-1} z_i = \frac{T_j^c}{\eta L \cdot T_{cm}}, \qquad j = 2, \dots n \tag{2.4}$$

Using (2.2), $w_j$ can also be calculated. With known $w_j$ and $\sum_{i=1}^{j-1} z_i$, the first part of load can be dispatched according to:

$$\alpha_1^1 w_1 L_1' T_{cp} = \alpha_j^1 \cdot \sum_{i=1}^{j-1} z_i \cdot L_1' T_{cm} + \alpha_j^1 w_j L_1' T_{cp} \tag{2.5}$$

and together with the normalization condition

$$\alpha_1^1 + \alpha_j^1 = 1 \tag{2.6}$$

we can solve the above equations for the respective load fractions.

It should be noted that the exact load that has been dispatched during $phase_1$ is not $L_1$. This is due to the fact that before the fastest node sends back its PTC, the root processor would have started processing a part of $L_1$. If $P_j$ is the fastest processor, then the actual load that has been dispatched in $phase_1$ is given by,

$$L_1' = L_1 - T_j^p / w_1 T_{cp} \tag{2.7}$$

where,

$$T_j^p = \eta L (\sum_{i=1}^{j-1} z_i T_{cm} + w_j T_{cp}) \tag{2.8}$$

From equations (2.5) - (2.7), we obtain,

$$\alpha_1^1 = \frac{\sum_{i=1}^{j-1} z_i \cdot T_{cm} + w_j T_{cp}}{\sum_{i=1}^{j-1} z_i \cdot T_{cm} + w_j T_{cp} + w_1 T_{cp}} \tag{2.9}$$

Then the time consumed for processing in $phase_1$ is given by,

$$T_1 = \alpha_1^1 L_1' \cdot w_1 T_{cp}$$

$$= w_1(L_1 - T_j^p/w_1T_{cp})(\frac{\sum_{i=1}^{j-1} z_i \cdot T_{cm} + w_jT_{cp}}{\sum_{i=1}^{j-1} z_i \cdot T_{cm} + w_jT_{cp} + w_1T_{cp}})T_{cp} \tag{2.10}$$

as $w_1 = K$, if we define $T_{cm}/T_{cp} = \lambda$, $\sum_{i=1}^{j-1} z_i = z_{1j}$, (2.10) becomes

$$T_1 = K(L_1 - \eta L(\lambda z_{1j} + w_j)/K)(\frac{\lambda z_{1j} + w_j}{\lambda z_{1j} + w_j + K})T_{cp} \tag{2.11}$$

At the end of $phase_1$, more nodes (say $m \leq n - 2$) may have returned their PTCs. These processors together with $P_1$ will be engaged in the next phase. We denote those set of processors participating in $phase_2$ as $A_{phase_2}$.

$$A_{phase_2} = \{P_1, P_i \mid T_i^p = \eta L(z_{1i}T_{cm} + w_iT_{cp}) < T_1 + T_j^p \quad i = 2, \dots n\} \tag{2.12}$$

We sort the processors belonging to $A_{phase_2}$ in ascending order according to their subscripts and we re-index these processors for mathematical ease as, $\{P_1', P_2', \dots P_m'\}$ respectively. We define

$$\sum_{k=j}^{i-1} z_k = Z_{i'} \tag{2.13}$$

where $i, j$ are the original subscripts of $P_{i'},\ P_{(i-1)'}$

$Z_{i'}$ can be calculated by,

$$Z_{i'} = \frac{T_{i'}^c - T_{(i-1)'}^c}{\eta L \cdot T_{cm}}, \qquad i = 2, \dots m, \tag{2.14}$$

Now, recursive equations for $phase_2$, from the timing diagram (Figure 2.2), can be written as

$$\alpha_{i'}^2 w_{i'} T_{cp} = \alpha_{(i+1)'}^2 w_{(i+1)'} T_{cp} + Z_{(i+1)'} T_{cm} \sum_{k=(i+1)}^{m} \alpha_{k'}^2 \tag{2.15}$$

$$i = 1, \dots (m-1)$$

and the normalization condition is

$$\sum_{i=1}^{m} \alpha_{i'}^2 = 1 \tag{2.16}$$

Note that we have a total of $m$ equations with $m$ variables (from(2.15) and (2.16)) that can be solved to obtain the individual load fractions. Then the processing time consumed in $phase_2$ is given by,

$$T_2 = K\alpha_{1'}^2 L_2 T_{cp} \tag{2.17}$$

By and large, in $phase_i$, the processors involved in computation are,

$$A_{phase_i} = \{P_1, P_k \mid T_k^p < T_j^p + \sum_{n=1}^{i-1} T_n \quad k = 2, \ldots n\} \tag{2.18}$$

Similar to the above derivation, recursive equations can be generated for the set $A_{phase_i}$ to dispatch $L_i$. This process continues until either all the processors have been detected and used or the last load has been dispatched. Thus, supposing there are totally $m$ phases, the overall processing time of the entire load is

$$T_{overall} = T_j^p + T_1 + T_2 + \ldots + T_m \tag{2.19}$$

### 2.2.2 Design and Analysis of Wait-and-Compute Strategy

WCS will attempt to wait for more processors returning their PTCs, before starting $phase_1$. This is in the hope of accumulating more processing power to accommodate the load in the initial phase. This strategy particularly favors networks that have more fast processors. We will discuss the performance in our simulation study and elicit a number of observations. Thus, by virtue of this operation, the computation

phase of WCS will start later compared to ESS, however this does not mean ESS performs better than WCS at all times.

Assuming that WCS waits for $k$ processors, and $P_l$ is the last processor to issue its PTC in this $k$ processor set. Using set $A_{phase1}$ to denote these processors, we have $A_{phase1} = \{P_1\} \cup \{P_j \mid T_j^p < T_l^p,\ j = 2, \ldots, n\}$ and $|A_{phase1}| = k + 1$. Similar to ESS, $\forall P_i \in A_{phase1}$, we can obtain $Z_i$, which denotes the link speeds between $P_i$ and its adjacent processor $P_j$ $(i < j)$ in $A_{phase1}$, by (2.14)

To clarify this point, let $k = 2$, and $P_i$ and $P_j$ $(i < j)$ be the first two responded processors in the linear network except the root. After $P_1$ receives the PTCs from $P_i$ and $P_j$, it will trigger $phase_1$. The remaining load of $L_1$, which is equal to $(L_1 - \frac{\max(T_i^p, T_j^p)}{KT_{cp}})$, will be distributed among $P_1$, $P_i$ and $P_j$ following an optimal load distribution (within the current phase) [4]. To obtain the optimal load distribution, one can solve the recursive equations,

$$\alpha_1^1 KT_{cp} = (\alpha_i^1 + \alpha_j^1) Z_i T_{cm} + \alpha_i^1 \omega_i T_{cp} \tag{2.20}$$

$$\alpha_i^1 \omega_i T_{cp} = \alpha_j^1 Z_j T_{cm} + \alpha_j^1 \omega_j T_{cp} \tag{2.21}$$

together with

$$\alpha_1^1 + \alpha_i^1 + \alpha_j^1 = 1 \tag{2.22}$$

Solve (2.20) - (2.22) to obtain $\alpha_1^1$ as,

$$\alpha_1^1 = \frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)} \tag{2.23}$$

where,

$$\gamma = \frac{\omega_j + \lambda Z_j}{\omega_i}, \qquad \lambda = \frac{T_{cm}}{T_{cp}} \tag{2.24}$$

Then, $P_1$ can estimate the time consumed for processing in $phase_1$, which is given by,

$$T_1 = K(L_1 - \frac{\max(T_i^p, T_j^p)}{KT_{cp}})\frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)}T_{cp} \tag{2.25}$$

Therefore, $P_1$ knows when to trigger $phase_2$. As in ESS, the above process continues in all phases.

An interesting and noteworthy point at this juncture is as follows. ESS triggers an early start of the computation using the fastest processor in the network. However, despite receiving the PTC from this fastest processor, WCS adds some idle time before starting the computation. This later start approach by WCS may be compensated with the presence of additional fast processors whose PTC would have been just-in-time when the $Phase_1$ of ESS would have started. It would be interesting to see which strategy finishes $Phase_1$ earlier.

From the previous subsection, we can obtain the time when ESS finishes $phase_1$, denoted as $T^{ESS}_{phase_1}$, by (2.26)

$$T^{ESS}_{phase_1} = \theta_{ESS}(L_1 - T_j^p/KT_{cp}) + T_j^p \tag{2.26}$$

where $\theta_{ESS}$ is,

$$\theta_{ESS} = KT_{cp}\frac{\lambda z_{1j} + \omega_j}{\lambda z_{1j} + \omega_j + K} \tag{2.27}$$

On the other hand, when $k = 2$, from the discussion above, we can obtain the time when WCS finishes $phase_1$, denoted as $T^{WCS}_{phase_1}$, by (2.28),

$$T^{WCS}_{phase_1} = \theta_{WCS}(L_1 - T_i^p/KT_{cp}) + T_i^p \tag{2.28}$$

where $\theta_{WCS}$ is,

$$\theta_{WCS} = KT_{cp}\frac{\lambda(1+\gamma)Z_i + \gamma\omega_i}{\lambda(1+\gamma)Z_i + \gamma\omega_i + K(\gamma+1)} \tag{2.29}$$

Then, by equating the expressions (2.26) and (2.28), we can obtain the critical size for $L_1$, denoted as $L_1^c$, by (2.30).

$$L_1^c = (\frac{\theta_{ESS} T_j^p - \theta_{WCS} T_i^p}{K T_{cp}} + T_i^p - T_j^p)/(\theta_{ESS} - \theta_{WCS}) \tag{2.30}$$

When $L_1 = L_1^c$, WCS and ESS will have exactly the same overall performance. When $L_1 > L_1^c$, since WCS has more computation power in $phase_1$, WCS will finish $phase_1$ earlier. On the other hand, when $L_1 < L_1^c$, WCS does not have enough time to catch up with ESS, and hence ESS will finish $phase_1$ earlier.

Notice that the strategy that finishes $phase_1$ earlier does not guarantee it will finish processing the entire load earlier. However, processors which are engaged or respond in $phase_1$ are fast processors in the network. Finishing $phase_1$ earlier implies starting $phase_2$ earlier with most fast processors in the network, and hence will highly probably have a shorter overall processing time. This is verified by the simulation studies later. Further, in all the above analysis, we assume $k = 2$ for WCS. However, this does not mean $k = 2$ is the best choice. In the next section, we conduct experiments to identify the best possible value of $k$ with respect to certain information about the network.

## 2.3 Performance Evaluation and Discussions

In this section, we present simulation tests to validate and quantify the performance of ESS and WCS. We now present an illustrative example which shows that a recently proposed strategy, referred to as *Probing and Selective Distribution* (PSD) [23] for bus networks may perform worse, if not, inapplicable for linear networks architecture.

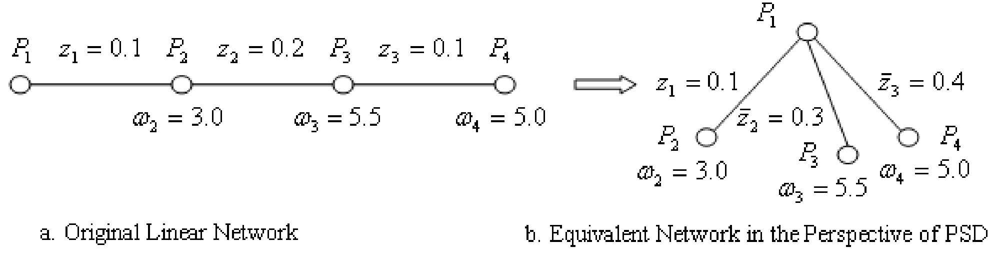


Figure 2.3: Network model for Example 1

**Example 1.** Consider a linear network shown in Figure 2.3.(a). Suppose $L = 60$, $T_{cp} = 2$, and $T_{cm} = 1$. If the PSD strategy is applied in the above network, because it is proposed for bus-like network, most probably it will treat the linear network as an equivalent network shown in 2.3.(b). According to the PSD strategy, initially (from t=0) $P_1$ will dispatch equal size load fractions (served as PLs) to $P_2$, $P_3$, and $P_4$, respectively, in a round robin fashion, until the first PTC is received by $P_1$. Suppose $PL = 1$. Then, $P_2$ will respond first at $t = 6.1$. At this time, $P_1$ has already dispatched 8 units of load to $P_2$ and $P_3$, 7 units of load to $P_4$, respectively, and now is sending another 1 unit of load to $P_4$. At t=6.4, $P_1$ finishes sending load to $P_4$ and then will continuously dispatch 1 unit of load to $P_2$ until it receives the second PTC responses from $P_4$ at $t = 10.8$. However, notice that there is a big gap between the first and the second PTC responses (10.8-6.4=4.4), and hence even before the time $t = 10.8$ all the 36 units of remaining load has been dispatched to $P_2$. As a result, $P_2$ totally received 44 units of load, and $P_3$ and $P_4$ only receive 8 units of load each. Obviously, $P_2$ get overloaded and there is no room for balancing.

The reason which leads to an overload is because of the continuous dispatching nature of the PSD strategy [4]. This mode of working makes full use of the link, which lets computation and communication highly overlapped and it is clearly a distinct advantage in bus-like networks, wherein the PTC responses are likely to come back consecutively. However, in a linear network set-up, this may have an adverse effect, as the PTCs are more common to be highly separated because of the presence of cumulative communication delays. Therefore, in linear network, it is wiser if load distribution can be controlled to wait for subsequent processors, when dispatching the load. This characteristic is captured in ESS and WCS, as only a portion of the load will be dispatched and processed in each phase.

Now, before presenting our experimental results, we further discuss some key assumptions on the choice of certain parameters and networks below.

The first issue is on our decision of parameter $\eta$. From the description of our strategy, it is evident that a larger $\eta$ indicates larger proportion of the load is used in probing, which means that except for the root, other processors will do more unimportant computation. In this sense, one may prefer a small $\eta$. On the other hand, the computation speeds of the nodes and the communication speeds of the links are not constants. Thus, a relatively large probing load is needed to precisely detect computation and communication parameters. In this sense, large $\eta$ is needed. Thus, to strike a balance between these two situations, we let parameter $\eta$ fall into the range $(0.03, 0.07)$, which is seen to be appropriate in our experiments.

The second issue is on partitioning the total load prior to the load distribution,

[4] PCD also has this style of working.

i.e., in deciding the number of phases of load distribution. Dividing the load into $N$ equal portions is a natural choice, but not wise. Because as time progresses, more and more processors will be engaged in the computation work. To make use of this property, the total load should also be divided in an increasing fashion. It may be noted that the number of processor engaged in each phase is expected to be a non-decreasing function. Together with the root, we set the ratio of the loads in successive phases as $(2^1 + 1) : (2^2 + 1) : \ldots$. Another point worth considering is that we let the root processor compute part of $L_1$ before the first PTC returns[5]. Furthermore, as we will see later, an initial choice of size of $L_1$ plays a crucial role in influencing the performance of ESS. For an $n$ processor system, the total load is partitioned as follows and a partition $L_k$ is to be distributed in $Phase_k$ among processors in set $A_k$, respectively.

$$L_i = (2^i + 1)(L - \eta L - L_1) / \sum_2^m (2^k + 1) \qquad i = 2, \ldots m$$

$$L_1 = \varepsilon * L$$

where, $\varepsilon \geq \eta$ and $m = \lceil log_2 n \rceil + 1$. We refer to the above distribution of the load portions as $\Pi$.

Let $L = 100$, communication parameter $z$ and computation parameter $w$ fall into the ranges $(0.2, 0.7)$ and $(2, 7)$ respectively, and the root processor has an average speed given by, $w_1 = K = 4.5$ and we let $T_{cm} = 1$ and $T_{cp} = 2$. We refer to the processing time for ESS, WCS, and conventional DLT strategy as T-ESS, T-WCS, and T-pureDLT, respectively. Note that the "pureDLT" strategy refers to an

[5]When the root is very fast, it may even finish computing $L_1$ before the first PTC. Then we let it to compute $L_2$ and re-index $L_2$ to $L'_1$,$\ldots L_k$ to $L'_{k-1}$, and so on.

approach wherein we wait for all the PTCs and then start to compute. This naturally serves as an upper bound on the time performance of our strategy. We denote the fraction of the fast processors and links in the network by $r_f$[6]. All the parameters in our simulation experiments are generated in a random fashion following a uniform distribution in their respective ranges. Each category of experiments is repeated 25 times and average values are reported for understanding the performance of the strategies. Tables 2.1 and 2.2 show the results of our experiments.

We will now present the results that demonstrate the influence of parameters $\eta$, $\varepsilon$ and $n$.

**Effect of $\eta$ :** Parameter $\eta$ fundamentally determines the size of PL. From our results, we observe that for a given network size, as $\eta$ increases the processing time increases, which is expected. However, the increase in the processing time of both ESS and WCS is less when compared to pureDLT strategy. This is due to the fact that the waiting time for the last PTC to arrive penalizes the performance significantly. The behavior with respect to $\eta$ remains the same regardless of the fraction of fast processors in the network ($r_f$). Also, for a given $\eta$, as we increase the network size, the processing time decreases. We will discuss this in detail when we present our results for the effect of network size. Now, the effect of $\eta$ on the network size can be observed as follows. The difference in the processing time between two different values of $\eta$ for pureDLT increases dramatically as network size increases. However, this difference almost remains the same for ESS and WCS. Thus each of the strategies

[6]We designate a processor and a link as fast when their speeds fall in $(0.2, 0.3)$ and $(2, 3)$, respectively.

seem to be robust in behavior with respect to the variation of $\eta$ and $n$. This behavior can also be observed for different $r_f$ values.

When comparing the performance of ESS with WCS with respect to $\eta$, we find that when $\eta$ is small, WCS shows a better performance than ESS, however, when $\eta$ is large, the performance of ESS and WCS are almost the same. This is because as $\eta$ grows, the range of PTC responses will increase correspondingly, which will naturally benefit ESS, as WCS has to wait longer to start computing.

**Effect of $\varepsilon$ :** Parameter $\varepsilon$ fundamentally determines the size of $L_1$ and here, ESS and WCS are our only concern. As shown in Tables 2.1 and 2.2, both ESS and WCS have a better performance for a smaller $\varepsilon$. This is because of the fact that a smaller $L_1$ implies a shorter $Phase_1$; and hence, those fast processors that returned their PTCs during $Phase_1$ will start their computation during $Phase_2$ much earlier than for a larger $L_1$ choice. We also observe that the influence of $\varepsilon$ to WCS is less significant than to ESS. This is because of the fact that as $\varepsilon$ grows, the increasing amount of load in $L_1$ will be calculated by only two processors, while in WCS more processors are expected to share it. Therefore, WCS shows a significantly better performance than ESS with a larger $\varepsilon$.

Actually, the sizes of $L_1, L_2, ...L_k$ all influence the performance of ESS and WCS, but the choice of $L_1$ plays a crucial role. In general, if we partition the load into several smaller portions, the performance of ESS and WCS will increase dramatically, and will be close to the lower bound where computation and communication speeds parameters are known in advance. However, in practice one cannot partition the load

Table 2.1: Experimental Results when $r_f = 0.75$

| $\varepsilon$ | n | $\eta = 0.03$ | | | $\eta = 0.07$ | | |
|---|---|---|---|---|---|---|---|
| | | T-ESS | T-WCS | T-pure | T-ESS | T-WCS | T-pure |
| 0.07 | 5 | 189.70 | 185.50 | 195.70 | 203.43 | 202.99 | 228.34 |
| | 8 | 173.73 | 169.29 | 182.32 | 190.06 | 188.51 | 221.19 |
| | 13 | 161.83 | 156.27 | 177.48 | 175.17 | 173.43 | 227.45 |
| | 20 | 157.01 | 153.60 | 182.86 | 170.99 | 171.31 | 242.57 |
| 0.12 | 5 | 194.77 | 186.59 | 195.70 | 207.20 | 203.49 | 228.34 |
| | 8 | 182.41 | 172.32 | 182.32 | 192.60 | 189.56 | 221.19 |
| | 13 | 171.99 | 159.49 | 177.48 | 182.02 | 174.18 | 227.45 |
| | 20 | 167.27 | 156.22 | 182.86 | 176.98 | 173.06 | 242.57 |

Table 2.2: Experimental Results when $r_f = 0.25$

| $\varepsilon$ | n | $\eta = 0.03$ | | | $\eta = 0.07$ | | |
|---|---|---|---|---|---|---|---|
| | | T-ESS | T-WCS | T-pure | T-ESS | T-WCS | T-pure |
| 0.07 | 5 | 244.65 | 241.87 | 249.71 | 278.81 | 282.13 | 283.45 |
| | 8 | 220.59 | 219.12 | 228.85 | 252.98 | 254.50 | 271.78 |
| | 13 | 204.40 | 202.77 | 220.78 | 227.69 | 227.31 | 270.37 |
| | 20 | 201.57 | 199.42 | 224.98 | 224.39 | 223.90 | 286.82 |
| 0.12 | 5 | 247.70 | 242.12 | 249.71 | 262.31 | 264.60 | 283.45 |
| | 8 | 226.93 | 219.75 | 228.84 | 244.85 | 244.44 | 271.78 |
| | 13 | 213.16 | 206.04 | 220.78 | 227.67 | 226.79 | 270.37 |
| | 20 | 211.94 | 204.80 | 249.71 | 225.44 | 224.91 | 286.82 |

into infinitesimally small size portions for processing.

From the following discussion we note that an equal division of load in all the phases[7] is not a wise choice for minimizing the processing time. From our analysis and experimental results it is evident that first few phases are crucial to the performance of ESS and WCS, in the sense of minimizing the processing time. ESS and WCS takes advantage of the response of fast processors in the network in the initial few phases by proportionally scaling the phase sizes, while an equal phase partitioning will render more load in the first phase thus increasing the processing time. Thus, ESS and WCS with distribution $\Pi$ assures a minimum idle time for processors that have responded in the previous phase while this idle time could be as large as a period of a phase under equal phase partitioning distribution.

However, an inverse effect of $\varepsilon$ can be observed when both $r_f$ and $n$ are small, and $\eta$ is big. In this case, as opposed to a common expectation that the finish time would increase as we tend to increase $\varepsilon$, our results show that the finish time decreases. An explanation to this anomalous behavior may be explained as follows. When $r_f$ and $n$ are small, there are limited number of fast processors in the network. Furthermore, as $\eta$ is large, the range of PTC responses is also large. In this situation, a small $\varepsilon$ implies that even when the processors finishes computing $L_1$, few processors will return their PTCs during $Phase_1$, while a large $\varepsilon$ gives more chance for these processors to respond during $Phase_1$, and hence this computation power can be utilized earlier. From this point, we can see that the fact that an unequal partitioning of load among the phases

[7] It may be noted that the distribution $\Pi$ and the equal load partitioning uses identical number of phases.

may also under-utilize processors. For instance, a smaller choice of $L_1$ and a larger choice of $L_2$ may have an adverse effect, as there may be fewer processors returning their PTCs during $Phase_1$, and hence, very few processors will be engaged in $Phase_2$.

**Effect of $r_f$ :** The ratio of fast processors in network is important in determining the performance of ESS and WCS. As we can see from Tables 2.1 and 2.2, when $r_f$ increases, overall finish time decreases, which is expected. Further, comparing the performances of ESS and WCS with respect to $r_f$, we find that WCS prefers a network with a large ratio of fast processors. This is because the fact that when a network has a large number of fast processors, there is higher chance that more fast processors will return their PTCs immediately after the fastest processor's response. In ESS, these fast processors will wait until the end of $Phase_1$ to be engaged in computation, while in WCS, this fast processor will be used much more earlier. Thus, when $r_f = 0.75$, we find WCS shows a significant improvement than ESS. However, when a network has only a few fast processors, WCS may have to wait more time for the response from another processor than ESS. Therefore, when $r_f = 0.25$, WCS and ESS exhibit approximately the same performance.

**Effect of network size ($n$) :** Network size is somewhat a crucial parameter to handle large scale data processing, especially when speeds of resources are unknown. This fact is captured in our simulation results. From Tables 2.1 and 2.2, we observe that as the network size increases, the finish time for both ESS and WCS decreases. However, as network size grows, the difference between ESS and pureDLT (and between WCS and pureDLT ) increases. This propensity can be explained by observing

that the range of PTCs stretches as $n$ increases. Therefore, the difference between the first PTC and the last PTC increases correspondingly, which penalizes pureDLT for its "greedy" waiting nature. Actually, after $n$ grows to some extent, increasing $n$ further will have an adverse effect on pureDLT, as its finish time starts to increase rather than decrease, as seen from the tables.

Having seen the influence of all the parameters, from the tables, we observe that the parameter $\eta$ is more sensitive in influencing the performance. This is due to the fact that the difference in processing time between ESS and pureDLT increases as $\eta$ increases. The response of a node (PTC), which depends on the size of the PL, decides when to start computing and this affects the processing time. This difference is observed to be more than the difference when network size or $\varepsilon$ is varied. Network size is a parameter that allows minimization of the processing time provided more fast processors are present in the network. While this depends on the underlying network, influence of $\eta$ remains independent, as its influence is critical to $Phase_1$ (start of the entire computation).

In all the above experiments, WCS only waits for two processors before starting $Phase_1$. Actually, WCS can be allowed to wait for more processors. We use WCS(i) to denote the algorithm that will wait on $i$ processors before starting $Phase_1$. Thus, ESS is WCS(1). For a given network, with fixed $\varepsilon$ and $\eta$, the finish time obtained in computing $L$ is a function of $i$ (we denote this finish time as $T_{WCS}(i)$). The performances for different $i$ with respect to $\varepsilon$ and $\eta$ can be seen from Figures 2.4 and 2.5, for different $r_f$ distribution.

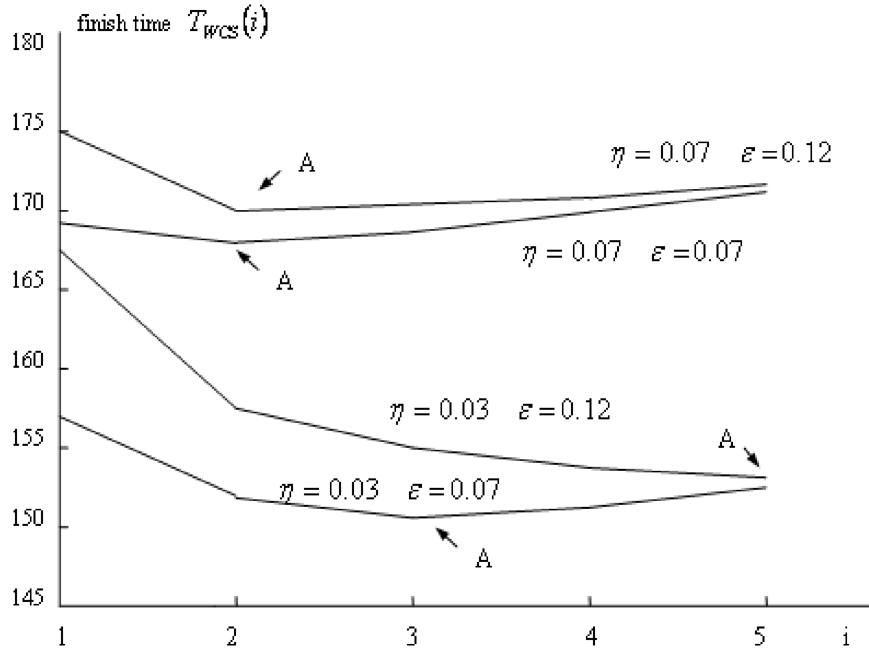


Figure 2.4: Figure of $T_{WCS}(i)$ for Different $\varepsilon$ and $\eta$ when $n = 15$ $r_f = 0.75$

In general, for a given $\eta$ and $\varepsilon$ value, as $i$ increases, $T_{WCS}(i)$ is observed to decline first, reaching a minimum point (shown as point $A$ in the figures) and then increases (See Figure 2.4 and 2.5). However, as we decrease $\eta$, the entire curve $T_{WCS}(i)$ shifts down, which means less finish time is achieved. Further, the minimum point $A$ starts to shift to right, which means that minimum finish time will be obtained for larger $i$ values. This observation could be useful while implementing the strategies. Thus, if we choose a small $\eta$, it would be appropriate to choose a relatively larger $i$, say 3 or more, to minimize finish time. $L_1$ is another factor that affects the shape of the curve $T_{WCS}(i)$. When $L_1$ decreases, the curve $T_{WCS}(i)$ shifts down, and its minimum point shifts to left, which means minimum finish time is obtained for a small $i$. Furthermore, as shown in Figures 2.1 and 2.2, $r_f$ also affects $T_{WCS}(i)$. Larger $r_f$ naturally benefits

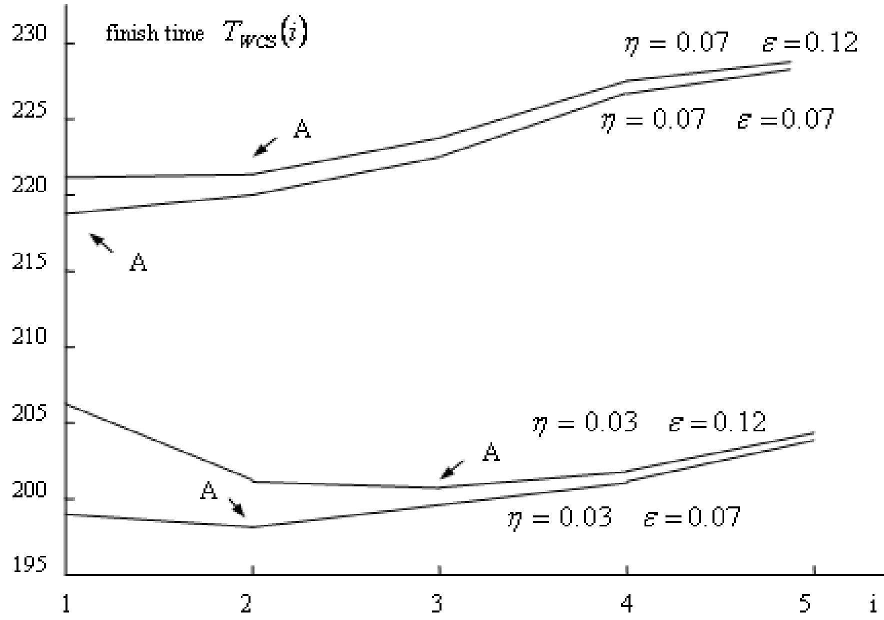


Figure 2.5: Figure of $T_{WCS}(i)$ for Different $\varepsilon$ and $\eta$ when $n = 15$ $r_f = 0.25$

WCS as discussed above, which drives the minimum point of $T_{WCS}(i)$ to right.

As we have seen above, the value of $i$ which achieves the minimum finish time (point $A$ in the figures) is affected by the combined effect of $\eta$, $\varepsilon$, $r_f$ and $n$, where $\eta$ and $\varepsilon$ are determined by the strategy, while $r_f$ and $n$ are characteristics of the network. $n$ is usually a known parameter. Hence, if we have some prior knowledge about $r_f$, we can choose a suitable $i$ according to the value of $\eta$ and $\varepsilon$. The following simulations reveal the most probable best value of $i$ with respect to certain $\eta$, $\varepsilon$, $r_f$ and $n$.

For a network with 15 processors, we first set $\eta = 0.03$ and $\varepsilon = 0.12$, and vary the value of $r_f$ to 0.2, 0.5, 0.8, respectively. Each category of experiments is repeated 25 times. We find that when $r_f = 0.2$, most of the times (19/25) the minimum finish

time is obtained at $i = 2$. When $r_f$ increase to 0.5, to obtain the minimum finish time, $i$ should increase to 3 (17/25), and when $r_f = 0.8$, most of the times the minimum finish time is obtained at $i = 4$ (13/25) or $i = 5$ (9/25). Then we adjust $\eta$ to 0.07, $\epsilon$ to 0.07 and redo the experiment. We find when $r_f = 0.2$, the best value of $i$ equals to 1 (15/25), and when $r_f = 0.5$ or $r_f = 0.8$, most of times (around 21/25) $i = 2$ is the best choice.

# Chapter 3

# Scheduling in Multi-Level tree networks

## 3.1 Problem Definition, Assumptions and Remarks

In this chapter, we consider the problem of scheduling divisible loads in resource unaware general multi-level tree networks. Compared to linear networks, tree networks have several unique characteristics. In tree networks the root processor can directly communicate with several processors, unlike in linear networks, where only one processor directly connects to the root. Further, as the tree networks are normally larger (or "fatter") than linear networks, when probing the tree network, the processors' response times tend to be much closer to each other. Also, in tree networks, there exists more than one route, communication congestion problem should be addressed explicitly, if processors are not equipped with multiple independent ports. These issues will be considered in this chapter when designing scheduling strategies. Below, we will introduce the problem setting.

A tree network with processing nodes/processors and communication links is shown in Figure 3.1. We assume the initial load to be processed is stored in the root processor ($P_0$), where the scheduler resides. For simplicity, we allow $P_0$ as a

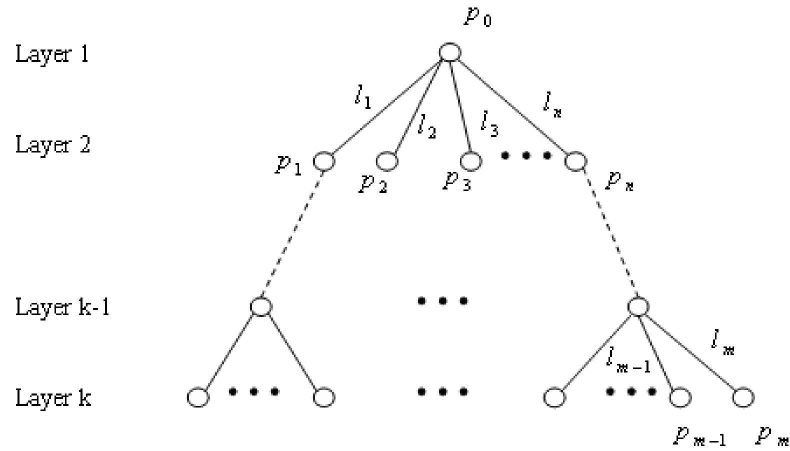


Figure 3.1: General tree network

dispatcher and we assume that it does not participate in the computation process. Similarly, every processor is assumed to be equipped with a front-end processor and has only a single independent communication port. We consider two distinct cases of practical interest - *Static Network Parameter* (SNP) case and *Dynamic Network Parameter* (DNP) case. In SNP case, the processor speeds and link speeds are assumed to be constant which is also common to dedicated networks, while in the DNP case the mild fluctuation of link and processor speeds is considered. In both cases (SNP and DNP) we assume that no prior knowledge about computation and communication speeds is available in advance. It should be noted that, in the DNP case, even when one processor's computation speed becomes extremely slow, the routing function of that processor may not be affected, as communication responsibilities are off-loaded to the front-end processor.

Without loss of generality we make the following assumptions which apply to both

cases. First, except for computation and communication speeds, $P_0$ is aware of other information about the tree, such as the number of children, and routing information. Secondly, each processor can store and process any amount of data. Thirdly, we neglect the time needed by the scheduler to compute a suitable load distribution and . Finally, as in the DLT literature, we continue to assume a linear cost model, but it can be easily extended to an affine model.

Our objective is to minimize the total processing time of the entire load under the above model. We design two strategies, referred to as *Static Load Distribution (SLD) Strategy* and *Dynamic Load Distribution (DLD) Strategy*, to schedule divisible loads under the SNP and DNP cases, respectively. However, before we present the adaptive strategies, we present a general discussion on certain common principles that are followed in the design of our strategies for both cases.

Since the speeds of the processors and links are unknown, we attempt to use a fraction of the processing load to probe the speeds, as in the previous chapter. However, since the underlying topology is a multi-level tree network of arbitrary depth and width, we carry out this probing process in a different manner so as to make this phase more time efficient. At the beginning of probing phase[1] (i.e., $t = 0$), $P_0$ partitions a portion of the total load $L$, say $\eta L$, equally into $n$ parts, and each part is referred to as one probing load (PL). Then $P_0$ dispatches the $n$ PLs to its children, one by one. Because computation and communication can be overlapped, each processor, after receiving its PL, will start to compute immediately while, at the same time, it will send a copy of the PL to its children, if it has any. Hence, PLs start

[1]The probing phase is defined in Chapter 2

percolating down through the tree. Each processor will report to $P_0$ the information about receiving and processing PLs, and these information will be collected by $P_0$ to estimate the respective processor and link speeds. To avoid long waiting times (delays) $P_0$ will start considering the inclusion of processors into the computation process progressively in phases as and when the processors respond. The rest of the load will be dynamically dispatched to processors based on the current information on processor and link speeds. This stage is referred to as the Computation Phase (CP) and it can be further divided into several sub-phases, denoted as $Phase_1, Phase_2, \ldots$, respectively. Each computation phase will consume a portion of the load. It should be noted that a PP and CP can overlap in time. Usually, the probing phase will last over several computation phases.

Below we give some notations used in this chapter. We define $L$ to be the total load to be processed and $L_q^p$ to be the load which $P_0$ will dispatch at the beginning of $Phase_q$. The size of $L_1^p$ is set by our algorithm initially, where as $L_2^p, L_3^p, \ldots$ are determined dynamically. We use $\alpha_i$ to denote the fraction of the load dispatched to $P_i$ at one batch of load distribution. $w_i, z_i, T_{cm}$ and $T_{cp}$ are defined as in the previous chapter. We also define $\eta$ as the fraction of $L$ used to detect $z_i$ and $w_i$, and $\mu$ as the final threshold load, that at the beginning of each phase, if $P_0$ finds the remaining load to be less than $\mu L$, it will dispatch the entire remaining load to all available processors. Finally, for ease of presentation, we define that $\pi_{ij}$ as the path from $P_i$ to $P_j$ (thus, $\pi_{0i}$ is the path from $P_0$ to $P_i$.); $|P_i|$ is the subtree rooted at $P_i$; $T_q^d$ is the duration of $Phase_q$; and finally $T_q^f$ is the finish time of $Phase_q$.

## 3.2 Static Network Parameter (SNP) Case

In SNP case, computation and communication speeds are assumed to be constant, and hence a single-time estimation of speed parameters would suffice. The SLD strategy starts the probing phase at time $t = 0$. As PLs reach the respective processors down the tree, each processor records three important time instants of relevance to it: (i) the time it starts to receive PL (ii) the time it finishes receiving PL and (iii) the time it finishes computing PL. We use $T_i^{cs}$, $T_i^{cf}$, and $T_i^p$ to denote the three time instants of $P_i$, respectively. Each processor will send back $T_i^{cs}$ and $T_i^{cf}$ through a message, referred to as communication task completion message (CTC) to $P_0$, when it finishes receiving PL, and it will send back $T_i^p$ through a message, referred to as processing task completion message (PTC), to $P_0$, when it finishes computing PL. This process is shown in the timing diagram of Figure 3.2 [2]. Furthermore, as CTC and PTC are short messages, the transmission time becomes negligible.

From the timing diagram, we have,

$$z_i = \frac{n(T_i^{cf} - T_i^{cs})}{\eta L T_{cm}}, \qquad i = 1, 2, \ldots m, \tag{3.1}$$

$$w_i = \frac{n(T_i^{p} - T_i^{cf})}{\eta L T_{cp}}, \qquad i = 1, 2, \ldots m, \tag{3.2}$$

It may be noted that a processor always returns its CTC later than its parent, so when an $i^{th}$ processor returns CTC, all the processors in $\pi_{0i}$ have returned their CTCs. However, the arrival of PTCs could be arbitrary in time, which depends on the combined effect of processing speed and cumulative communication delay. A

[2] $P_i$ in Figure 3.2 indicates an arbitrary processor, other than the children of $P_0$, in the multilevel tree

Figure 3.2: Time Diagram For SLD Strategy

relatively slow processor (having large $w$) may return its PTC early, if its cumulative communication delay ($\sum_{k\in\pi_{0i}} z_k$) is small. Therefore, as in Chapter 2, we define an effective speed of a processor to represent its computation capability.

$$\beta_j = w_j T_{cp} + \sum_{k\in\pi_{0j}} z_k T_{cm} \tag{3.3}$$

where, a lower value of $\beta$ means the processor is faster. Notice that a processor with the fastest effective speed is not guaranteed to return its PTC first, because of the PL distribution sequence. However, the first processor which returns its PTC is certain to have a very fast effective computation speed. Suppose $P_i$ returns its PTC first (e.g., at time $t = T_i^p$). Then, in order to circumvent large waiting delay times owing to late responses from slow processors, the SLD strategy will send $L_1^p$ amount of load

to $P_i$ immediately after receiving the PTC response. In the SLD strategy the time $P_i$ needs to compute $L_1^p$ determines the duration of $Phase_1$, denoted as $T_1^d$, and $T_1^d$ can be calculated as,

$$T_1^d = L_1^p \beta_i = L_1^p (w_i T_{cp} + \sum_{k \in \pi_{0i}} z_k T_{cm}) \tag{3.4}$$

We use $T_1^f$ to denote the end time of $Phase_1$, and we have $T_1^f = T_i^p + T_1^d$. The SLD strategy will use $T_1^f$ to control all the processors which have responded through their PTCs during $Phase_1$. According to the optimality principle, the amount of load being dispatched to relative processors should be proportional to the processors' computation speed and communication delay so that all participating processors stop computing at the same time instant. The load distribution satisfying this requirement is referred to as optimal load distribution. However, in a "blind environment" wherein speeds of the processors and links are unknown, we cannot directly apply traditional DLT equations to calculate an optimal load distribution, but we can meet the optimality principle by assigning proportional amount of load to the processors that have responded during $Phase_1$ so that all participating processors stop computing at $T_1^f$. Further, $T_1^f$ is also a natural starting point for the next phase, because all the participating processors finish their jobs at that instant and are ready for the new load. Below, we discuss this idea in detail.

Suppose $P_j$ is a child of $P_0$ in $\pi_{0i}$ that passed $L_1^p$ to $P_i$ through $P_j$ and suppose $P_0$ takes $T_{delay}$ units of time to perform this operation. Then we have,

$$T_{delay} = L_1^p z_j T_{cm} \tag{3.5}$$

At time $t = T_i^p + T_{delay1}$, $P_0$ finishes passing $L_1^p$ to $P_j$ and is ready to respond to

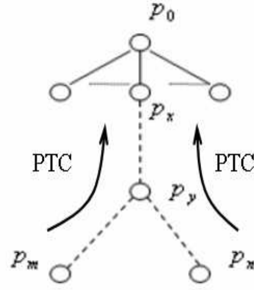


Figure 3.3: Demonstration of Congestion

another PTC, if one exists. If at $\hat{t}$, with $T_i^p + T_{delay1} \leq \hat{t} \leq T_1^f$, $P_0$ receives a PTC from $P_k$ and is also idle, then it will send an amount of load $L_k$ to $P_k$. The size of $L_k$ is calculated as,

$$L_k = \frac{T_1^f - \hat{t}}{\beta_k} = \frac{T_1^f - \hat{t}}{w_k T_{cp} + \sum_{k \in \pi_{0k}} z_k T_{cm}} \tag{3.6}$$

Therefore, $P_k$ will stop computing $L_k$ at time $T_1^f$.

However, in a tree topology, communication congestion may happen if our scheduler is not well designed. Consider the following situation that demonstrates the effect of congestion as shown in Figure 3.3. Two processors, $P_m$ and $P_n$, belonging to the same branch rooted at $P_x$, return their PTCs one by one very closely, and $P_0$ responds to them consecutively. Communication congestion may happen at $P_x$, because when it is passing down the load to $P_m$, it cannot receive load intended for $P_n$ from $P_0$ simultaneously. Actually, such congestions may happen at any processor in $\pi_{0x}$ which is common to both $\pi_{0m}$ and $\pi_{0n}$.

As we have seen above, we have to design our strategy prudently to avoid congestions without sacrificing performance. The SLD strategy clearly strikes a balance in

avoiding congestions to a large extent. The tree shown in Figure 3.1 can be considered to be composed of $P_0, |P_1|, |P_2|, \ldots, |P_n|$, while $|P_j|$ denotes a subtree rooted at processor $P_j$. The respective link speeds between $P_0$ and $n$ sub-trees are, $z_i, i = 1, 2, \ldots, n$. Now, we can group all PTCs into $n$ sets, depending on the $|P_i|$ from which those PTCs come.

Thus, to avoid congestion, after $P_0$ responds to the first PTC from processor $P_i$ (suppose it is from the set $|P_j|$), it should not send another load to any processor in the same sub-tree until $P_i$ finishes its communication. Further, let us suppose it takes $t_i$ units of time for $P_i$ to receive $L_1^p$. Then we have,

$$t_i = L_1^p \sum_{k \in \pi_{0i}} z_k T_{cm} \tag{3.7}$$

Therefore, $P_0$ will "shadow" the set $|P_j|$ for $t_i$ unit of time from the instant when $P_0$ starts sending $L_1^p$ to $P_j$. During this time, PTCs coming from the shadowed set will only be buffered at $P_0$ but cannot be responded to. When $P_0$ becomes idle (e.g., at time $t = T_i^p + T_{delay1}$), it will check whether it has received a PTC from any of the unshadowed sets. Three different cases may happen: ($i$) There are no PTCs from unshadowed sets ($ii$) $P_0$ receives only one PTC from unshadowed sets ($iii$) $P_0$ receives more than one PTC from unshadowed sets.

Case ($i$): Before $T_1^f$, $P_0$ will remain idle until it receives another PTC from the unshadowed set. Suppose $P_0$ receives $P_x$'s PTC from the unshadowed set $|P_y|$ before $T_1^f$, it will send $P_x$ a proportional amount of load $L_x$, which can be calculated by Eqn.(3.6), and at the same time, it will shadow $|P_y|$ for a certain duration in time. If $P_0$ does not receive any unshadowed PTCs until $T_1^f$, it will trigger $Phase_2$ at $T_1^f$. We

will discuss how $Phase_2$ will be performed later in this section. It should be noted that any sets of processors that were shadowed earlier will become available as time progresses.

Case (*ii*): $P_0$ will immediately send a proportional amount of load to the processor which returns the PTC, and as in Case (*i*), the corresponding set will be shadowed for some time.

Case (*iii*): In this case, the SLD strategy uses optimal sequence[3], given in [31], to decide which processor(s) should be engaged in computation first. Among all the unshadowed sets which generate at least one PTC, we choose the set, say $|P_j|$, that has the fastest communication link $z_j$ to $P_0$. If more than one processor in the set have returned their PTCs, an equivalent computation power of these processors is calculated (the concept of equivalent computation power is given in [4] and will be described later in the section) and $P_0$ will send loads to these processors in a single batch. Furthermore, the longest communication delay, which can be derived by $max\{T_1^f - t - L_i w_i \mid P_i \in |P_j| \ and\ has\ returned\ PTC\}$, where $t$ denotes the current time, is taken as the time for which this set is being shadowed.

Processor $P_0$ executes the above procedure till time $T_1^f$. Notice that even if a processor returns its PTC to $P_0$ at a time very close to $T_1^f$, $P_0$ will still send some load to this processor, if it is idle, and the performance can certainly improve from this dispatch.

At time $t = T_1^f$, all those processors that have participated in $Phase_1$ would have

[3] A sequence in which the load distribution follows the same order in which link speeds decrease.

accomplished their jobs and become available again. Now, $P_0$ will trigger $Phase_2$. In our SLD strategy, we first construct a "virtual tree" using all those available processors, based on the following rules:

1. All those currently available processors are kept as virtual tree nodes.
2. A processor in Level 2 is retained as a *virtual tree node*, if at least one PTC returns from its sub-tree. However, if the processor itself has not returned a PTC, its computation speed will be set to infinity (extremely slow processor).
3. From bottom to top, a processor where two or more virtual tree nodes converge is marked as virtual tree node, and its computation speed is set to infinity, if it has not returned the PTC.
4. In the virtual tree, the link speed between a parent and its child is the cumulative link speed between these two nodes in the original tree.

Figure 3.4 clarifies these four rules. Now, $P_0$ will dispatch $L_2^p$ to the "virtual tree", based on an optimal load distribution. As more processors are engaged at the beginning of $Phase_2$, more loads should be consumed by the "virtual tree". Thus, $L_2^p$ can be easily decided as,

$$L_2^p = kL_1^p \tag{3.8}$$

where $k$ is the number of available processors. To compute an optimal load distribution for each processor, $P_0$ can use a similar RAOLD-OS [51] scheduling strategy based on optimal sequencing. It will calculate an "equivalent processor" for each subtree, and shrink the "virtual tree" into a single-level tree . The process is described below.

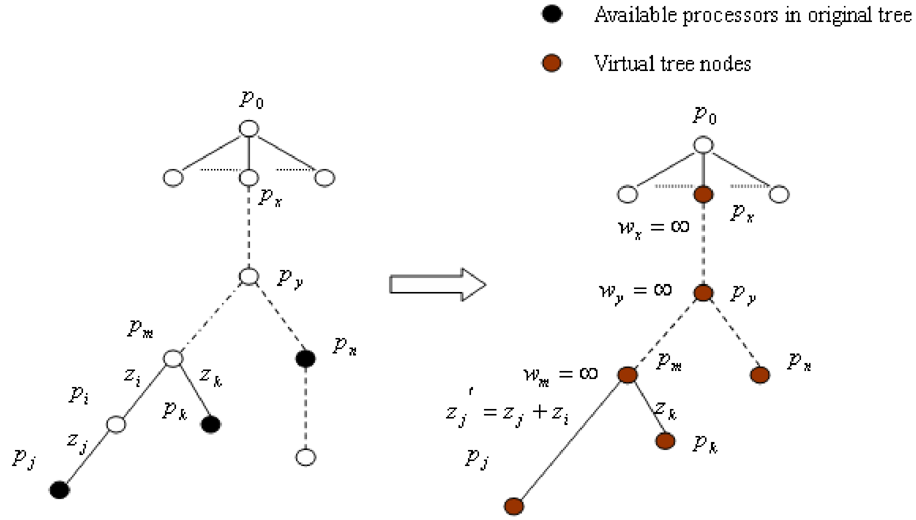


Figure 3.4: Virtual Tree Construction

A single-level tree, following an optimal sequence ($z_i \leq z_{i+1}, i = 2, 3, \ldots, m$) is shown in Figure 3.5(a). For an optimal load distribution $\alpha_1, \alpha_2, ..., \alpha_m$, the corresponding recursive load distribution equations are,

$$\alpha_k w_k T_{cp} = \alpha_{k+1} z_{k+1} T_{cm} + \alpha_{k+1} w_{k+1} T_{cp},\ k = 1, 2, ..., m \tag{3.9}$$

with the normalizing equation,

$$\sum_{j=1}^{m} \alpha_j = 1 \tag{3.10}$$

Thus, we can compute the value of $\alpha_1$ from Eqns.(3.9)–(3.10) as,

$$\alpha_1 = \frac{\prod_{j=1}^{m} f_j}{1 + \sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{3.11}$$

where,

$$f_k = \frac{w_k + z_k \frac{T_{cm}}{T_{cp}}}{w_{k-1}} \tag{3.12}$$

In the sense of computation power, this single-level tree is the same as a processor $P_1^{'}$

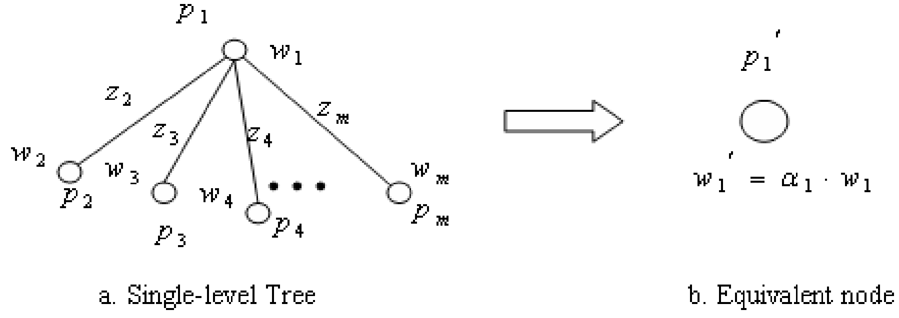


Figure 3.5: Equivalent Processor for Single-level Tree

with $w_1^{'} = \alpha_1 w_1$, as shown in Figure 3.5(b). For a given amount of load L, both of them need $\alpha_1 L w_1 T_{cp}$ units of time to complete processing of the load.

The above process is carried out recursively on the "virtual tree" from the bottom layer to the second layer, until the "virtual tree" shrinks into an equivalent single-level tree. Now, an optimal load distribution for the equivalent single-level tree can be easily computed, by solving relative single-level tree DLT equations, as shown in Eqns.(3.9)–(3.10) [4]. Then, by inflating the equivalent processor to the original sub-tree, the relative load fraction for each available processor can be computed. Respective load fractions intended for $L_2^p$ will be percolated down through the tree to available processors.

As in $Phase_1$, $P_0$ will record the finish time, denoted as $T_2^f$, for the "virtual tree" to complete $L_2^p$, and set this time as the end of $Phase_2$ (also the beginning of $Phase_3$). Meanwhile, $P_0$ will also set relative shadow time for each sub-tree which is engaged in computation. During $Phase_2$, $P_0$ responds to new arrival of PTCs using the same

[4] As we do not include $P_0$ in computation, $\alpha_0$ should not appear in the DLT equations.

strategy as in $Phase_1$. At $T_2^f$, $P_0$ will construct a new "virtual tree" for available processors, and trigger $Phase_3$. This process continues until all processors have been discovered or at the start of $Phase_i$, the remaining load is smaller than either $\mu L$ or the expected $L_i^p$. Then, $P_0$ will dispatch the whole remaining load to the currently available processors. The entire algorithm described above is shown in a flow-chart in Figure 3.6.

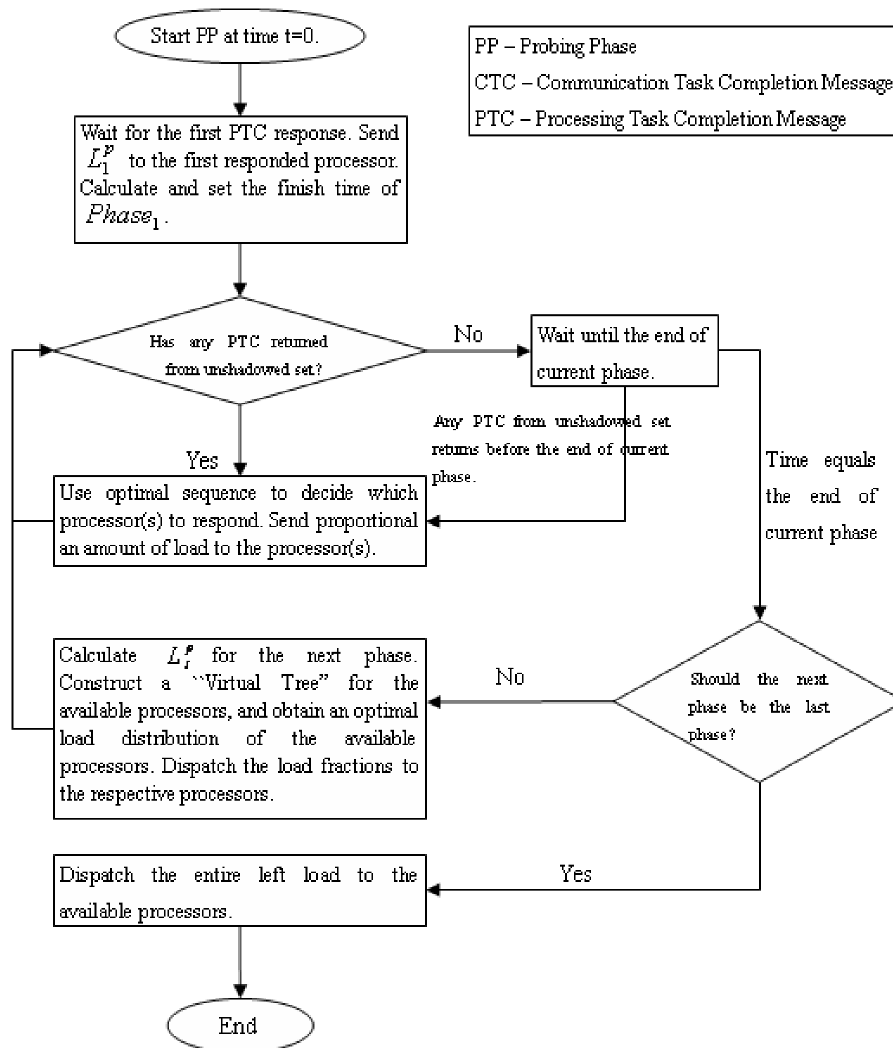


Figure 3.6: Flow Chart for SLD Strategy

## 3.3 Dynamic Network Parameter (DNP) Case

In DNP case, computation and communication speeds may vary with time, which would make the SLD strategy unsuitable for the following reasons: (*i*) SLD strategy adopts only a single time probing to detect computation and communication speeds, which cannot accommodate dynamical changes of computation and communication speeds. (*ii*) SLD strategy sets artificial time bound for each phase (e.g., $T_1^f, T_2^f, ...$), and it makes each processor that has responded to catch up with this bound. However, if $w$ and $z$ fluctuate, those processors may not finish their job at the estimated time. (*iii*) $P_0$ has to track every load fraction which is sent out to be processed by each processor, and hence a processor's computation speed may become extremely slow and, in that case, $P_0$ may need to resend this load to another processor for processing.

To address these issues, which arise in the DNP case, we develop a dynamical version of SLD strategy, which is referred to as *Dynamic Load Distribution (DLD)* strategy. In DLD strategy, a processor will return a PTC each time it finishes computing a given amount of load. This is different from the SLD strategy, wherein a PTC is sent only once by each processor. Also, in the DLD strategy, a CTC will be generated every time a communication ends, whereas in the SLD strategy a CTC is generated only in the probing phase. Furthermore, in DLD strategy, for the processor which accomplishes its job much earlier than the estimated finish time, $P_0$ will incorporate this actual finish time in computation again, and hence a processor may receive more than one load fraction during each phase. However, for the processor which accomplishes its job much later than the estimated finish time, $P_0$ will incorpo-

rate this finish time after receiving its PTC or it may resend its load fraction to other processors if $P_0$ does not receive its PTC until the beginning of the last phase. Now, we shall describe the DLD strategy in a step by step fashion for ease of understanding.

Similar to the SLD strategy, at the beginning of the probing phase, $P_0$ will dispatch $n$ PLs, each $\eta L/n$ in size, to its $n$ children respectively, and the $n$ children will pass down their PLs to their children. Each processor will send back a CTC to the $P_0$ processor, when it finishes receiving the PL and it will send back a PTC, when it finishes computing the PL. After dispatching PLs to its children, $P_0$ will wait for the first PTC. Suppose the first PTC comes from $P_i$, which belongs to a sub-tree set $|P_j|$, $P_0$ will send $L_1^p$ to $P_i$ and compute the duration of $Phase_1$ using Eqn. (3.4), based on the current value of $z$ and $w$, and then derive the start time of the next phase $T_1^f = T_i^p + T_1^d$. During the transmission of $L_1^p$, each processor in $\pi_{0i}$ will send back a CTC after it finishes receiving $L_1^p$ from its parents, and through these CTCs, $P_0$ can update the link speeds in $\pi_{0i}$.

To avoid congestion, as discussed above, we will shadow $|P_j|$ until $P_i$ receives its load. However, since we do not know the current value of $z \in \pi_{0i}$, we cannot directly apply Eqn. (3.7) to compute the shadow time and hence $P_0$ will shadow $|P_j|$ until it receives CTC from $P_i$. During the shadow time, all PTCs from this set will only be buffered at $P_0$ but will not be responded to. Furthermore, as $P_i$'s computation speed may become extremely slow when it starts to compute $L_1^p$, to avoid prohibitively long waiting times for $P_i$, we store a label in $P_0$ indicating that $L_1^p$ is being processed by $P_i$. Later, when $P_i$ finishes computing, it will again send back a PTC. According to

this PTC, $P_0$ will update $w_i$, and it will remove the corresponding label, indicating that the computation of $L_1^p$ has been completed. However, if $P_0$ does not receive $P_i$'s PTC from that time until the beginning of the last phase, $P_0$ will dispatch $L_1^p$ again, together with the remaining load to other processors.

Unlike in the SNP case, we would introduce a threshold $\zeta \in [0, 1)$ in the DNP case. Before $T_1^f - T_1^d\zeta$, when $P_0$ becomes idle and it receives another PTC or PTCs from unshadowed sets, the similar process as in SLD strategy will be adopted to respond to the PTC(s). However, $P_0$ will store respective labels indicating that this load fraction is now being processed, whenever a proportional load fraction is sent to a processor or an equivalent processor, and further it will shadow relative sub-tree sets until all the communication within the sub-tree comes to a halt.

The following points may be noted here: $(i)$ Whenever a processor finishes a computation or a communication, it will send back a PTC or a CTC, which are used by $P_0$ to dynamically update relative values of $w$ and $z$. When using Eqn. (3.6) to calculate load size, we should use the most recently updated $w$ and $z$ value. $(ii)$ $P_0$ may receive PTCs both from processors completing PL computation and those completing the actual assigned load computation. For the PTCs, which are completing the assigned load computation, corresponding labels stored in $P_0$ should be removed. $(iii)$ The processors that return their PTCs within the range $(T_1^f - T_1^d\zeta, T_1^f)$ will not receive another load fraction from $P_0$, but their PTCs will be used to estimate their recent computation speed and to remove the corresponding labels in $P_0$.

Time $T_1^f$ is the estimated finish time for processors which are engaged in com-

putation during $Phase_1$. Thus, for a given processor $P_k$, if its effective speed $\beta_k$ remains the same, it will stop computation at $T_1^f$, and become available. Further, those processors which send back their PTCs a little earlier than expected but within the interval $(T_1^f - T_1^d\zeta, T_1^f)$, will not receive another job, and hence remain available at $T_1^f$. Therefore, $P_0$ will trigger $Phase_2$ at $T_1^f$. The process is the same as in the SLD strategy – $P_0$ will construct a "virtual tree" for all available processors, determine an equivalent single-level tree, and then compute an optimal load distribution for available processors.

However, in the DNP case, at time $T_1^f$ two distinct exceptions might occur. First, the available processor set may be empty. Such a situation will occur when the effective speed of all processors engaged in computation in $Phase_1$ become slower. This means that all processors engaged in computation in $Phase_1$ are still processing their load fractions at $T_1^f$, meanwhile other processors are still calculating their PL. In this case, $P_0$ will wait until one processor finishes its job and sends back its PTC. After receiving the "first responded" PTC, $P_0$ will send $L_2^p = L_1^p$ to this processor, thus starting $Phase_2$.

Second, at time $T_1^f$, one or more sub-tree sets may still be shadowed, because some communication links within the sub-trees become extremely slow. If no available processors is from the shadowed set, $Phase_2$ can be carried out normally as we described above. Thus, an interesting and relevant question to address is how to avoid potential congestion when one or more available processors belong to the currently shadowed set. One possible method is to exclude these available processors

from the "virtual tree", and $P_0$ will start $Phase_2$ using the rest of the available processors. This method is simple but only applicable to respective stable tree networks. This is because, in stable tree networks, the chance that a sub-tree set will continue to be shadowed at $T_1^f$ is rare, as communication speed is usually much faster than computation speed ($z < 0.1w$), and hence simply excluding available processors from shadowed sub-tree set will not degrade total performance significantly.

However, if the tree network is highly unstable, the above method is no longer suitable. Instead of blocking the whole shadowed sub-tree, we only want to block a given portion of the sub-tree, so that computation power can be utilized efficiently. Actually, $P_0$ can detect which link is still communicating load at $T_1^f$. Suppose, during $Phase_1$, $P_0$ dispatched a load fraction to $P_k$, and $P_i$ and $P_j$ are two adjacent processors in $\pi_{0k}$. If at $T_1^f$, $P_0$ receives CTC from $P_i$ and does not receive CTC from $P_j$, then communication using $l_j$ must continue. Therefore, $P_0$ only needs to block the branch rooted at $P_i$. Available processors from the rest of the shadowed sub-tree can still be included to construct the "virtual tree".

As in the SLD strategy, the above process continues until the beginning of $Phase_i$ when unprocessed load (remaining load plus the load being processed) is less than a pre-defined threshold $\mu L$ or the remaining load is less than $L_i^p$. It should be noted that, during the above process, the "virtual tree" for each phase may shrink as time progresses, unlike in SLD strategy, where "virtual tree" for each phase never shrinks. During the final phase (after estimation), $P_0$ will dispatch all the remaining load to currently available processors, and this phase is expected to be the last phase.

However, this does not imply finishing of computation, because $P_0$ has to check the PTC responses so as to make sure every load fraction is processed. If at the end of expected last phase, some processors that are engaged in computation, do not return their PTCs, or all the processors from a sub-tree rooted at one of the $P_0$'s children do not return their PL completion PTCs, $P_0$ will re-dispatch the corresponding load (assigned load or PL) to other available processors. The above process continues until all load is guaranteed to be processed, thus completing the entire computation process. The entire algorithm described above is shown in a flow-chart in Figure 3.7.

It should be noted that DLD may not be suitable for an extremely unstable network, where the link speeds and processor speeds change constantly. In fact, divisible load paradigm cannot be applied in this case, as any decision made at one time is outdated at the next time. DLD is designed to cope with mild changes in the network, and it is much more robust and resilient than SLD.

## 3.4 Performance Evaluation

In this section we shall quantify the performance of the two algorithms. We present two illustrative examples that demonstrate the operation of the algorithms. As the design of algorithms involve complex decisions during the load distribution process, a step-by-step illustration and discussions on the operation of the algorithm seems to be a more appropriate procedure than running conventional simulation tests that capture only the processing time performance. From this perspective, we design two examples, one for each case (SNP and DNP) and discuss the finer aspects that

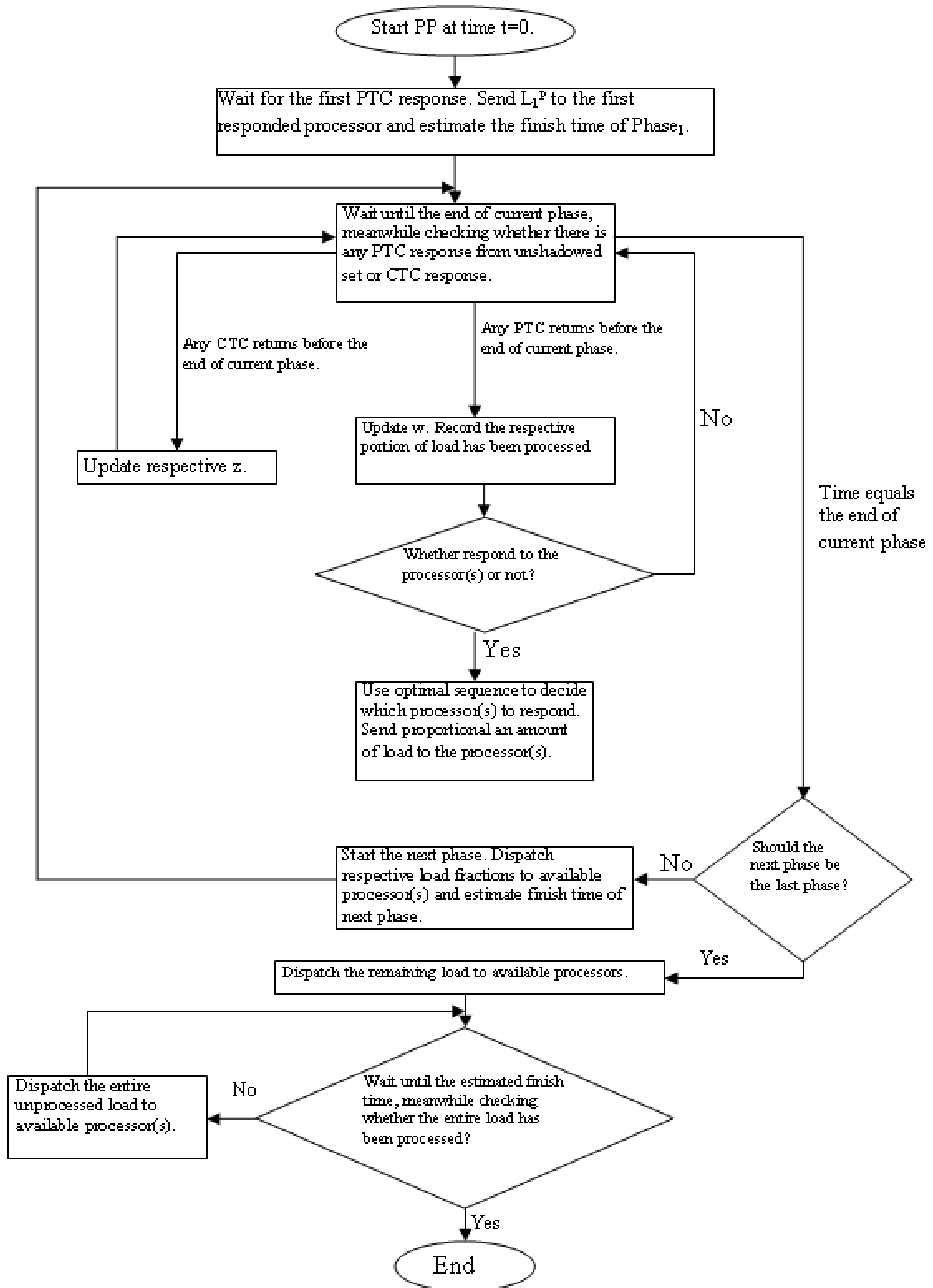


Figure 3.7: Flow Chart for DLD Strategy

Figure 3.8: Tree model for the experiment

went into the design of these algorithms. This approach, we believe, reveals several important characteristics that the algorithms exhibit.

### 3.4.1 Experiment with Static Network Parameter Case using SLD strategy

Consider a 5-level tree with 20 processors, as shown in Figure 3.8. We assume that $T_{cm} = 1$, $T_{cp} = 2$, $L = 80$, $L_1^p = 4$, $\eta = 0.05$, and $\mu = 0.15$.

The first step starts by sending the probe loads as described in Section 2.2. As $P_0$ has 4 children $(P_1, P_2, P_3, P_4)$, we have totally 4 PLs, each of size 1. The corresponding CTCs $(T_i^{cs}, T_i^{cf})$ and PTCs $(T_i^p)$ times used are listed in Table 3.1, from which we can calculate their $z_i$ and $w_i$ using Eqns.(3.1)–(3.2), respectively.

From Table 3.1, we observe that two processors ($P_4$ and $P_7$) and one link ($l_{15}$)

Table 3.1: PTC and CTC responses from Processors

| $P_i$ | $T_i^{cs}$ | $T_i^{cf}$(CTC) | $T_i^{p}$(PTC) | $w_i$ | $z_i$ |
|---|---|---|---|---|---|
| 1 | 0 | 0.5 | 14.5 | 7.0 | 0.5 |
| 2 | 0.5 | 0.7 | 4.7 | 2.0 | 0.2 |
| 3 | 0.7 | 1.1 | 3.1 | 1.0 | 0.4 |
| 4 | 1.1 | 2.0 | 202 | 100.0 | 0.9 |
| 5 | 0.5 | 1.0 | 5 | 2.0 | 0.5 |
| 6 | 0.7 | 1.0 | 3 | 1.0 | 0.3 |
| 7 | 1.0 | 1.3 | 201.3 | 100.0 | 0.3 |
| 8 | 2.0 | 2.6 | 18.6 | 8.0 | 0.6 |
| 9 | 2.6 | 2.8 | 20.8 | 9.0 | 0.2 |
| 10 | 1.0 | 1.4 | 9.4 | 4.0 | 0.4 |
| 11 | 1.4 | 2.2 | 16.2 | 7.0 | 0.8 |
| 12 | 2.2 | 3.0 | 9.0 | 3.0 | 0.8 |
| 13 | 3.0 | 3.2 | 13.2 | 5.0 | 0.2 |
| 14 | 1.0 | 1.1 | 17.1 | 8.0 | 0.1 |
| 15 | 2.8 | 12.8 | 16.8 | 2.0 | 10.0 |
| 16 | 1.1 | 1.3 | 4.3 | 2.0 | 0.2 |
| 17 | 12.8 | 13.3 | 21.3 | 4.0 | 0.5 |
| 18 | 13.3 | 13.7 | 21.7 | 4.0 | 0.4 |
| 19 | 13.7 | 14.4 | 26.4 | 6.0 | 0.7 |

have extremely slow speed. Later we will show how our algorithm copes with these extremely slow processors and links. Below, we present the operation of the SLD strategy in a step-by-step fashion.

At time $t = 0$, $P_0$ dispatches PLs to its children. This process takes 2 units of time, and no PTCs are returned during this period. Thus, $P_0$ remains idle until $t = 3.0$ at which time it receives the first PTC response from $P_6$. According to our algorithm, $P_0$ dispatches $L_1^p = 4$ to $P_6$, and hence starts $Phase_1$. Based on the estimated processing and link speeds, $P_0$ can determine the time $P_6$ needs to process $L_1^p$, using the speed parameters in Table 3.1, which is $((0.2 + 0.3) + 1.0 \times 2) \times 4 = 10$ units of time. $P_0$ sets the finish time of $Phase_1$ as $10 + 3 = 13$ units, which is equal to its duration from its start time, and shadows the set $|P_2|$ for 2 units of time (as it spends 2 units of time for $P_6$ to finish its reception) from $t = 3$.

At time $t = 3.8$ $(= L_1^p z_2 + 3)$, $P_0$ finishes sending $L_1^p$ to $P_2$, and finds that $P_3$ has already returned its PTC. Hence, $P_0$ will send $(13 - 3.8)/2.4 = 3.8333$ units of load[5] to $P_3$ to make it stop computing at $t = 13$, and shadows the set $|P_3|$ for 1.5333 units of time (time for $P_3$ to finish its reception) from $t = 3.8$. Note that the factor 2.4 (effective speed of $P_3$) in the denominator is computed using Eqn.(3.3) and the values in Table 3.1.

Thus, at time $t = 5.3333$ $(= 3.8333 z_4 + 3.8)$, $P_0$ becomes idle again and it finds that three processors from the unshadowed sets, $P_2, P_5, P_{16}$, have returned their PTCs. Based on the optimal sequence concept, $P_0$ will respond to $P_2$ and $P_{16}$ first. The equivalent computation power of $P_2$ and $P_{16}$ is 1.0698, and hence $P_0$ will send a total

[5](Finish time of $phase_1$ - current time)/effective speed of $P_3$, as expressed in Eqn. (3.6)

load of $(13-5.3333)/2.3396 = 3.2769$ units[6], among which 1.7528 units of load will be assigned to $P_2$ and 1.5241 units of load will be assigned to $P_{16}$. The shadow time for the set $|P_2|$ is 1.5701 (time to complete reception by $P_{16}$). After $P_0$ sends load to $|P_2|$, $P_0$ will respond to $P_5$ at time $t = 5.9887$ (after it finishes sending load to $P_2$), as no more PTCs return before this time. According to Eqn. (3.6), $(13-5.9887)/5 = 1.4023$ units of load is sent to $P_5$, and set $|P_1|$ would be shadowed for 1.4023 units of time (time to complete reception by $P_5$) from $t = 5.9887$.

Note that $P_0$ finishes sending load to $P_1$ at $t = 6.6899$. However, as no new PTC responses arrive before that time, it remains idle until it receives PTC from $P_{12}$ at $t = 9$ (See Table 3.1). Similarly, $P_0$ sends $(13 - 9)/7.8 = 0.5128$ units of load to $P_{12}$ and shadows the set $|P_1|$ 0.9392 units of time. Then, we note that at $t = 9.4$, $P_{10}$ returns its PTC. Although $P_0$ is idle now, it will not respond to $P_{10}$ until $t = 9.9392$, when the set $|P_1|$ is no longer shadowed, and then $P_0$ will send 0.3256 units of load (as per Eqn. (3.6)) to $|P_{10}|$. This is the last dispatch in $Phase_1$.

The results of the above description and for all other phases are tabulated in Table 3.2. From the table, we can see that the load which has been processed during $Phase_1$ is 13.2291, and hence the remaining load is 62.7709 (4 units of load have been consumed as PLs), larger than the end threshold. At time $t = 13$, $P_0$ triggers $Phase_2$. Because there are seven participating processors, $L_2^p = 7L_1^p = 28$. $P_0$ first constructs a "Virtual Tree" as shown in Figure 3.9(a). Because all processors adopt optimal sequence when dispatching loads to their children, the "Virtual Tree" can be shrunk to an equivalent single level tree as shown in Figure 3.9(b). Applying the optimality

[6] The denominator is the effective speed of equivalent node of $P_2$ and $P_{16}$, which equals 2.3396

Table 3.2: Load Distribution of SNP Case

| Phase | $P_i$ | $T_i^p$ | $P_0$ Responds Time | Dispatched Load |
|---|---|---|---|---|
| $Phase_1$ | 2 | 4.7 | 5.3333 | 1.7528 |
| | 3 | 3.1 | 3.8 | 3.8333 |
| | 5 | 5.0 | 5.9887 | 1.4023 |
| | 6 | 3.0 | 3.0 | 4 |
| | 10 | 9.4 | 9.9392 | 0.3256 |
| | 12 | 9.0 | 9.0 | 0.5128 |
| | 16 | 4.3 | 5.3333 | 1.4023 |
| $Phase_2$ | 2 | | 13.0 | 4.5211 |
| | 3 | | 13.0 | 7.5348 |
| | 4 | | 13.0 | 2.4971 |
| | 6 | | 13.0 | 7.4134 |
| | 10 | | 13.0 | 1.1893 |
| | 12 | | 13.0 | 1.3988 |
| | 16 | | 13.0 | 3.4487 |
| | 14 | 17.1 | 21.6328 | 0.7547 |
| | 8 | 18.6 | 21.7837 | 0.6120 |
| | 9 | 20.8 | 21.7837 | 0.5643 |
| | 15 | 16.8 | 21.7837 | 0.4369 |
| | 17 | 21.3 | 21.7837 | 0.1958 |
| | 18 | 21.7 | 21.7837 | 0.2081 |
| | 1 | 14.5 | 25.7672 | 0.5397 |
| | 11 | 16.2 | 25.7672 | 0.4570 |
| | 13 | 13.2 | 25.7672 | 0.6764 |
| | 19 | 26.4 | 32.5972 | 0.0657 |
| $Phase_3$ | 1 | | 34.16 | 0.7880 |
| | 2 | | | 4.2475 |
| | 3 | | | 7.0779 |
| | 5 | | | 2.0751 |
| | 6 | | | 6.8656 |
| | 8 | | | 0.5447 |
| | 9 | | | 0.5023 |
| | 10 | | | 0.9687 |
| | 11 | | | 0.5242 |
| | 12 | | | 1.0785 |
| | 13 | | | 0.8142 |
| | 14 | | | 0.8333 |
| | 15 | | | 0.3470 |
| | 16 | | | 3.1747 |
| | 17 | | | 0.1555 |
| | 18 | | | 0.1652 |
| | 19 | | | 0.0979 |

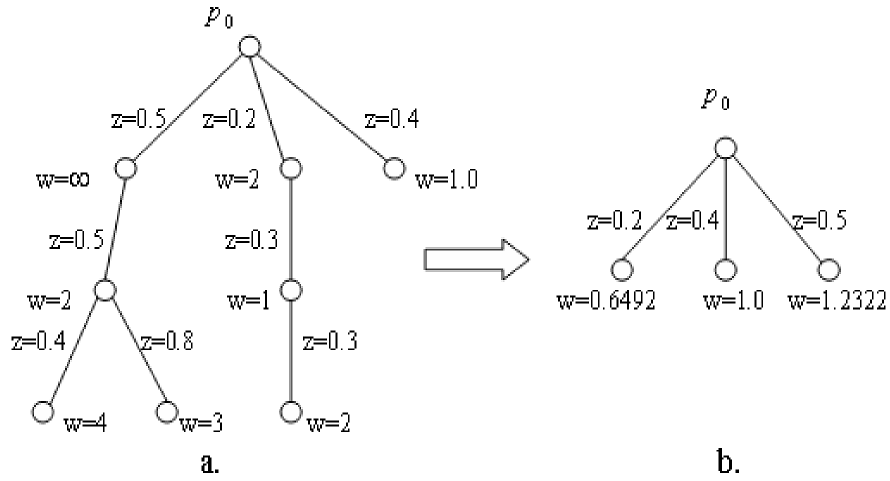


Figure 3.9: Virtual Tree and Equivalent Single Level Tree of SNP Case

principle in DLT literature on the "Virtual Tree", we can obtain an optimal load distribution as shown in Table 3.2. The "Virtual Tree" will spend 21.16 units of time to process $L_2^p$. Therefore, $P_0$ sets the finish time of $Phase_2$ as 34.16 and it will also set the corresponding shadow times, 12.7672, 7.3652, 6.0904, for sets $|P_1|, |P_2|, |P_3|$.

At time $t = 21.6328$ (time to finish sending $L_2^p$ to the virtual tree by $P_0$), $P_0$ becomes idle, and processors $P_1, P_8, P_9, P_{11}, P_{13}, P_{14}, P_{15}, P_{17}$ have returned their PTCs. However, because set $|P_1|$ continues to be shadowed, those PTCs from $|P_1|$ will not be responded to. Based on the optimal sequence criterion, $P_0$ will respond to $P_{14}$ first and send 0.7547 units of load to $P_{14}$. $|P_2|$ will be shadowed for 0.4528 units of time. At $t = 21.7837$ (after $P_0$ finishes sending load to $P_2$), $P_0$ dispatches a total of 2.0171 units of load to $P_8, P_9, P_{15}, P_{17}, P_{18}$, whose equivalent computation power is 2.6178, and shadows $|P_4|$ for 10.8099 units of time. Then, at time $t = 25.7672$, set $|P_1|$ becomes available, and hence $P_0$ dispatches totally 1.6731 units of load to

$P_1, P_{11}$, and $P_{13}$. At $t = 32.5972$, $P_0$ sends 0.0657 units of load to $P_{19}$, which sends back its PTC at $t = 26.4$. As we can see above, totally 32.5106 units of load have been processed during $Phase_2$, and the whole load distribution in this phase is also shown in Table 3.2.

Processor $P_0$ starts $Phase_3$ at $t = 34.16$. Since a total of 17 processors will participate in the computation at the beginning of this phase, $L_3^p$ should be equal to $17 * 4 = 68$ units of load. However, there are only 30.2603 units of remaining load, and hence we will dispatch the whole of the remaining load to the available processors at the beginning of $Phase_3$ (last phase), as shown in Table 3.2. The duration of the last phase is 23.04 units of time, and the total processing time is 57.2.

From the above example, we have the following observations. First, the extremely slow processors, such as $P_4$ and $P_7$, are automatically excluded from computing, and an extremely slow link $l_i$ will significantly affect the performance of processors that belong to set $|P_i|$. In our example, although $|P_{15}|$ contains very fast processors, such as $P_{15}$, this whole branch only receives 1.6271 units of load, because of an extremely slow link $l_{15}$.

Secondly, from this example, we conclusively see that the concept of shadow time can effectively avoid communication congestion. For instance, in our example, at time $t = 9.4$, when $P_{10}$ returns its PTC, $P_0$ does not respond to it immediately, although it is idle, because set $|P_1|$ is shadowed. Actually, this is due to the fact that $P_1$ was dispatching load to $P_5$ at $t = 9.4$, and it would have caused congestion if $P_0$ responded to $P_{10}$ immediately.

Thirdly, our algorithm adopts an optimal sequence criterion in every load dispatch to optimize the performance. For example, at time $t = 5.3333$, $P_0$ receives three PTCs, from $P_2, P_5$, and $P_{16}$, respectively. Based on the optimal sequence, $P_0$ responds to $P_2$ and $P_{16}$ first, and then responds to $P_5$. Compared with $P_0$ dispatching load to $P_5$ first and then to $P_2$ and $P_{16}$, it enables $P_2, P_5, P_{16}$ to process approximately a total of 0.2 units more load during $Phase_1$.

### 3.4.2 Experiment with Dynamic Network Parameter Case using DLD Strategy

Now we will present an experiment with the DLD strategy, which is capable of handling fluctuations in the network and node speeds, and demonstrate its similarities and differences with SLD strategy.

As before, we assume that $T_{cm} = 1$, $T_{cp} = 2$, $L = 80$, $L_1^p = 4$, $\eta = 0.05$, $\mu = 0.15$, and $\zeta = 0.1$. We further assume the tree topology and the first round of PTC and CTC responses are also the same as in the SNP case (shown in Figure 3.8 and Table 3.1, respectively). Further, we randomly choose a set of processors and links, and vary their speeds as shown in Figure 3.10. The speeds of other links and processors are assumed to remain the same during the processing of the total load.

It should be noted that $P_0$ does not know the variance of $w$ and $z$ in advance. However, as time progresses, $P_0$ can detect such variance and will incorporate the new value of $w$ and $z$ into load dispatching, as we will see in our example.

As in the SNP case, at $t = 3.0$, $P_0$ receives the first PTC response from $P_6$. It

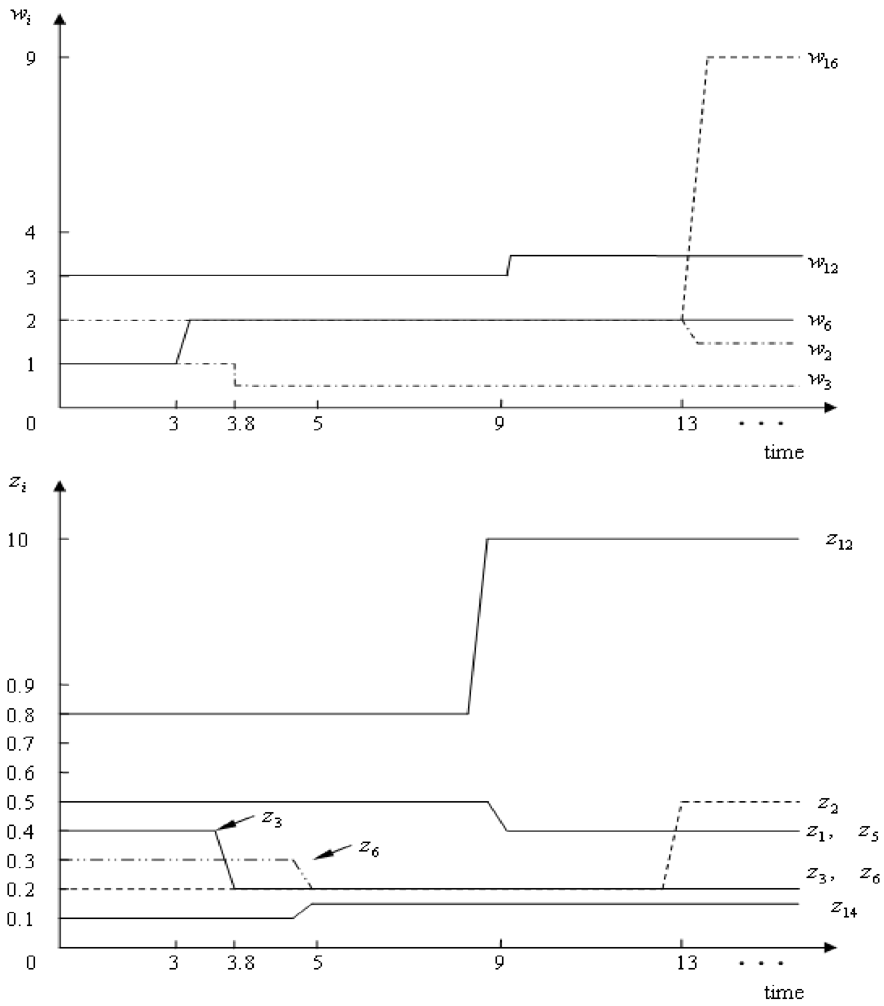


Figure 3.10: The variance of $w$ and $z$ with time

dispatches $L_1^p = 4$ to $P_6$, and using the estimated value of $z_2$, $z_6$ and $w_6$, as shown in Table 1, $P_0$ can determine the expected time for $P_6$ to process $L_1^p$, which is 10 units of time. However, unlike in the SNP case, during the transmission of $L_1^p$, in our design of DLD algorithm, every processor in $\pi_{06}$ will send back a CTC again when it finishes receiving load from its parent. These CTCs are used to update the relative link speeds. Further, $P_0$ will shadow set $|P_2|$ until it receives a CTC from $P_6$ again.

As shown in Figure 3.10, $z_3$ and $z_6$ remain the same at this time, but $P_6$ becomes slower and using the new value of $w_6$ we can deduce that $P_6$ will return its PTC at $t = 21.0$. From $P_0$'s perspective, it will receive two CTCs from $P_2$ and $P_6$ at $t = 3.8$ $(= 3 + z_3 L_1^p)$ and $t = 5.0$ $(= 3 + (z_3 + z_6)L_1^p)$, respectively, however, it will not receive any PTC response from $P_6$ by $t = 13.0$. Hence, $P_6$ will be automatically excluded from computation right at the beginning of $Phase_2$. However, later at $t = 21.0$ when $P_0$ receives this PTC, it can re-estimate the value of $w_6$ using Eqn. (3.2) and can obtain the recent value for $w_6$ which now increases by a factor of 2 during its computation. Clearly, this case demonstrates the possibility of processor exclusion in a phase by our algorithm.

At time $t = 3.8$ $(= L_1^p z_2 + 3)$, $P_0$ finishes sending $L_1^p$ to $P_2$, and starts to dispatch load to $P_3$, which has already returned its PTC at $t = 3.1$. As in SLD strategy, $P_0$ use Eqn.(3.6) to determine the amount of load that should be sent to $P_3$. Consequently, 3.8333 units of load is sent to $P_3$. However, as shown in Figure 3.10, both $l_3$ and $P_3$ becomes twice faster ($z_3 = 0.2$, $w_3 = 0.5$) when it computes the load, and hence $P_0$ will receive CTC and PTC from $P_3$ at $t = 4.5667$ $(= 3.8 + 3.8333z_3)$ and $t = 8.4$ $(= 3.8 + 3.8333(z_3 + 2w_3))$.

At time $t = 4.5667$, when $P_0$ receives CTC from $P_3$, it updates $z_3$ to 0.2. Because, by this time, $P_0$ has not received the CTC from $P_6$, it will not respond to $P_{16}$, which returned its PTC at $t = 4.3$. At $t = 5.0$, set $|P_2|$ is no longer shadowed, and by this time $P_0$ has received PTCs from $P_2, P_5$, and $P_{16}$. Similar to the SNP case, $P_0$ adopts the optimal sequence criterion, and hence sends a total of 3.4194[7] units of

[7]Computed as described in the SNP case. However, when computing the effective speed of equiv-

load to $P_2$ and $P_{16}$ first, whose currently equivalent computation power is 1.0698. As indicated in Figure 3.10, $z_6$ and $z_{14}$ change to 0.2 and 0.15, respectively, and all the other link speeds and processing speeds remain the same. Therefore, $P_2$ will finish its job exactly at the expected time $t = 13$, while $P_{16}$ will finish at a time slightly earlier than expected (at $t = 12.92$).

At time $t = 5.6839$ ($= 5 + 3.4191 \times 0.2$), $P_0$ finishes sending load to $P_2$, and then sends 1.4632 units of load (as per Eqn. (3.6)) to $P_5$. Because $z_1, z_5, w_5$ all remain unchanged, $P_5$ will return its PTC at $t = 13$. After dispatching load to $P_1$, $P_0$ remains idle until it receives PTC from $P_3$ again at $t = 8.4$. Then, $P_0$ knows that the load sent to $P_3$ earlier has been processed, and it will send another 3.8333 units of load to $P_3$. Because, at this time, $z_3$ and $w_3$ do not change, $P_3$ will finish its job at $t = 13$. $P_3$ receives a total of 7.6666 units of load during $Phase_1$. Thus, we note that our algorithm tracks and attempts to reuse fast processors (in this case $P_3$) as shown in the above case.

At time $t = 9.1667$ ($= 8.4 + 3.8333 \times 0.2$), $P_0$ sends 0.4914 units of load to $P_{12}$. However, as shown in Figure 3.10, at this time the values of $z_1, z_5, z_{12}$, and $w_{12}$ become 0.4, 0.4, 10.0, and 3.5, respectively. We observe that, because $l_{12}$ becomes extremely slow now, $P_0$ cannot receive the CTC response from $P_{12}$ before the beginning of $Phase_2$. Thus, PTC from $P_{10}$ (at $t = 9.4$, shown in Table 3.1) will not be responded to during $Phase_1$. This event indicates that subsequently, while constructing a virtual tree, the sub-tree rooted at $P_5$ must be shadowed, which highlights the tracking capability of the algorithm.

---

alent node of $P_2$ and $P_{16}$ we use the most recent estimates of $z$ and $w$.

Now, at $t = 12.92$, $P_0$ receives PTC from $P_{16}$. However, because 12.92 falls into the range $[13 - 10 \times 0.1, 13]$, $P_0$ just records the portion of the load that has been processed by $P_{16}$ and does not send any more load fractions to the processor. This is because the current time is very close to the start of the next phase. It may be noted that sending another amount of load to $P_{16}$ improves the performance only by a very little amount. However, it increases the risk that $P_0$ may miss the start time of the next phase, should $l_2$'s speed become very slow. Thus it may be noted that, for the next phase, DLD is able to retain all the processors that had responded before the expected finish time in the current phase (in this case, this is $P_{16}$).

In $Phase_1$, totally 17.0403 units of load has been dispatched, while only 12.5492 units of load has been processed. Thus, the remaining load is 58.9597 (4 units of PL have been dispatched), and the unprocessed load is 64.0403 (3 units of PL have been processed). The load distribution is shown in Table 3.3.

At time $t = 13$, $P_0$ will trigger $Phase_2$. Based on the DLD strategy, $P_0$ shadows the branch $|P_5|$, because now $P_5$ is still sending data to $P_{12}$. A "Virtual Tree" is constructed as shown in Figure 3.11(a) and its equivalent single level tree is shown in Figure 3.11(b). $P_0$ will dispatch $L_2^p = 3 \times 4 = 12$ units of load to the "Virtual Tree", and an optimal load distribution is shown in Table 3.3. The "Virtual Tree" is supposed to spend 10.0744 units of time to process $L_2^p$, and hence the expected beginning time of $Phase_3$ is 23.0744. However, as shown in Figure 3.10, the values of $z_2$, $w_2$ and $w_{16}$ become 0.5, 1.5 and 9.0, respectively, and the finish times of $P_2, P_{16}, P_3$ are 22.0699, 39.7026, and 24.372, respectively. Note that because of the distribution

Table 3.3: Load Distribution of DNP Case

| Phase | $P_i$ | PTC Return Time | $P_0$ Responds Time | Dispatched Load | Actual Finish Time |
|---|---|---|---|---|---|
| $Phase_1$ | 6 | 3.0 | 3.0 | 4 | 21.0 |
| | 3 | 3.1 | 3.8 | 3.8333 | 8.4 |
| | 2 | 4.7 | 5.0 | 1.8289 | 13.0 |
| | 16 | 4.3 | 5.0 | 1.5905 | 12.92 |
| | 5 | 5.0 | 5.6839 | 1.4632 | 13.0 |
| | 3 | 8.4 | 8.4 | 3.8333 | 13.0 |
| | 12 | 9.0 | 9.1667 | 0.4914 | 17.9136 |
| $Phase_2$ | 2 | 13.0 | 13.0 | 2.3023 | 22.0699 |
| | 16 | 12.92 | 13.0 | 2.0237 | 52.7026 |
| | 3 | 13.0 | 13.0 | 7.674 | 24.372 |
| | 1 | 14.5 | 16.6978 | 0.3786 | 23.0744 |
| | 5 | 13.0 | 16.6978 | 1.0943 | 23.0744 |
| | 10 | 9.4 | 16.6978 | 0.5109 | 23.0744 |
| | 11 | 16.2 | 16.6978 | 0.2761 | 23.0744 |
| | 13 | 13.2 | 16.6978 | 0.4291 | 23.0744 |
| | 14 | 17.1 | 17.7735 | 0.3146 | 23.0744 |
| | 15 | 16.8 | 17.9308 | 0.3406 | 23.0744 |
| | 12 | 17.9136 | 19.209 | 0.2172 | 23.0744 |
| | 6 | 21.0 | 21.0 | 0.4414 | 23.0744 |
| | 8 | 18.6 | 21.712 | 0.0688 | 23.0744 |
| | 9 | 20.8 | 21.712 | 0.0635 | 23.0744 |
| | 17 | 21.3 | 21.712 | 0.0387 | 23.0744 |
| | 18 | 21.7 | 21.712 | 0.0411 | 23.0744 |
| $Phase_3$ | 1 | | 23.0744 | 2.6614 | 68.572 |
| | 2 | | | 9.4955 | |
| | 5 | | | 7.5594 | |
| | 6 | | | 6.7031 | |
| | 8 | | | 1.4227 | |
| | 9 | | | 1.3118 | |
| | 10 | | | 3.529 | |
| | 11 | | | 1.9074 | |
| | 12 | | | 1.5708 | |
| | 13 | | | 2.9645 | |
| | 14 | | | 1.6601 | |
| | 15 | | | 1.0156 | |
| | 17 | | | 0.4552 | |
| | 18 | | | 0.4838 | |

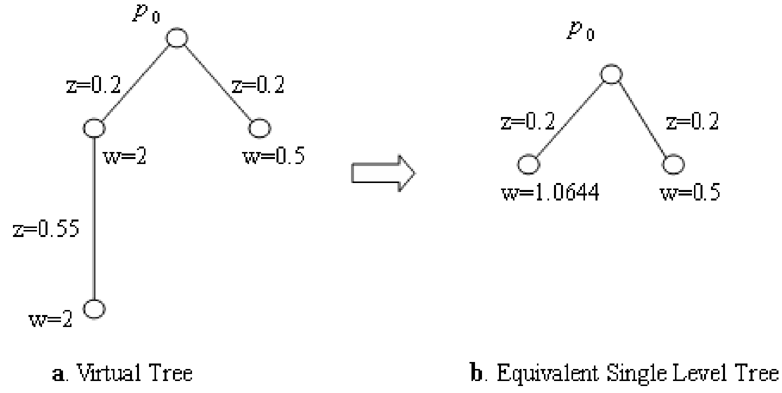


Figure 3.11: Virtual Tree and Equivalent Single Level Tree of DNP Case

sequence, even if $l_3$ and $w_3$ remain the same, $P_3$ still can not finish computing at the expected time.

The load distribution in $Phase_2$ is shown in Table 3.3. During $Phase_2$, totally 16.2151 units of load have been dispatched, and hence only 42.7446 units of load is left at time $t = 23.0744$. As 14 processors will now participate at the beginning of $Phase_3$, the remaining load is less than $L_3^p = 56$, therefore all the remaining load will be dispatched at $t = 23.0744$. The load distribution is shown in Table 3.3. $Phase_3$ lasts for 45.4976 units of time. Notice that $P_3$ and $P_{16}$, which have not returned their PTCs at the beginning of $Phase_3$, finish their job during this phase. Therefore, all the load has been processed at the end of $Phase_3$ and the total processing time is 68.572.

From the above example, we can see that the time-varying nature of the tree network is captured in the DLD strategy, and is dynamically incorporated into the load distribution. This is a unique characteristic of the DLD strategy. If the SLD

strategy is applied in the above example, it detects the link speed and node speed only once, and hence will not take into account the speed variance. Because the initial speed parameters are the same in the above two examples, the SLD strategy will try to distribute the total load in the same way as in the first example. However, such a distribution by SLD may cause communication congestion in the above example, as it does not take into account the link speed variance. For instance, in the above example, $l_{12}$ becomes extremely slow at time around $t = 9$. The SLD strategy will not detect such a change, and hence may send some load to $|P_5|$, while $P_5$ is still sending load to $P_{12}$. Further, we notice that a total of 18.279 units of load is sent to $P_6$ in the first example. However, as shown in the DNP example, $P_6$ should only receive a total of 11.1445 units of load because its speed becoming slower during the processing. Obviously, $P_6$ will be overloaded if we apply the SLD strategy in the above example, as it does not take into account the $P_6$'s computation speed variance. This will highly degrade the total performance.

# Chapter 4

# Issues in Handling Divisible loads on Arbitrary Networks

## 4.1 Probing & Reporting Techniques

In the previous two chapters, we addressed the problem of scheduling divisible loads in linear networks and multi-level tree networks in resource unaware environment. One common technique they used is probing technique. To some extent, the probing technique can be regarded as a centralized scheme in the sense that the root controls the whole process. For different network topologies, the root will probe the network in different manners. There are two advantages of the probing technique. First, through probing, the root obtains the speed parameters of links and processors, which serves as a basis for the future scheduling. Secondly, while probing the network, although may be small amount (depending on the size of probing load), at least a portion of the real job has been done.

However, probing technique also has some limitations. First, the underlying network should have a regular topology. Then, depending on the topology of the network (be it bus, linear, or tree), the root can conduct probing in a corresponding manner. The probing technique may not be suitable for the network with an arbitrary topol-

ogy. In an arbitrary graph, a processor may have more than one route connecting to the root, and may receive different or duplicate probing loads from the root through different routes. It is difficult for the root to control the probing, especially when each processor has a single port for transmission. In this case, as we discussed in Chapter 3, communication congestion problem may happen. Some probing loads may be blocked somewhere in the network, and some part of network may not be probed in time.

The probing becomes even more unpredictable if multiple sources exist in the network. In this case, a processor may not only receive probing loads from different routes but also from different sources. Further, the probing technique may not be really suitable for large-scale networks. In probing technique, all probing loads are initially sent out by the root. All processors will wait until receiving the probing load and then start to compute. In large-scale networks, a probing load may travel a long time before reaching the processor which is far from the source. This leads to quite long idle time for those processors, which could be used more efficiently.

From what we discussed above, we can see that because of exercising a centralized control, the probing technique may not be a suitable technique to handle multi-source divisible loads in large-scale arbitrary networks. Actually, some processors may be aware of its own speed and the speeds of the links that directly connect to them. Even if this speed information is not known, it is much easier to detect this information by a local processor other than the root which may be far away.

Therefore, an alternative way to detect the network speeds can be a reporting

based scheme. In reporting technique, when the source needs the speed information of the remaining network, it can simply signal other processors. A processor, after receiving the signal can report back the speed information if it already had, or can start to detect the local speed information and then report back. The source can set up a time threshold, say $T^*$, and will start to schedule and process the load at time $t = T^*$ based on the information it received before this time.

Compared to probing technique, in reporting technique, no real job is done in the first $T^*$ units of time. However, a reporting scheme can work more efficiently than a probing scheme in the complex cases, since in reporting scheme, the root may get the speed information much faster. Further, in a reporting scheme, a processor is not limited to report back only the speed information. Actually, even with the local information, a processor can still obtain other useful results such as, which is the nearest (in terms of communication delay) source, or what is the best route for it to receive its chunk of load. The processor can also report these useful results back. This idea is discussed in more detail in the next chapter, where we address the problem of scheduling multi-source divisible loads in arbitrary networks, under both resource aware and resource unaware cases. However, below we will first discuss another relevant issue in scheduling in arbitrary networks.

## 4.2 Common Spanning Trees - Performance Evaluation

### 4.2.1 Problem Formulation and Notations

As we have mentioned in the introductory chapter, optimal solution to single-instalment based divisible load scheduling problem on a network with an arbitrary graph topology indeed occurs on a spanning tree of the graph [53]. However, Byrnes et al. [52] proved that finding the optimal spanning tree (the spanning tree that generates minimum total processing time) on the arbitrary network is NP-hard. Therefore, one immediate question to address is which spanning tree(s) can deliver efficient solutions.

In this section, we evaluate the performance of different spanning trees over a wide range of arbitrary networks with varying connectivity and processor densities and study the effect of network scalability. The underlying network considered comprises heterogeneous processors interconnected by heterogeneous links in an arbitrary manner. Each processor is assumed to be equipped with "front-end", and only has a single port for transmission. We study and compare the performances of different spanning trees - minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), and the robust spanning tree (RST) [53], with respect to a variety of performance metrics, such as complexity, time performance and robustness. In addition, to minimize the total processing time of the entire load submitted for processing, we propose a novel spanning tree routing strategy, which

is referred to as *minimum equivalent network spanning tree* (EST) and compare its performance as well. Resource-aware optimal load distribution with optimal sequencing (RAOLD-OS) [51] scheduling algorithm is applied to all the above spanning tree routing strategies for obtaining an optimal solution. We set up experiments systematically to evaluate the performance of these spanning tree routing strategies over a wide range of arbitrary dense graphs with varying connectivity and processor densities. This work attempts to pool all known and applicable divisible load scheduling algorithms for arbitrary networks and presents a collective and comparative view of their performance.

The notations, definitions, and the terminology that are used in this chapter are given in Table 4.1.

### 4.2.2 Common Spanning Tree Routing Strategies

For an arbitrary graph, there normally exist many spanning trees. Below, we introduce several common spanning tree construction strategies and their characteristics in brief.

**Minimum spanning tree (MST)**: In MST, the total link weight (the link weights depend on the speed of the links) is the minimum among all the spanning trees. Since MST always tends to incorporate the link with small weight without considering its hop count to the root, normally MSTs are very deep and "skinny". Kruskal's or Prim's algorithm are used to construct such a spanning tree.

**Shortest path spanning tree (SPT)**: In SPT, each node has the shortest path

Table 4.1: List of notations

| Notation | Description |
|---|---|
| $L$ | The total amount of load originating at a node for processing. |
| $p_s$ | Node $s$ in the given graph $G$. |
| $l_{p_s,p_t}$ | Communication link connecting nodes $s$ and $t$ in the graph $G$. |
| $p_{x,i}$ | This denotes node $i$ in a spanning tree whose parent is node $x$. |
| $\Sigma(x, i, m + 1)$ | This is a single-level tree network (sub-tree) defined in a spanning tree, consisting of $(m + 1)$ nodes, with root node $p_{x,i}$ and $m$ child nodes $p_{i,1}, ..., p_{i,k}, ..., p_{i,m}$. |
| $T_{cp}/T_{cm}$ | Time taken to compute/transmit a unit load by a standard node/link, as defined in Chapter 2. |
| $w_{x,i}/z_{x,i}$ | Ratio of the time taken to compute/transmit a certain amount of data by $p_{x,i}/l_{x,i}$ to the time taken by a standard node/link, as defined in Chapter 2. Note that, $w_{i,k}/z_{i,k}$ is actually the inverse of the speed of $p_{i,k}/l_{i,k}$ in $\Sigma(x, i, m + 1)$. |
| $T(\alpha)$ | The total processing time of the entire load under the distribution $\alpha$. |
| $\epsilon$ | The eccentricity is the depth of the deepest leaf node from the root node, in terms of number of hops, in a spanning tree. |

(in terms of link weights) to the root. To construct such a spanning tree, either the efficient Dijkstra's or Bellman-Ford's algorithm could be used. The shape of the tree depends on the distribution of the link weights. The SPT trees are generally deeper and have smaller node degrees than FHT trees.

**Fewest hops spanning tree (FHT)**: In FHT, each node's hop count to the root is the minimum. The breadth-first search (BFS) algorithm [92] could be used to construct the FHT. FHTs tend to be shallow and "fat".

**Robust spanning tree (RST)**: RST is designed to seek a trade-off between link weight and hop count. Such a tree is immune to data loss when nodes or links fail and yet provides good performance. RST minimizes each node's combined cost of link weight and hop count as follows.

$$\lambda * hop\ count + (1 - \lambda) * link\ weight \tag{4.1}$$

The weight $\lambda$ is actually a function of a node's depth in the tree, which falls into the range $[0, 1)$ When an edge $(i, j)$ is being considered for inclusion in the tree, then

$$\lambda_i = 1 - \frac{h_i}{\epsilon_1} \tag{4.2}$$

where $i$ is the new vertex not already in the tree, $h_i$ is the hop count of node $i$ from the root and $\epsilon_1$ is the depth of the deepest leaf in the shortest path spanning tree (SPT) or in other words it is the deepest of the shortest paths from the root node to all other nodes in the network, and this gives the relative importance of hop count versus link weights. RST strives for a balance between SPT and FHT.

**Minimum equivalent network spanning tree (EST)**: Our EST algorithm assumes optimal sequencing load distribution and maximizes the equivalent computa-

tion power of the spanning tree by considering both processor and the link weights (or speeds) while constructing the spanning tree as follows. In [4], the equivalent processor value $w_{x,eq(i)}$ for a single-level tree $\Sigma(x, i, m+1)$, is derived as

$$w_{x,eq(i)} = \left( \frac{\Pi_{v=1}^{m} f_v}{1 + \Sigma_{u=1}^{m} \Pi_{v=u}^{m} f_v} \right) w_{x,i} \tag{4.3}$$

where

$$f_v = \frac{w_{i,v} + z_{i,v}(T_{cm}/T_{cp})}{w_{i,(v-1)}} \tag{4.4}$$

Given a single-level tree network, the entire network could hence be replaced with an equivalent processor with speed parameter as given by the equation 4.3. Our EST spanning tree construction algorithm uses this procedure in a recursive fashion. Given an arbitrary network ($G$) containing nodes ($N$) and links ($E$), we first add the root node to the spanning tree and then, consider all the links originating from this spanning tree, one by one, and add the $(E, N)$ pair that provides minimum effective equivalent processor value ($w_{eq(0)}$) (as in the equation 4.3) to it and continue until all the nodes in $G$ are added. The shape of EST trees depend on the distribution of the links as well as processor speeds.

The Figure 4.1 (b), (c), (d), (e) and (f) presents the MST, SPT, FHT, RST, and EST spanning trees rooted at $p_1$ constructed for an arbitrary graph network $G$ given in Figure 4.1 (a). Though in this example the spanning tree results are different, in general it may not be so, the trees constructed by different spanning tree algorithms might be identical. In our simulation study, we apply the RAOLD-OS algorithm on all the generated spanning trees, capture the critical parameters and analyze the performance of various routing strategies.

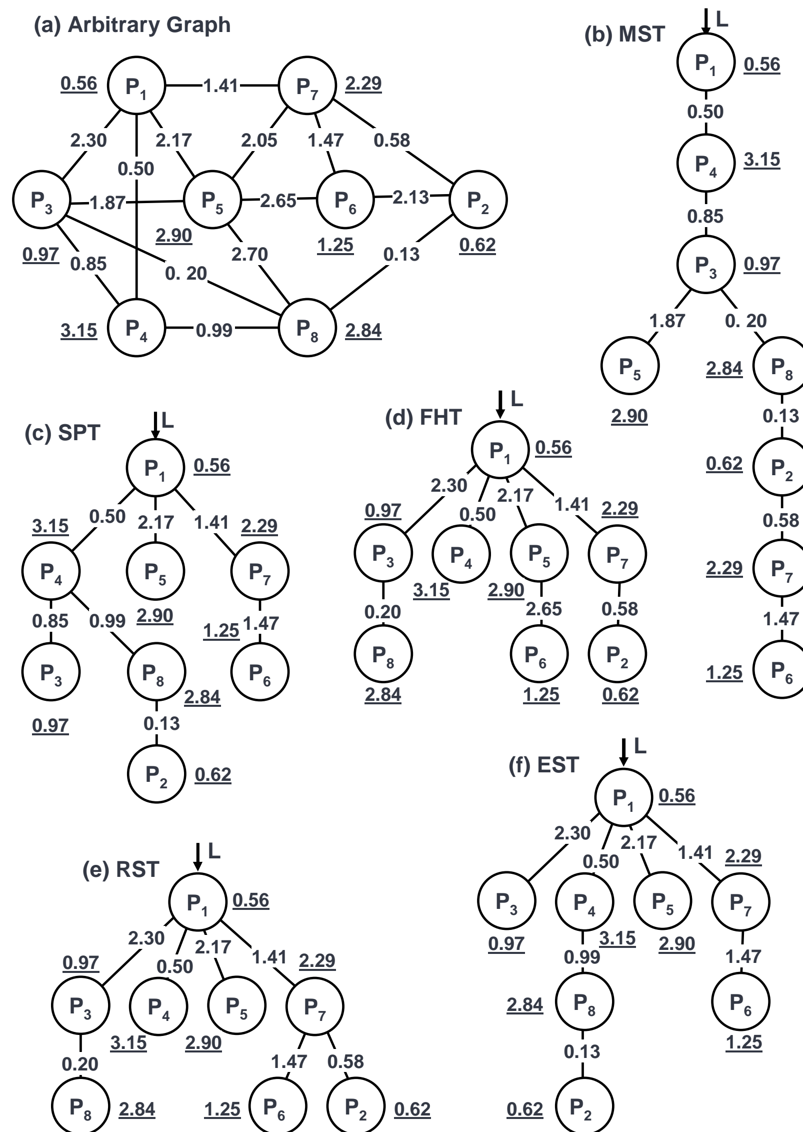


Figure 4.1: An arbitrary graph network and spanning trees (number on the links denote the link weights and the number near the nodes denote the processor weights). (a) An arbitrary graph network $G$ with 8 processing nodes; (b) Minimum spanning tree; (c) Shortest path spanning tree; (d) Fewest hops spanning tree; (e) Robust spanning tree; (f) Minimum equivalent network spanning tree.

### 4.2.3 Performance Evaluation

In this Section, we shall describe the simulation platform setup and present the performance of various spanning tree routing strategies for various situations through extensive simulations. We also highlight and discuss all the important simulation results. We compare the performance of various routing strategies based on the total processing time, and network eccentricity, which we define as the distance in number of hops from the root node to the farthest leaf node in the spanning tree. In addition to the total processing time, we also consider network eccentricity in our study, since it provides an indication on how far the nodes are from the root node in a spanning tree. This metric gives a measure of robustness of the network, since the farther the nodes are from the root node, more pronounced will be the effect of network disruptions on the performance because of data loss [53].

We now describe how the arbitrary graphs and other required parameters for our performance evaluation are generated. The graph generation procedure is made to be non-deterministic so as to reflect the real-life situations. We set the node $p_0$ in the network as the root node. The parameter $P_{link}$ denote the degree of connectivity, or *link density*. By varying the number of processing nodes and the $P_{link}$ parameter in our simulations, we generate various types of networks. This allows us to generate graphs with very small number of processors with high connectivity and graphs with large numbers of processors with low or sparse connectivity, to reflect real-life scenarios.

In our study, we vary the $P_{link}$ parameter from 30% to 100% in steps of 10%, and generate various types of networks and for each type of network. We also vary

the number of processing nodes from 10 (small-size graph) to 200 (large-size graph) to study the effects of network size scalability. It shall be noted that in order to guarantee the generated graph is a connected graph, the value of $P_{link}$ parameter cannot be close to zero and when the value of $P_{link}$ parameter is 100%, we have a completely connected graph network where in all the nodes are connected to each other by a direct link. The speed parameters for the processing nodes and the links are chosen based on a uniform probability distribution in the range $[0.01, 3.34]$ for low, and $[6.67, 10.0]$ for high values. In all our studies, we let $L = 10^8$, $T_{cm} = T_{cp} = 1.0$, and vary the number of nodes in the network, speed and $P_{link}$ parameter values and analyze the performance.

The network eccentricity results are plotted in the Figure 4.2. The total processing time results are plotted in the Figure 4.3 and 4.4 for low and high link speed values respectively. In the Figures 4.2, 4.3, and 4.4, we denote the results for minimum spanning tree, shortest path spanning tree, fewest hops spanning tree, robust spanning tree, and minimum equivalent network spanning tree as $MST, SPT, FHT, RST$, and $EST$ respectively. Since the EST construction depends on both processor and link speeds, its eccentricity value vary when the network has high or low processing speed nodes. Hence, they are plotted separately as $EST_H$ and $EST_L$ in Figure 4.2.

#### 4.2.3.1 Effect of network scalability

We study the effect of network scalability by comparing the performance of the routing strategies for various processing node configurations for a given link density value. We first notice that MST has the largest eccentricity values (tree depth in terms of

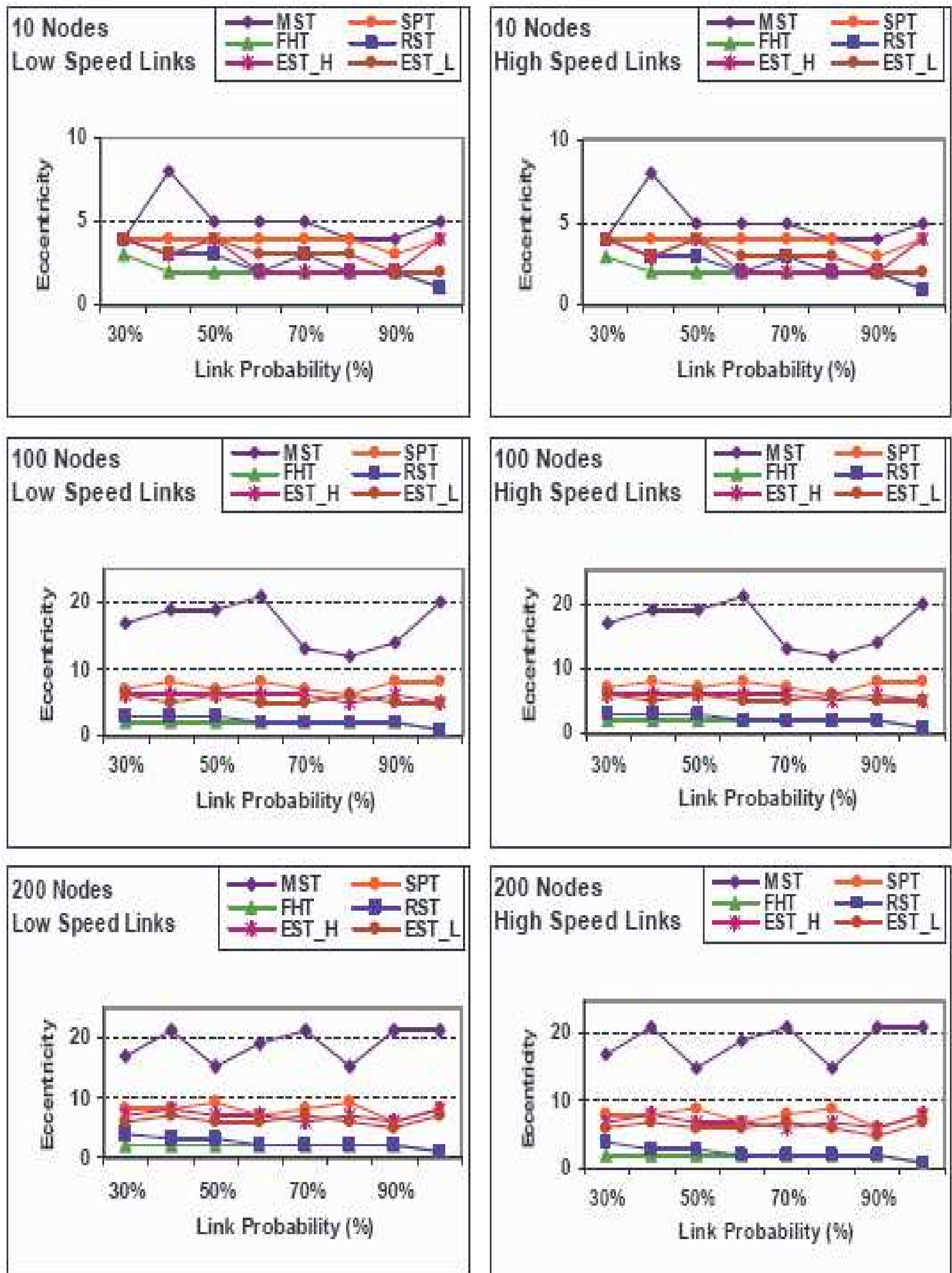


Figure 4.2: Network eccentricity simulation results for 10, 100, and 200 nodes network with low and high speed links

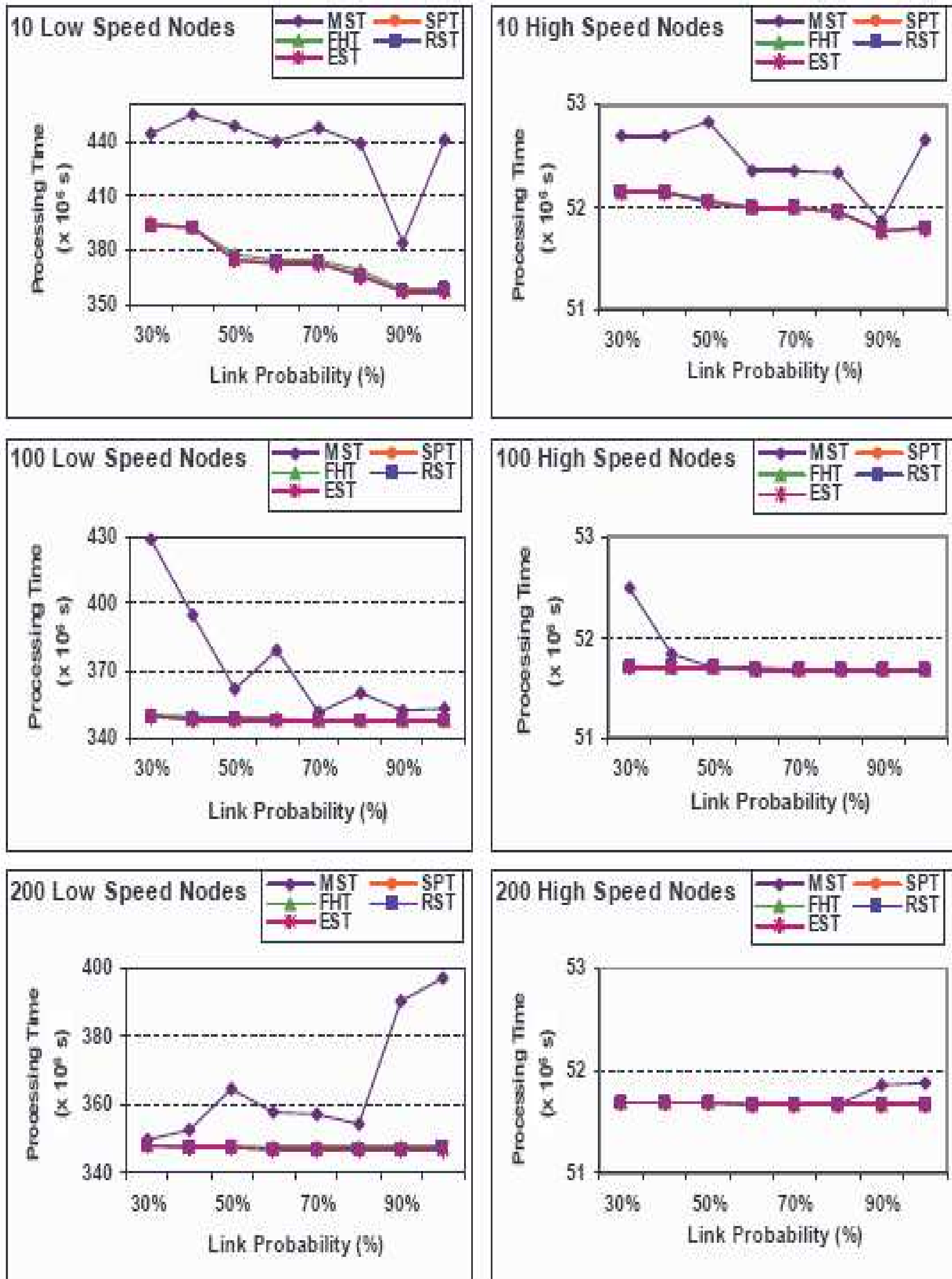


Figure 4.3: Total processing time simulation results for 10, 100, and 200 nodes with low and high processing speeds in a network with low speed links

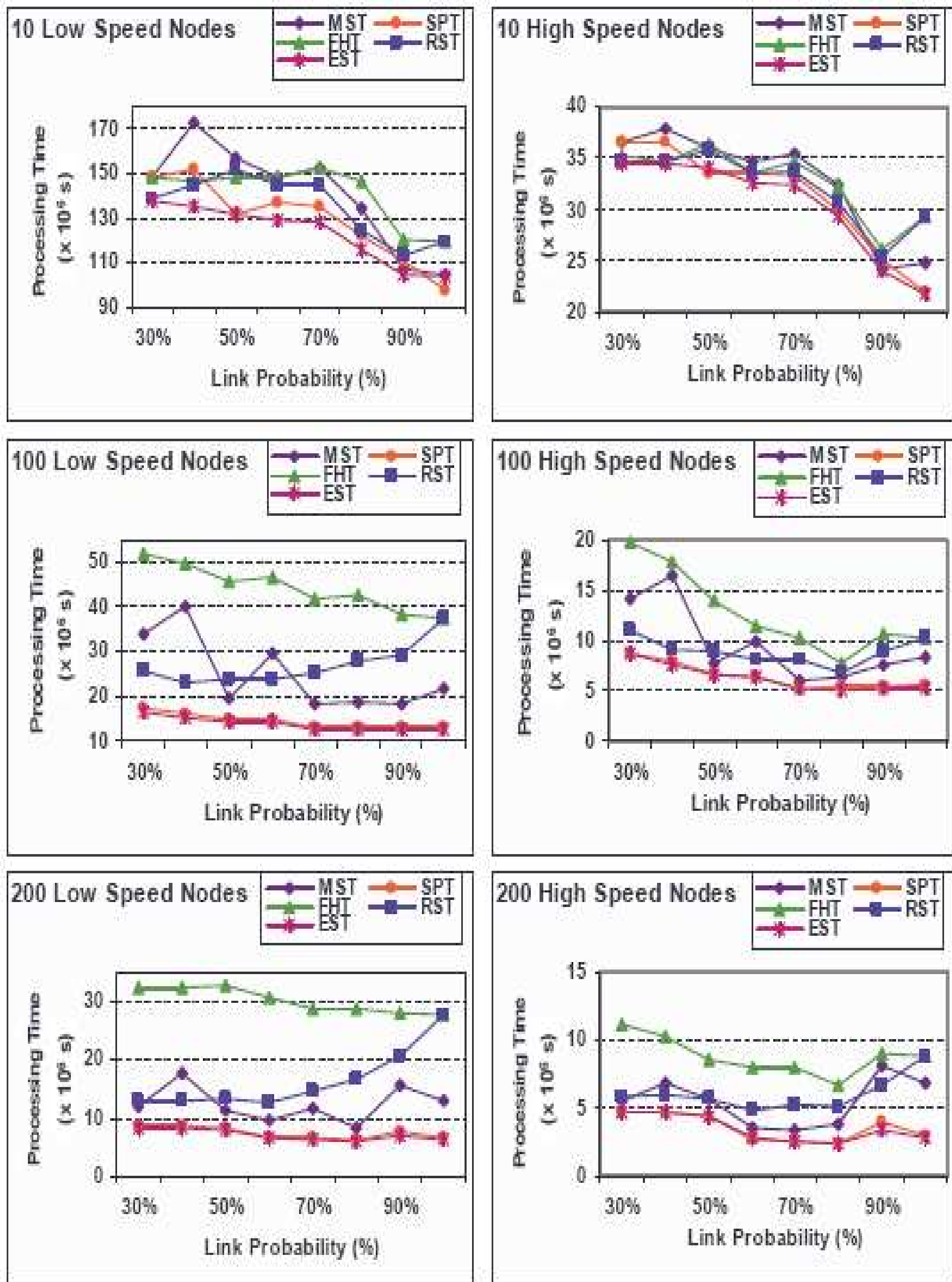


Figure 4.4: Total processing time simulation results for 10, 100, and 200 nodes with low and high processing speeds in a network with high speed links

number of hops), while FHT has the smallest eccentricity values and RST always has a value close to FHT, as shown in Figure 4.2. When the network size (the number of processing nodes) increases, the eccentricity value for MST also increases, whereas the eccentricity value of FHT almost remains unchanged. Both in the low and high link speed networks, the eccentricity values for SPT, FHT, RST, and EST remain almost identical as the number of nodes in the network increases beyond 100.

Further, from the Figures 4.3 and 4.4, it is observed that the total processing time decreases when the processing speed of node increases or network size increases. However, the diminishing effect of the increase of network size is observed, that is as the network size becomes larger and larger, the decrease in total processing time gets smaller and smaller. For example, in both low link speed networks and high link speed networks, when the network size increases from 10 to 100 nodes, the total processing times for SPT, FHT, RST, and EST routing strategies decrease much larger compared to the decrease in total processing times when network size increases from 100 to 200. In low link speed networks, MST and EST are observed to produce upper and lower bounds respectively for the total processing time, where as in the high link speed networks (except for the very sparse network with just 10 nodes) FHT and EST are observed to produce the upper and lower bounds.

#### 4.2.3.2 Effect of network connectivity

In order to analyze the effect of network connectivity, we compare the performance of the routing strategies by varying the link density values for a given number of processing nodes. When the link density value ($P_{link}$) is increased for a given network,

there are more links between processing nodes and hence there are more options available for the spanning tree routing strategies.

From the Figure 4.2, it is observed that the eccentricity values for SPT, RST, EST, and FHT tend to decrease slightly, and those for MST vary significantly as the $P_{link}$ increases in both low and high link speed networks.

Our simulations (Figure 4.3 and 4.4) show that the total processing time of an MST varies significantly as $P_{link}$ increases in a low link speed networks, whereas the variations are minor in high link speed networks. Compared to MST, the variations in the processing time for SPT, FHT, RST, and EST trees as $P_{link}$ increases are smaller. Also, the processing times in the low link speed networks for SPT, FHT, RST, and EST trees are almost identical for networks larger than 100 nodes. It is also observed that the total processing time for RST is closer to SPT values for low $P_{link}$ values and tends to move closer to FHT values as $P_{link}$ increases.

#### 4.2.3.3 Comparison of complexity and performance of algorithms

Given an arbitrary graph $G = \langle N, E \rangle$, using Fibonacci heap, an MST and SPT could be constructed in $O(E + NlogN)$ steps. The complexity of BFS to construct FHT is $O(|E| + |N|)$. The complexity of constructing RST is $O(E^2)$. Assuming that there are $m$ processors in every sub-tree and that there are $R$ sub-trees in every level, with a total number of $Q$ levels in the entire tree network, the complexity to compute an equivalent processor value and construct an $G_{opt}^{os}$ is given by $O(RQ + RQlog(m))$. Since, $m \leq N$, $R \leq N$, $Q \leq N$, $N \leq NlogN$, and $RQ \leq RQlog(m)$, the total

complexity of RAOLD-OS shall be approximated as $O(N^2 logN)$, for MST, SPT, and FHT and $O(E^2+N^2logN)$ for RST routing strategies. The construction of EST takes about $O(NERQ+NERQlog(m))$ steps. Hence, under similar assumptions, the total complexity of RAOLD-OS with EST shall be approximated as $O(E \cdot N^3 logN)$.

The complexity and time performance comparisons are summarized in the Table 2. In general, it is seen that EST provides the lowest processing time among all the routing strategies. However its complexity increases with number of nodes as well as number of links in a network. On the other hand, SPT provides comparable time performance to EST, while having far less complexity. The time performance of RST lies between that of SPT and FHT, and it provides robustness when there are link failures. MST seems to be the last option for divisible load scheduling in both low and high link speed networks. It is also seen that the eccentricity of FHT is the lowest and RST is comparable to that of FHT. The eccentricity of SPT and EST are slightly higher than FHT but much lower than that of MST.

In the case of low link speed networks, the processing time performance of all the spanning trees except MST are similar, but the complexity of FHT and SPT are lower than that of RST and EST. Hence, FHT and SPT are better routing strategies for divisible load scheduling in low link speed networks.

In the case of high link speed networks, EST and SPT seem to provide a better performance in terms of total processing time; and their trees are neither as "skinny" as MST nor as "fat" as FHT or RST. However, the performance degradation of RST is minimal for large network sizes as long as the link densities are moderate. Hence, SPT

Table 4.2: Comparison of complexities[1] and performances of various spanning tree algorithms for divisible load scheduling with RAOLD-OS scheduling strategy for arbitrary graphs

| **Spanning tree algorithm** | **Complexity** | **Performance** | | |
|---|---|---|---|---|
| | | **Processing time** | | **Eccentricity** |
| | | **Low speed links** | **High speed links** | |
| EST | $O(E \cdot N^3 logN)$ | Low | Lowest | Medium |
| FHT | $O(N^2 logN)$ | Low | Highest | Lowest |
| MST | $O(N^2 logN)$ | Highest | Medium | Highest |
| RST | $O(E^2 + N^2 logN)$ | Low | Medium | Low |
| SPT | $O(N^2 logN)$ | Low | Lowest | Medium |

is a better routing strategy for divisible load scheduling in high link speed networks.

Overall, SPT is shown to provide the best trade-off between time performance and complexity in both low and high link speed networks. This is a very useful characteristic of SPT, as in the next chapter we will use SPT to schedule multi-source divisible loads in arbitrary networks. Notice that if robustness against link failure is desired, RST may be the better option .

# Chapter 5

# Scheduling Multi-source Divisible Loads in Arbitrary Networks

## 5.1 General Introduction of the Presented Problem: Scope, Network Model and Problem Formulation

As we mentioned in the previous chapter, scheduling multi-source divisible loads on a arbitrary network is quite a challenging task as different sources should cooperate and share their computing power with others to balance their loads and minimize total computational time. Besides that, since the underlying network has an arbitrary topology it is difficult to decide from which source and which route a processing node should receive loads. Further processing nodes may be allocated to different sources when they become available. Because of the complexity, this problem has not been rigorously addressed in the literature, even for the resource aware case (i.e., link and processor speeds are known a priori).

Therefore, in this chapter we attempt to design and analyze multi-source divisible load scheduling strategies on arbitrary networks within the DLT domain, starting

from the resource aware case. We consider two different cases of interest, the static case and the dynamic case. In the static case, we assume no new loads will arrive at the system, while in the dynamic case, new loads may arrive as time progresses. To address the scheduling issue, we propose a novel *Graph Partitioning Scheme* (GP), which partitions the network into several totally disjoint regions. We then propose two novel strategies, which are referred to as *Static Scheduling Strategy* (SSS) and *Dynamic Scheduling Strategy* (DSS), one for each case. Both strategies use GP to partition the network, and balance the loads in an iterative fashion. We study the performance of these strategies both analytically and through simulation.

In this study, we also show that by using a simple reporting scheme, the presented algorithms can be easily adapted to the resource unaware case. Further, as we mentioned in the previous chapter, in the reporting scheme, a processing node can report back to the source nodes not only the speed information, but also from which source(s) and which route(s) the processing node should receive its loads. This solves a fundamental issue in scheduling multi-source divisible loads on arbitrary networks. Below, we will present the network model and how the problem is formulated.

### 5.1.1 Network Model and Problem Formulation

The network considered in this chapter is an arbitrarily connected network comprising a total of $m$ source processors, to which users submit loads for processing. These source processors (or simply "sources") will share the loads either with the entire network or a portion of the network. We denote them as $S_0, S_1, ..., S_{m-1}$. Besides

the sources, we assume that there are another $n$ processors (To distinguish them from the sources, they are referred to as "processing nodes" below.) in the network. These processing nodes can receive loads from any source, and we denote them as $P_m, P_{m+1}, ..., P_{m+n-1}$.

We make the following assumptions in our formulation. First, both sources and processing nodes are allowed to participate in the computation process, and processing nodes can perform routing functions. As pertaining with the previous chapter, "with front-end" and "single port" are assumed for all processors. We also assume that sources can share information with each other through messages, and we neglect any overheads incurred by transmitting such short messages. Further, a linear cost model for communication and computation is adopted as in the literature.

Now, the problem can be stated as: *Given an arbitrary graph $G =< V, E >$, with $V = m + n$, where $m$ equals the number of sources and $n$ equals the number of processing nodes, how do we schedule and process loads submitted by the source nodes in the system such that the total processing time is minimized.*

We consider two distinct cases of interest - the static case and the dynamic case. For the static case we assume that in the network there are $m$ divisible loads $L_0, L_1, ..., L_{m-1}$ residing on $m$ sources, respectively. We assume that no additional loads will arrive. For the dynamic case, we assume each source has an independent load in-flow. Therefore, new loads may be expected to arrive at any point in time and the network should accommodate the new arrivals dynamically.

Two different approaches [86] are possible to tackle this problem. One is based

on “superposition” wherein, all or part of the processing nodes will receive multiple fractions of loads from different sources and the total load that each processing node received will be balanced according to its computation capacity. The other approach is referred to as “network partitioning” wherein, the entire network will be partitioned into several non-overlapping regions centered at each source respectively, and each source will only dispatch its load to its own region. Both techniques have advantages and disadvantages. Under “network partitioning”, since the entire network is partitioned into non-overlapping regions, each source can carry out load dispatching separately without interfering with each other. However, the challenge lies in partitioning the network into regions where each region's equivalent computation power is exactly proportional to this region's load size. In most cases we cannot strike a perfect balance across the network. On the other hand, under the “superposition” technique, each processing node can receive loads from several sources, and hence it is easier to balance the load across the network. However, because of communication contention problems, exercising control of “superposition” is much more complicated than “network partitioning”, and may induce large overheads.

In the divisible load context, all load fractions are homogenous, and a node that receives multiple fractions of load from different sources can receive a single fraction of load, which equals the summation of the multiple fractions, from the “nearest” source. Therefore, we adopt “network partitioning” technique[1] to schedule and process the loads.

[1]Since in our problem context the network has an arbitrary graph topology, “network partitioning” is actually “graph partitioning”.

However, designing an efficient strategy to partition the graph into several regions such that each region's optimal *equivalent computation power* is proportional to the region's load size is very difficult. The reason is that one must first solve a (equivalent) sub-problem : *Given a region, how do we identify this region's optimal equivalent computation power?* To identify a region's optimal equivalent computation power, one should find the best/optimal spanning tree first, because the a region's computation power is denoted by the equivalent computation power of its optimal spanning tree for the single-instalment based divisible load scheduling problem. Unfortunately, as we mentioned in the previous chapter, this problem is proven to be NP-hard [52].

In this study, we will demonstrate that our proposed GP works efficiently, in the sense that it partitions the network into several regions and generates a shortest path spanning tree (SPT) for each region simultaneously. Remember that, in Chapter 4 SPT is shown to offer the best trade-off between time performance and complexity among the commonly used spanning tree strategies. Our two algorithms, SSS for the static case and DSS for the dynamic case, will use GP to partition the network and then schedule the loads.

## 5.2 Static Scheduling Strategy (SSS)

Now suppose the speed information is known to all sources in the network, let us see how the GP works. For the ease of presentation, we define an *ordered communication delay 2-tuple* $(C_{ki}, i)$ which captures the cumulative communication delay from processing node $P_k$ to source $S_i$. As there are $m$ sources in the network, each processing

node has $m$ 2-tuples. We can define their relations as follows. When $C_{ki} > C_{kj}$, $(C_{ki}, i) > (C_{kj}, j)$. When $C_{ki} = C_{kj}$, then $(C_{ki}, i) > (C_{kj}, j)$, iff $i > j$. Since each source has a unique subscript, according to our definition, each processing node can locate a unique source which has smallest 2-tuple (i.e. smallest cumulative communication delay). We denote this source as target source to the corresponding processing node. Further, we also define $\pi_{ij}$ as the shortest path (in term of communication delays) from $S_i$ (or $P_i$) to $S_j$ (or $P_j$)

**Graph Partitioning Scheme:** In our approach, the given graph is divided into $m$ regions $Region_0$, $Region_1$, $...Region_{m-1}$, centered at $S_0, S_1, ..., S_{m-1}$, respectively. An arbitrary processing node $P_i$ is attached to its target source by the shortest path (path with smallest communication delay). We observe that GP intuitively uses each processing node effectively. This is because what determines the real computation capacity of a node in a network is not only the computation speed of this processor, but also its communication delay to the source. In GP, all the processing nodes are attached to their target sources by the shortest path, and hence from the processing nodes perspective they have been used efficiently. We define the following.

*Totally disjoint regions*: Totally disjoint regions mean that any two regions have (*i*) no common node, (*ii*) no common link, and (*iii*) no intersection. Notice that (*i*) and (*ii*) do not imply (*iii*). Even without common nodes and links, two regions may still intersect with each other. For example, suppose $P_x$ belongs to $Region_i$, and $P_y$ belongs to $Region_j$ respectively. However, $P_y$ may be connected to $S_j$ through $P_x$ (i.e. $P_x$ is an intermediate node of $\pi_{jy}$), so $Region_i$ and $Region_j$ still intersect with

each other. Thus, for two regions to be totally separable/disjoint, they must satisfy ($i$), ($ii$), and ($iii$) simultaneously. Now we state the following.

**Theorem 1** Using GP the graph is divided into $m$ totally disjoint regions. **Proof:** In order to realize the proof we proceed as follows. We realize that ($i$) is immediately apparent, since every processing node has a unique smallest ordered communication delay 2-tuple. However, to complete the proof including ($ii$) and ($iii$), we need to prove the following lemma first.

**Lemma 1** *Suppose $P_i$ is attached to $S_j$, and $\pi_{ij}$ is the shortest path (with respect to communication delay) from $P_i$ to $S_j$. Then, all the processing nodes belonging to $\pi_{ij}$ are also attached to $S_j$.*

*Proof:* The proof is by contradiction. Suppose one processing node $P_k$ belongs to $\pi_{ij}$ and is not attached to $S_j$, but is attached to another source $S_x$. Thus, according GP, for $P_k$, we have $(C_{kj}, j) > (C_{kx}, x)$. Notice that the path from $P_x$ to $P_i$ has a constant communication delay, denoted as $C_{\pi_{xi}}$. Then, for $P_i$, we have $(C_{kj}+C_{\pi_{xi}}, i) > (C_{kx} + C_{\pi_{xi}}, x)$, i.e., $(C_{ij}, j) > (C_{ix}, x)$. This means that $P_i$ should also be attached to $S_x$, which contradict our assumption. Therefore, Lemma 1 is proved.

Next, we use Lemma 1 to complete the proof of separability of the regions including ($ii$) and ($iii$) of Theorem 1. According to GP, each processing node is attached to its target source by the shortest path. Suppose two paths $\pi_1$ and $\pi_2$, which belong to two different regions $Region_i$ and $Region_j$ respectively, intersect with each other at $P_k$. Then, from lemma 1 we know $P_k$ belongs to $Region_i$ and also $Region_j$, which is impossible. Therefore, any two paths belonging to two different regions have no

intersection, and hence any two regions have no intersection or common links. Hence the proof. ∎

From the above proof, we know that by using GP, the graph can be divided into $m$ disjoint regions. Further, since in GP all processing nodes in the same region are attached to the corresponding source by the shortest path, we automatically generate one SPT for each region. This is a very useful characteristic of GP, since the optimal solution of scheduling divisible load for an arbitrarily connected graph occurs on a spanning tree of the graph. In our context, after applying GP, each source can directly dispatch load to its SPT, using a similar RAOLD-OS strategy [51]. Therefore, GP actually performs two tasks together. It effectively divides the graph into $m$ disjoint regions and at the same time it generates one SPT for each region.

Now we will introduce how SSS works. SSS progresses in an iterative fashion. At the beginning of the first iteration, SSS will apply GP to partition the graph. After the network is partitioned into $m$ totally disjoint regions, each source will compute the equivalent computation power for its own region based on the SPT and this process of obtaining an equivalent computation power is described in Chapter 3. Notice that since we do not take into account each source's load size when we partition the network, each region's "equivalent computation power" is not proportional to this region's load size. We may expect that regions will complete processing their respective loads at different time instants. Therefore, in the first iteration, to balance the computation power across all the regions, the amount of load that will be consumed for processing by a region will be altered proportionally. Suppose

when each source dispatches and processes its load only on its own region, the finish time of $L_0, L_1, ..., L_{m-1}$ are $T_0, T_1, ..., T_{m-1}$ respectively, and suppose $i = argmin\{T_j\}$, i.e., $Region_i$ has the smallest finish time. To achieve balance, any region other than $Region_i$, say $Region_j$, will only consume $L_j T_i / T_j$ amount of load in the first iteration, using a similar RAOLD-OS strategy.

In the first iteration, the whole $L_i$ is consumed by $Region_i$, and hence no load remains in $S_i$. However, other sources will have some amount of load remaining, and the amount of load $L'_j$ remaining with source node $S_j$ is,

$$L'_j = L_j * \frac{T_j - T_i}{T_j} \tag{5.1}$$

Therefore, the second iteration will start with $m - 1$ remaining loads $L'_0, ...L'_{i-1}, L'_{i+1}, ..., L'_{m-1}$ residing on $m - 1$ sources $S_0, ...S_{i-1}, S_{i+1}, ..., S_{m-1}$, respectively. Then, SSS will apply GP again to partition the graph into $(m - 1)$ regions. Notice that this process actually is a reallocation of processing nodes which originally belong to $Region_i$, to other regions. Those processing nodes which belong to a region other than $Region_i$ remain in that region. As in the first iteration, the region with smallest finish time will consume the whole remaining load, while other regions will only consume a proportional amount of load, and hence the third iteration will start with $(m-2)$ sources and remaining loads. Obviously, SSS will come to a halt after $m$ iterations. Further, as long as a region is busy, its equivalent computation power will not decrease. Thus, in SSS the processing of load $L_i$ will complete within $T_i$. The total processing time $T_{sss}$ of the entire network, defined as time difference between the start time and the time instant when the last remaining load has been processed,

is,

$$T_{sss} \leq \max\{T_i, i = 0, 1, ..., m-1\} \tag{5.2}$$

We observe two issues here. First, in SSS, within each iteration, "network partitioning" technique is used to dispatch and process the loads. However, when we look at the entire process, a processing node may receive loads from different sources, and hence SSS also has an "superposition" characteristic. Therefore, SSS can be viewed as having a "hybrid" property.

Secondly, when implementing GP, it can either be the sources that can initiate the processing or the processing nodes[2]. In a source initiating scheme, each source will construct a shortest path spanning tree simultaneously, using Dijkstra's algorithm. Then, all the sources share information with each other, and hence each source can identify the processing nodes which have the smallest communication delay to itself. On the other hand, if processing nodes initiate the algorithm, each processing node will simultaneously compute its shortest path weight (communication delay) to each source using Dijkstra's algorithm or Bellman-Ford algorithm [93], and then choose the target source and report the shortest route to the target source. To reduce redundant computation, initially, each processing node can maintain a list of shortest paths and their weights to each source. Then, as long as a source completes its load, its processing nodes (nodes within its region) can quickly identify the next source it should be attached to, and hence reduce overheads.

[2]In the literature, these are commonly referred to as sender initiated and receiver initiated approaches.

### 5.2.1 Adapting to Resource Unaware Case

The SSS algorithm is easily adaptive to the resource unaware case using a reporting scheme. Notice that Bellman-Ford algorithm can be implemented in a distributed way, in which each node only needs to know the local link speed information (i.e., the speed of the links that directly connect to this processing node) to construct a shortest path spanning tree. This makes the Bellman-Ford algorithm naturally suitable for a reporting based SSS (RB-SSS) algorithm under the resource unaware environment.

In the RB-SSS, all sources will signal the network at the beginning (i.e., t=0), and then all processing nodes start to run the Bellman-Ford algorithm. Several important issues should be noted here. First, several Bellman-Ford algorithms (equals the number of sources) run concurrently in the network, but they can be easily distinguished by adding corresponding identifiers during implementation. Secondly, some processing nodes may already possess the local speed information, and they can immediately run the Bellman-Ford algorithm with this information. Other processing nodes which are not aware of the local speed information need to detect it. However, the link speed can easily be detected by a local processing node rather than by a possibly far away source. Further, notice that Bellman-Ford algorithm is actually a step-wise updating algorithm and there is no negative circles in our case. Therefore, the processing nodes can temporarily set the unknown speed to be infinite and run the Bellman-Ford algorithms immediately after receiving the signal from the sources. In this way, the distributed Bellman-Ford algorithms can start without delay. When a processing node detects a link speed which is originally unknown, it can update the

speed of the link from infinity to its actual value[3]. As we mentioned in the previous chapter, the sources can set up a threshold time $T^*$. The Bellman-Ford algorithm will stop at $T^*$, and all processing nodes will report back the routes and the speeds to sources based on the current detected speed information. Those processors/links whose speeds are still undetected at $T^*$ are normally very slow. Excluding them from computation will not influence the total performance too much. With the reported information, sources are able to schedule the loads. Notice that GP is actually integrated in this process.

Compared to the resource aware case, both SSS and RB-SSS start GP by running the spanning tree algorithm at the beginning (distributed Bellman-Ford in resource unaware case, and Dijkstra or Bellman-Ford in resource aware case.), and they are identical afterwards. SSS may run the spanning tree algorithm faster than RB-SSS, as in SSS all speeds are known in advance. In RB-SSS processing node may need to detect the local link speeds and its own speed, which will induce overheads. However, the overhead is simply a constant and bounded by $T^*$ before the real scheduling. Therefore, in our simulation test, we focus on assessing the performance of SSS.

## 5.3 Dynamic Scheduling Strategy (DSS)

Now we tackle a more realistic situation wherein each node is more independent in its operation and each source has an independent and dynamic load in-flow. In this case, a previous idle source may have new arrival loads and hence becoming busy,

[3] Since implementation is not a major concern of this thesis, we will not go into the implementing details.

while a busy source may become idle. Therefore, in DSS, at the beginning of each iteration, those sources which have changed their status will inform other sources by sending short messages. Notice that each time only one busy source may become idle, but more than one idle source may become busy. Then sources having loads to process will apply GP to partition the network, and similar to SSS, the region with the smallest finish time will consume the entire load, while other regions will only consume a proportional amount of load. After the current iteration, the sources will repeat the above process for every load that arrives to the system. Notice that DSS can also be adapted to the resource unaware case, by adding a reporting phase at the beginning. This will be a constant and one-time only overhead.

There are two major concerns here. First, a newly arrived load at $S_i$ will be stored in the buffer until all previous loads in $S_i$ have been processed. Secondly, unlike in SSS, where each active region's "equivalent computation power" will only increase, in DSS, it may also decrease. This is because a new load may arrive at a previously idle source, and this source will re-claim the resource which initially belongs to its region at the beginning of the next iteration. Therefore, the "equivalent computation power" of a currently active region fluctuates as time progresses.

Although in DSS each region's "equivalent computation power" fluctuates, we can still attempt to derive the *upper bound* of processing time of a given load[4]. When all sources in the network are busy, each region will occupy certain "domains" in the network. We refer to such domains as the "critical domains" to the corresponding

[4]Time difference between the instant at which the load is scheduled by the source and the time instant at which the load has been completed.

regions. This is because for any source, its "critical domain" will always be attached to it, as long as this source is busy. We denote the "equivalent computation power" of $Region_i$'s "critical domain" as $E_i^c$. Further, since we adopt a linear cost model, the processing time of a load is linearly related to the load size. Therefore, suppose the load $L_i$ at $S_i$ is processed with $k$ installments $L_i^0, L_i^1, ..., L_i^{k-1}$, and the average computation power is $\bar{E}_i$, we have

$$\begin{aligned} T(\bar{E}_i, L_i) &= T(\bar{E}_i, L_i^0 + L_i^1 + ... + L_i^{k-1}) = T(\bar{E}_i, L_i^0) + T(\bar{E}_i, L_i^1) + ... + T(\bar{E}_i, L_i^{k-1}) \\ &\leq T(E_i^c, L_i^0) + T(E_i^c, L_i^1) + ... + T(E_i^c, L_i^{k-1}) = T(E_i^c, L_i) = E_i^c L_i \end{aligned} \quad (5.3)$$

where, $T(E, L_i)$ denotes the processing time of load $L_i$ with computation power $E$.

Notice that Eqn (5.3) gives an upper bound of one load's processing time. However, a load may not be able to be processed immediately when it arrives and hence, the actual time the load spends in the network may be longer. Further, since a newly arrived load will be stored in the buffer until its previous loads have been processed, each source should have adequate buffer space to hold new arrival loads. Therefore, it will be more appropriate if we perform queueing analysis for understanding the performance of DSS in the next section.

## 5.4 Analysis of DSS

Suppose each source has independent poisson arrival loads, and the arrival rate at $S_i$ is $\lambda_i$. Further, we assume that the load size is exponentially distributed with parameter $\mu_i^c E_i^c$. Notice that when a region has fixed computation power, the processing time

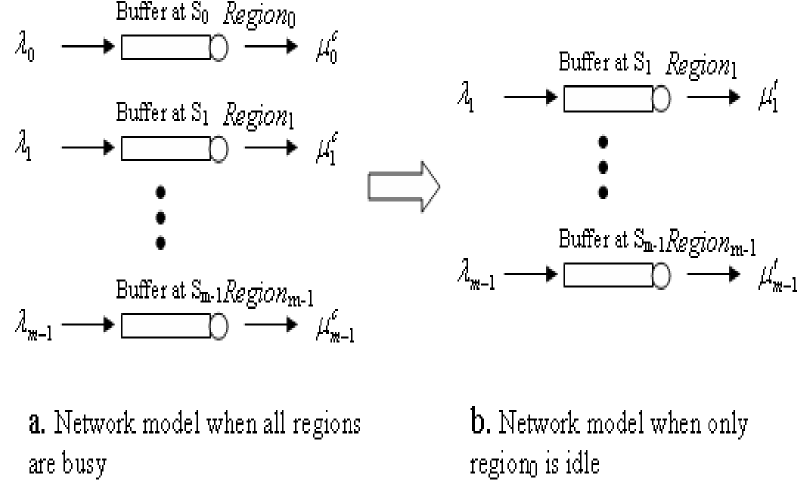


Figure 5.1: Network models

of the loads is also exponentially distributed. Therefore, when all regions are busy (and thus each region is processing its load within its critical domain only), we can map each region to a M/M/1 queue, and the service rate for $Region_i$ is $\mu_i^c$, as shown in Figure 5.1.(a).

However, in a dynamic situation, the service rate for each region is not constant. At any time instant, several regions may become idle and their computation power will be reallocated to other regions. For instance, when $Region_0$ is idle while other regions are busy, the network model is shown in Figure 5.1.(b). Notice that the arrival rate $\lambda$ is the same for the above two cases, but $\mu_i\prime = \frac{E_i'}{E_i^c}\mu_i^c$, where $E_i^c$ denotes the equivalent computation power of $Region_i$'s critical domain and $E_i'$ denotes $Region_i$'s equivalent computation powers when $Region_0$ is idle. Actually, there are $2^m$ different cases, where $m$ is the number of sources. In all cases, for a given region, arrival rates remain the same, but service rates are different. Notice that among all cases, when $S_i$ occupies the entire network (i.e. when all other regions are idle), $Region_i$ will have

the maximum service rate. We denote the maximum service rate for $Region_i$ as $\mu_i^{max}$.

Now, we consider the entire network as a system with $m$ in-flows of loads and the system will process these loads at a certain rate. However, if the aggregated load in-flow rate exceeds the service rate of the entire system, then system becomes unstable, and the number of loads left in the system will increase to infinity as time progresses. Therefore, the key question to address is as follows: "*Given a network, if we know the load arrival rate as well as the load size distribution at each source (i.e. $\lambda_i$ and each region's service rates in different cases are known), how can we decide whether this network can manage these loads or not, and what is the average queue length at each source and the average waiting time of a load?*". To address these questions, we will analyze our network for three important cases.

**Case 1:** $\forall\ i, \lambda_i < \mu_i^c$. This is considered as a stable case wherein arrival rates are less than the processing rates, which implies that each source can easily manage its own loads using its critical domain only.

According to DSS, an idle source node $S_i$ will release its computation capacity. Any load arriving during $S_i$'s idle time will wait for a certain amount of time until $S_i$ regains its computation capacity, and then $S_i$ will start to dispatch and process the load. Therefore, we can map $Region_i$ to a M/G/1 queue with vacations [5] [94]. Let $\zeta_i$ denotes the distribution of service time at $Region_i$, and $\nu_i$ denotes the distribution of $Region_i$'s vacation time. Then the average waiting time of loads at $S_i$'s buffer

[5] When all other regions are idle, the vacation time can be viewed as arbitrarily small.

(denoted as $T_i^w$) is given by,

$$T_i^w = \frac{\lambda_i E[\zeta_i^2]}{2(1-\lambda_i E[\zeta_i])} + \frac{E[\nu_i^2]}{2E[\nu_i]} \quad (5.4)$$

However, in our context, each region's service rate and vacation time are coupled with other regions in the network, and hence, it is more appropriate to view the entire network as $m$ coupled M/G/1 queues with vacations. In this case it may be noted that determining the exact value of $T_i^w$ is very difficult. To see this, consider the simplest case where there are only two regions ($Region_0$ and $Region_1$) in the network. To compute $E[\zeta_i]$, $E[\zeta_i^2]$, $E[\nu_i]$, and $E[\nu_i^2]$ $(i = 0, 1)$ in Eqn (5.4), one needs to know the probability when $Region_0$ is busy while $Region_1$ is idle, the probability when $Region_0$ is idle while $Region_1$ is busy, the probability when both regions are busy, and the probability when both regions are idle. This requires us to solve an infinite 3-dimensional Markov Chain, as shown in Figure 5.2. In the figure, state $(ij)$ denotes $i$ loads in $S_0$ and $j$ loads in $S_1$, while both $S_0$ and $S_1$ are busy. State $(i\prime j)$(or $(ij\prime)$) denotes there are $i$ loads in $S_0$ and $j$ loads in $S_1$, while $S_0$ is idle(or busy) and $S_1$ is busy(or idle).

The Markov Chain shown in Figure 5.2 is very complicated to solve. Further, the complexity of the problem increases dramatically as the number of sources increases and makes it complex to derive an exact value of $T_i^w$. Thus, in this study, we will attempt to derive the upper bound on $T_i^w$. In the derivation, we will use an important property of the exponential distribution – the combination property [94], which is stated as follows.

**Theorem 2:** Random variables $x_1, x_2, ..., x_k$ are exponentially distributed with

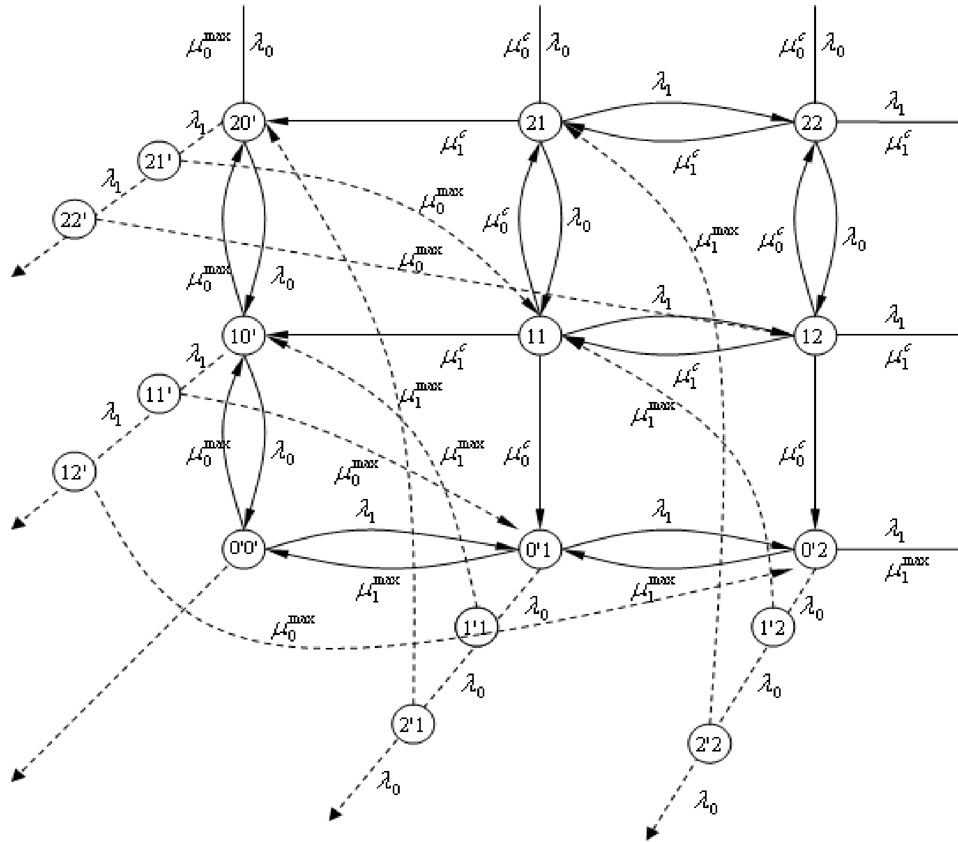


Figure 5.2: Markov chain of two regions case

parameters $u_1, u_2, ..., u_k$, Let random variables $Y = \min\{x_1, x_2, ..., x_k\}$. Then, $Y$ is also exponentially distributed with parameter $u = u_1 + u_2 + ... + u_k$.

Using Theorem 2, we can identify the distribution of $\nu_i$ and find the worst case $E[\nu_{iworst}]$ and $E[\nu_{iworst}^2]$. Suppose when $S_i$ becomes idle, there are $k$ busy regions in the network. Because of the memoryless property of the exponential distribution [94], all the $k$ regions' remaining processing time of their loads are exponentially distributed with parameters $\mu_0, \mu_1, ..., \mu_{k-1}$. It may be noted that $Region_i$'s vacation time is the minimum remaining processing time among the $k$ busy regions. According to Theorem 2, we know the $Region_i$'s vacation time $\nu_i$ is also exponentially distributed, with parameter $\mu = \mu_0 + \mu_1 + ... + \mu_{k-1}$. Notice that as $k$ becomes smaller (i.e., less regions are active), $\mu$ also becomes smaller. Therefore, when only the region with smallest $\mu_j^{max}$ is active, $Region_i$'s vacation time distribution has the smallest value $\mu = \mu_j^{max} = \min\{\mu_0^{max}, \mu_1^{max}, ..., \mu_{i-1}^{max}, \mu_{i+1}^{max} ..., \mu_{m-1}^{max}\}$. In this case, $Region_i$ has the largest average vacation time, and hence, loads at $S_i$ have largest waiting time in the buffer. We have,

$$E[\nu_{iworst}] = \frac{1}{\mu_j^{max}} \tag{5.5}$$

$$E[\nu_{iworst}^2] = \frac{2}{(\mu_j^{max})^2} \tag{5.6}$$

Further, we notice that as long as $Region_i$ is busy, its service rate is larger than or equal to $\mu_i^c$. Since the loads waiting time is inversely related to the $Region_i$'s service rate, the upper bound of $T_i^w$ is,

$$T_i^w = \frac{\lambda_i E[\zeta_i^2]}{2(1 - \lambda_i E[\zeta_i])} + \frac{E[\nu_i^2]}{2E[\nu_i]} \leq \frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}} \tag{5.7}$$

Then, applying *Little's Theorem* [95], we can derive the average number of loads in the $S_i$'s buffer (denoted as $Num_i^{ave}$), which is,

$$Num_i^{ave} = \lambda_i T_i^w \leq \lambda_i (\frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}}) \tag{5.8}$$

Since load size at $S_i$ is exponentially distributed with a parameter $\mu_i^c E_i^c$, the average load size is given by $\frac{1}{\mu_i^c E_i^c}$. Thus, the average queue length at $S_i$ (denoted as $Q_i^{ave}$) is bounded by,

$$Q_i^{ave} = Num_i^{ave}/E_i^c \mu_i^c \leq \lambda_i (\frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}})/E_i^c \mu_i^c \tag{5.9}$$

Eqn (5.9) gives us considerable hints on how much buffer should be assigned to each source when designing the system. To reduce the probability of dropping loads when the buffer is full, one should assign a larger buffer size than $Q_i^{ave}$ derived from Eqn (5.9) (say, two times larger than $Q_i^{ave}$), to $S_i$. However, since we have adopted some approximations to derive Eqn (5.9), in some cases $Q_i^{ave}$ may not give a tight estimation on real actual queue length. This behavior is carefully studied and discussed in the next section.

**Case 2:** $\forall\, i, \lambda_i \geq \mu_i^c$ **or** $\exists\, i, \lambda_i > \mu_i^{max}$. In this case, the network cannot manage these loads and is *critically stable*. The average queue length of the network and average waiting time of loads are expected to grow to infinity, as time progresses. Therefore, in this situation, one must reduce load arrival rates or discard low priority loads at one or more of the sources.

**Case 3:** $\forall\, i, \lambda_i < \mu_i^{max}$ **&** $\exists\, i, \lambda_i \geq \mu_i^c$ **&** $\exists\, j, \lambda_j < \mu_j^c$. This case is more difficult than the above two cases. In this case, some regions cannot handle their loads by using their critical domains only, but by "borrowing" computation power from other regions,

these regions may be able to handle their loads. Thus, the problem is "*whether the regions with excess resources can render enough computation power to other regions*". Obviously, addressing this problem is extremely complex. For this case, we attempt to study the simplest two regions case, which reveals some basic issues of the posed problem.

Consider that there are two regions in the network, with $\mu_0^c \leq \lambda_0 < \mu_0^{max}$ and $\lambda_1 < \mu_1^c < \mu_1^{max}$. Now, the key question is that whether $Region_0$ can borrow enough computation capacities from $Region_1$ to accommodate its excess loads. Consider the *boundary* situation, i.e., $Region_0$ can borrow "just enough" resources from $Region_1$. In this case $\lambda_0 = \bar{\mu_0}$, where $\bar{\mu_0}$ denotes $Region_0$'s average service rate. From $Region_1$'s perspective, it can be mapped to a M/M/1 queue with vacations, and vacation time is exponentially distributed with parameter $\mu_0^{max}$. Notice that vacation time will not affect the *idle-ratio* [6] of $Region_1$ (denoted as $R_1^{idle}$), and hence $R_1^{idle}$ is equal to the idle-ratio of a M/M/1 queue with the same service rate and arrival rate, but without vacations, which is,

$$R_1^{idle} = (\mu_1^c - \lambda_1)/\mu_1^c \tag{5.10}$$

From $Region_0$'s viewpoint, it has the maximum service rate $\mu_0^{max}$ in $R_1^{idle}$ ratio of time, and has a service rate of $\mu_0^c$ in the remaining $1 - R_1^{idle}$ ratio of time. Hence, the expectation of its service rate is,

$$\bar{\mu_0} = \frac{\mu_0^{max}(\mu_1^c - \lambda_1)}{\mu_1^c} + \frac{\mu_0^c \lambda_1}{\mu_1^c} \tag{5.11}$$

Therefore, if $\lambda_0 < \bar{\mu_0}$, this network is able to manage these loads. Otherwise, this

[6] Idle-ratio is defined as ratio of the time when the region is idle.

network cannot handle this amount of loads.

As in case 1, one can apply Eqn (5.9) to estimate the upper bound of $Q_1^{ave}$ (the average queue length at $S_1$), and as $\lambda_0$ approaches $\bar{\mu_0}$ (given by Eqn (5.11)), $Q_1^{ave}$ approaches the upper bound. When $\lambda_0 \geq \bar{\mu_0}$, $Q_1^{ave}$ is exactly equal to $\lambda_1(\frac{\lambda_1/\mu_1^c}{\mu_1^c-\lambda_1} + \frac{1}{\mu_0^{max}})/E_1^c\mu_1^c$ (as per Eqn (5.9)). For $Region_0$, when $\lambda_0 \geq \bar{\mu_0}$, $Q_0^{ave}$ goes to infinity. When $\mu_0^c < \lambda_0 < \bar{\mu_0}$, though we cannot directly use Eqn (5.9) to calculate the upper bound of $Q_0^{ave}$, we can use $\bar{\mu_0}$ instead of $\mu_0^c$ in Eqn (5.9) to estimate the approximate value of $Q_0^{ave}$.

Similarly, when there are more than two regions in the network, we have to compute how much computation power the regions with excess resource can borrow from other regions which cannot handle their loads alone. Unfortunately, solving this problem requires solving the similar Markov Chain[7] as shown in Figure 5.2. This remains an open problem.

## 5.5 Performance Evaluation and Discussion

### 5.5.1 Performance of SSS

As we mentioned above, in this section we focus on studying the performance of SSS (resource aware case), as in resource unaware case, there will be simply a one-time, constant and bounded overhead. We compare the performance of SSS with a strategy

[7]Similarly, the complexity of the Markov Chain increases dramatically as number of regions grows.

referred to as *Sequential Dispatching Strategy* (SDS)[8]. SDS works as follows. Consider a network with $m$ loads $L_0, L_1, ..., L_{m-1}$ residing on $m$ sources $S_0, S_1, ..., S_{m-1}$, respectively. In SDS, $S_0$ will first dispatch $L_0$ to the entire network based on a SPT of the network using a similar RAOLD-OS strategy, while other sources temporarily hold their loads. Then, after $L_0$ has been processed, $S_1$ will dispatch $L_1$ to the entire network. The above process continues until all loads have been processed.

As we can see from the above description, SDS is simple in nature. Further, we notice that if communication delay can be neglected (when all links in the network are sufficiently fast), SDS and SSS will have exactly the same performance. Suppose there are $m$ loads in the network, the total processing time $T_{total}$ for both SDS and SSS would be,

$$T_{total} = (L_0 + L_1 + ... + L_{m-1}) \cdot E(w)T_{cp} \tag{5.12}$$

where $E(w)$ is the equivalent computation capacity of the entire network. However, in the presence of communication delay, SSS and SDS will show different performance. We conduct experiments to study how SSS and SDS will react to communication delay. Our experiments reveal certain interesting characteristics of SSS.

In our experiments, the network has an arbitrary graph topology generated randomly with a specified number of nodes and link connectivity probabilities[9]. The computation speed parameters of sources and processing nodes $w$ [10] is uniformly distributed among $[1, 10]$, and both $T_{cm}$ and $T_{cp}$ are set to be 1. We simulate different

[8]As of this date, there are no multi-job strategies for scheduling divisible loads on arbitrary graphs in the literature.

[9]In our experiments, the link probability of a direct link between a pair of nodes is set to 0.4.

[10]The notations $w$, $z$, $T_{cp}$, and $T_{cm}$ are defined in Chapter 2

network scenarios (tightly-coupled and loosely-coupled) by assigning different distributions of the communication speed parameters. Further, we assume each source in the network has an amount of load $L = 10,000,000$. Later, we will also show the effect of load size on our strategies.

We first study networks with 20 nodes. To simulate the characteristics of a tightly-coupled network, $z$ is set to be uniformly distributed among $[0, 0.5]$, and to simulate the characteristics of a loosely-coupled network, $z$ is set to be uniformly distributed among $[1, 2]$. We vary the number of source nodes[11] in the network from 1 to 10, and the corresponding total processing time of SSS and SDS is shown in Figure 5.3.(a) and Figure 5.3.(b). From these figures, we observe that SSS outperforms SDS, and when the communication delay is large, SSS gains a significant speedup against SDS. This is expected since in the presence of communication delays, SSS utilizes the computation power of sources and processing nodes much more efficiently than SDS.

Further, we notice that the total processing time of SDS is approximately linearly related to the number of sources in both loosely-coupled networks and tightly-coupled networks, as shown in Figure 5.3.(a) and Figure 5.3.(b). SSS also exhibits the similar linear relationship in tightly-coupled networks. However, SSS shows a very interesting characteristic in loosely-coupled networks. As shown in Figure 5.3.(b), provided the number of sources is smaller than some threshold, increasing the number of sources (i.e., increasing the number of loads in the network) does not affect the total processing time significantly. Actually, as the number of sources increases from 1 to 7 (i.e. total amount of loads increases by 600%), the total processing time of SDS increases by

[11] Each time new source nodes are randomly generated, while previous source nodes are retained.

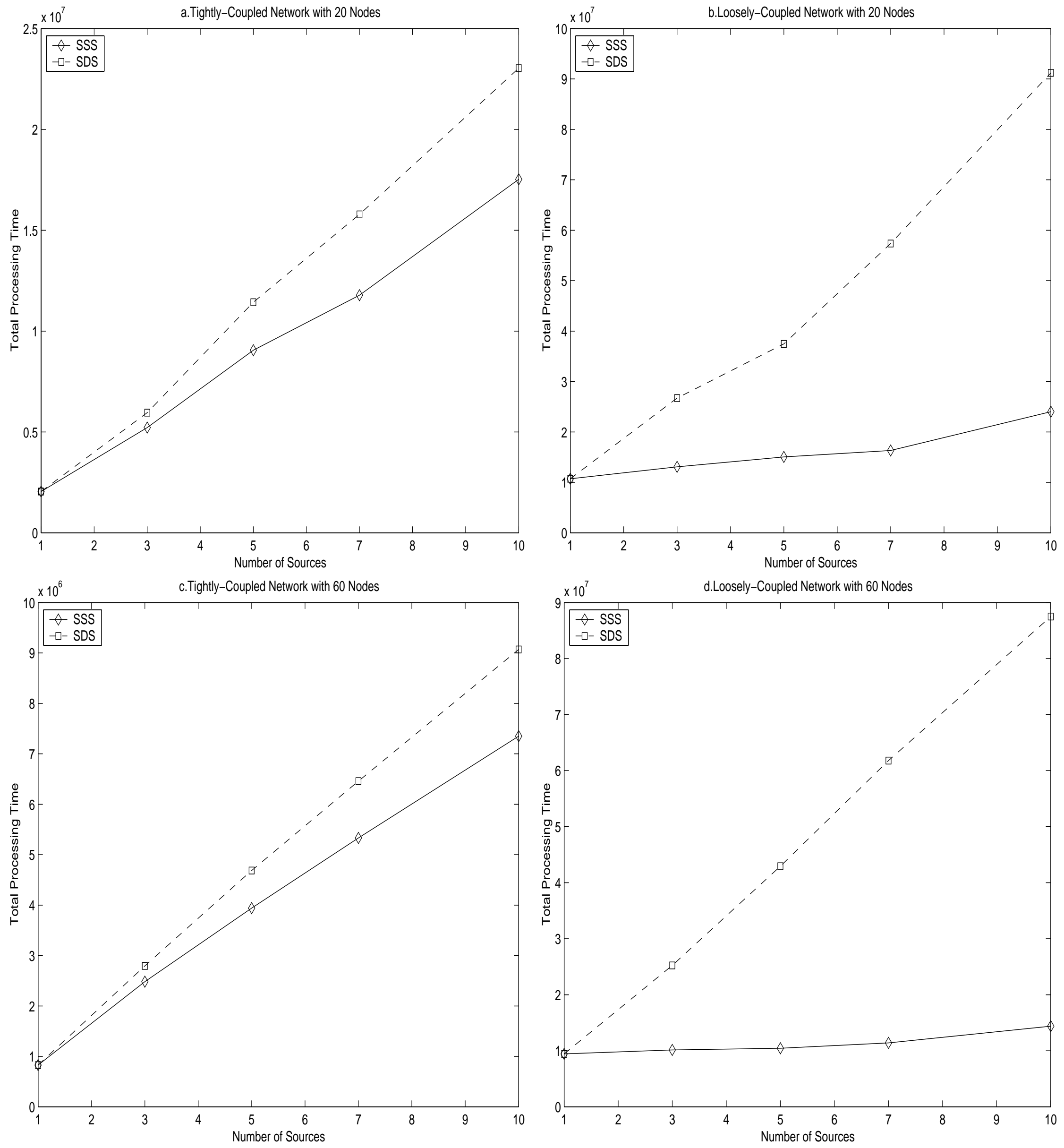


Figure 5.3: Experiment Results for the Static Case

662%($2.1 \times 10^6$ to $1.6 \times 10^7$) and 418%($1.1 \times 10^7$ to $5.7 \times 10^7$) in tightly-coupled networks and loosely-coupled networks, respectively. In the same case, the total processing time of SSS also increases by 471%($2.1 \times 10^6$ to $1.2 \times 10^7$) in the tightly-coupled networks, but it increases only by 45%($1.1 \times 10^7$ to $1.6 \times 10^7$) in the loosely-coupled networks. We refer to the above SSS' characteristic as the "*load insensitive*" property, because the total processing time of SSS seems "insensitive" to the increase of number of loads. Notice that this property can only be observed in a relatively loosely-coupled network.

The above load insensitive property can be explained by the "*Nearest Nodes Dominance*" effect, which is stated as follows. *In the presence of communication delays, for a given region, the source and its "nearest" nodes dominate this region's computation capacity.* Here, "nearest" is in terms of small communication delay. This effect reveals that the nodes which are "far from" (have a large communication delay) the source contribute little to the total computation capacity.

Now, let us see why SSS exhibits the load insensitive property. The total processing time of SSS is the maximum finish time of all loads, and its upper bound is shown in the Inequality (5.2), where the $T_i$s are determined by the respective critical domains' computation power. Notice that when the sources are sparse and communication delay is relatively large, each source's critical domain contains almost all its "nearest" nodes. Therefore, because of the nearest nodes dominance effect, $T_i$ is very close to the real processing time of $L_i$. Further, when adding a new source into the network, as long as sources remain sparse, it is highly probable that the new source

will not deprive other sources' "nearest" nodes. Therefore, previous existing critical domains' computation capacities will not decrease significantly, and hence the total processing time is less affected.

From the above discussion, we know that there are two prerequisites for SSS to exhibit the load insensitive property. First, the communication delay of the network should be relatively large. This explains why in the tightly-coupled network, the total processing time of SSS increases linearly with the number of sources, as shown in Figure 5.3.(a). Second, the sources in the network should be sparse enough, i.e., geographically well distributed. Therefore, when the number of sources exceeds some threshold, we should observe a sharp increase in the total processing time as number of sources increases further. As shown in Figure 5.3.(b), when the number of sources increase from 7 to 10, the total processing time increases from $1.6 \times 10^7$ to $2.4 \times 10^7$, almost triple the increment as compared to when the number of sources increases from 1 to 4 or from 4 to 7. However, notice that the threshold could be varied. For a larger network, SSS should be able to sustain the load insensitive property for a larger number of sources.

To verify this, we conduct another set of experiments on a network with 60 nodes. The network parameters are the same as previous experiments – $w$ is uniformly distributed among $[1, 10]$, $z$ is uniformly distributed among $[0, 0.5]$ for tightly-coupled network and among $[1, 2]$ for relatively loosely-coupled network. The results are shown in Figure 5.3.(c) and Figure 5.3.(d). We observe that for the loosely-coupled case when the number of sources exceeds 7 the total processing time does not increase

dramatically.

Further, we notice that as the network size grows, the total processing time for tightly-coupled network decreases significantly, but the total processing time for loosely-coupled network does not change significantly. Comparing Figure 5.3.(b) and 5.3.(d), we find that for the single source case, the total processing time only decreases from $1.1 \times 10^7$ to $0.95 \times 10^7$, as the network size grows from 20 nodes to 60 nodes. This indeed verifies the nearest nodes dominance effect.

Finally, it should be noted that since we adopt a linear cost model, the change of the initial load size does not affect the above observations. Consider the 60 nodes loosely-coupled network, we vary each region load size from $5,000,000$ to $15,000,000$, and the total processing time of SSS with respect to different number of sources are plotted in Figure 5.4. From the figure, we observe that the total processing time of SSS increases linearly with the load size.

### 5.5.2 Performance of DSS

Now, we will study the performance of DSS. Since DSS is a natural extension of SSS, its usefulness and effectiveness are shown in the above subsection. Therefore, in this subsection we mainly focus on the dynamic nature of DSS - the average queue length at each source. Notice that as long as we know the average queue length, applying Little's Theorem can easily yield other performance metrics.

We adopt the assumption made in Section 5.4, that is, the arrival of loads follow a poisson distribution and load size is exponentially distributed. Further, in our

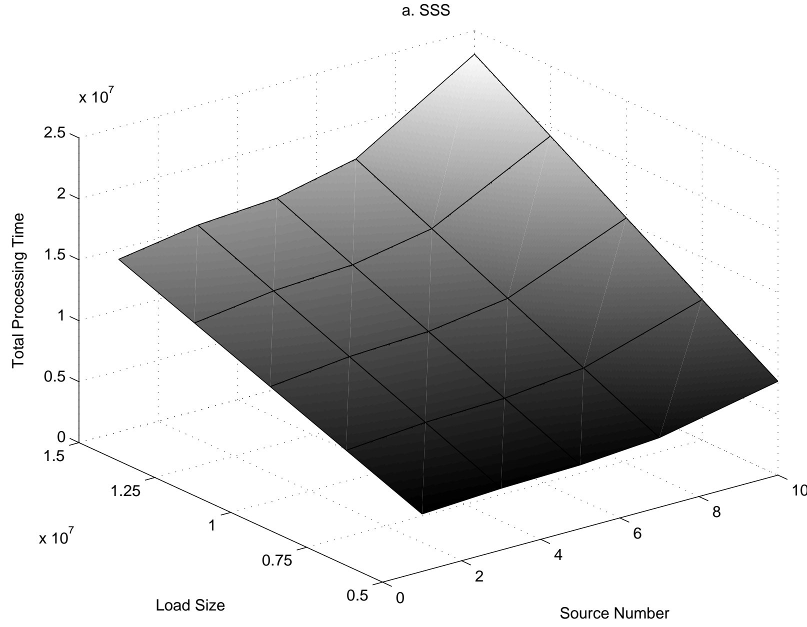


Figure 5.4: Total Processing Time of SSS with Different Load Size and Number of Sources

simulation, for any $Region_i$, we always let $\lambda_i < \mu_i^c$, which corresponds to Case 1 in Section 5.4. It is because the other two cases are either trivial (for Case 2) or too complex (for Case 3). Under the above constraints, an upper bound of each region's average queue length is given in Eqn (5.9). Below, we conduct experiments to study the actual average queue length of each region.

First, we consider networks with symmetric architecture and three sources. Similar to the static case, we generate two types of networks - loosely-coupled and tightly-coupled networks. In the loosely-coupled network, because of the presence of large communication delays, a region can only get a small amount of computation power from other idle regions. On the other hand, in the tightly-coupled network, since the communication delays are small, a region can get relatively larger amount of

Table 5.1: Regions' Equivalent Computation Capacities for Symmetric Networks

| Network Type | $E_i^c, i = 0, 1, 2$ | $E_i', i = 0, 1, 2$ | $E_i^{max}$ |
|---|---|---|---|
| Loosely-coupled Network | 2 | 1.8 | 1.65 |
| Tightly-coupled Network | 2 | 1.4 | 1 |

computation power from other idle regions. The respective equivalent computation power for each region in different cases are shown in Table 5.1, where $E_i'$ denotes the equivalent computation power of the other two regions, when only $Region_i$ is idle.

In our simulation, we let each region's average load size $\mu = 3$, and vary the load arrival rate $\lambda$. The average queue length with respect to different $\lambda$ is plotted in Figure 5.5. In the figure, the theoretical bounds are derived by Eqn (5.9). Notice that $\lambda$ denotes the average number of loads arriving in one unit of time. Since we consider divisible loads which are large in size, it is reasonable to let $\lambda < 1$.

From Figure 5.5, we observe that the average queue length increases with $\lambda$, which is natural. Further, we notice that the actual average queue length of the tightly-coupled network is much smaller than theoretical bound. However, as the network's communication delay becomes larger, the actual average queue length moves closer to the theoretical bound. This behavior is captured in the Figure 5.5.a. The loosely-coupled network's average queue length is quite close to the theoretical bound. Therefore, Eqn (5.9) serves as a very good estimator on average queue length for loosely-coupled networks, but may not give a "tight" estimation for tightly-coupled networks.

Next, we consider a more general case – regions having different computation

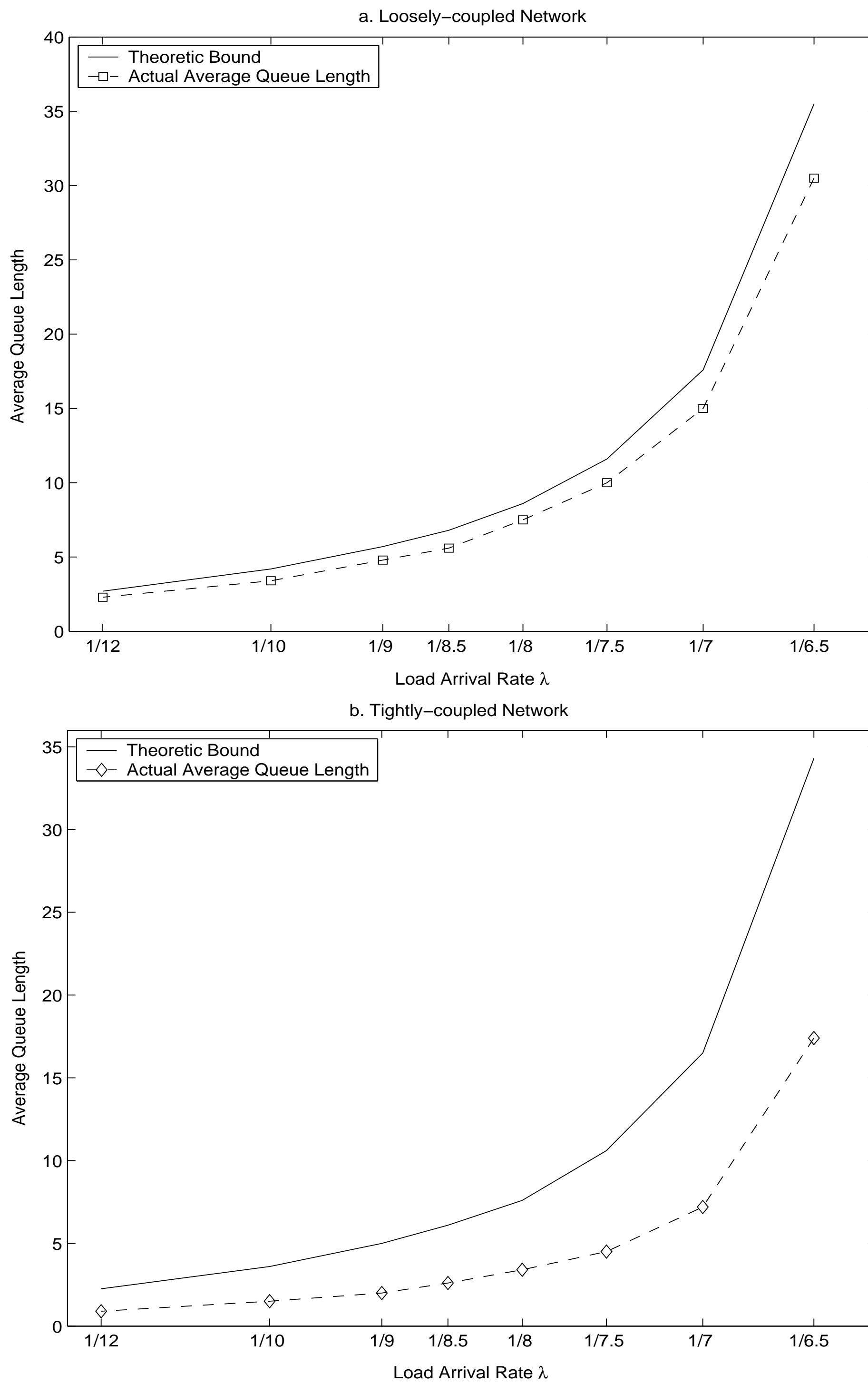


Figure 5.5: The Average Queue length of Loosely-coupled Network and Tightly-coupled Network With Respect to Different $\lambda$

Table 5.2: Regions' Equivalent Computation Capacities for the General Case

| $Region_i$ | $E_i^c$ | $E_i^0\prime$ | $E_i^1\prime$ | $E_i^2\prime$ | $E_i^{max}$ |
|---|---|---|---|---|---|
| $Region_0$ | 3.3 | $\infty$ | 2.9 | 2.65 | 2.5 |
| $Region_1$ | 4.5 | 3.5 | $\infty$ | 3.3 | 2.8 |
| $Region_2$ | 2.7 | 2.5 | 2.4 | $\infty$ | 2.25 |

Table 5.3: Experimental Results for the General Case

| | Experiment 1 | | | Experiment 2 | | | Experiment 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| $Region_i$ | $\lambda_i$ | $Q_i^t$ | $Q_i^a$ | $\lambda_i$ | $Q_i^t$ | $Q_i^a$ | $\lambda_i$ | $Q_i^t$ | $Q_i^a$ |
| $Region_0$ | 1/15 | 5.3 | 3.7 | 1/14 | 6.7 | 4.2 | 1/17 | 3.8 | 2.7 |
| $Region_1$ | 1/15 | 26 | 6.4 | 1/14 | 79.9 | 10.3 | 1/18 | 8.2 | 3.6 |
| $Region_2$ | 1/15 | 3.3 | 2.9 | 1/18 | 2.2 | 2.0 | 1/13 | 4.6 | 4.0 |

powers. We generate a network with 3 regions, and the regions' equivalent computation power in different cases is shown in Table 5.2, where $E_i^0\prime$ denotes the equivalent computation power for $Region_i$ when $Region_0$ is idle, and similar for $E_i^1\prime$ and $E_i^2\prime$.

Similarly, we let the average load size to be 3, and run the simulation for different sets of load arrival rates. Several results are reported in Table 5.3, where $Q_i^t$ denotes the theoretical bound of $Region_i$'s average queue length derived from Eqn (5.9) and $Q_i^a$ denotes its actual average queue length.

From Table 5.3, we find that $Q_2^a$ is very close to $Q_2^t$ independent of load arrival rates. This is because $Region_2$ is similar to a loosely-coupled network in that its main computation power lies within its critical domain, i.e., it cannot borrow too much extra computation power from other idle regions. However, as regions are able

to gain more computation capacity from other idle regions, the difference between $Q_i^t$ and $Q_i^a$ increases. This tendency is shown by $Region_0$ and $Region_1$'s performance. The above phenomena is also intuitive, as these regions' actual average computation capacity is much larger than their critical domains' computation capacity.

From the above discussions, we know that Eqn (5.9) can be directly used as a reference to assign buffer space to regions which exhibit more "loosely-coupled" characteristics. However, for the regions exhibiting more "tightly-coupled" characteristics, one should reduce the value predicted by Eqn (5.9) correspondingly, and then use the new value as the reference.

# Chapter 6

# Conclusions and Future Recommendations

In this thesis, we investigate the problem of scheduling divisible loads in the resource unaware environments, for different types of networks. Firstly, two novel strategies for scheduling and processing a divisible load on linear networks were presented. Both strategies used a portion of load to probe and estimate the speed parameters. Since the underlying network is a linear chain of processors, the choice of including the processors for computation becomes crucial in deciding the overall performance of the strategy, as the computing speeds and the link speeds are not known a priori. Any inadvertent choice of processors may slow down the computation. Taking this into account, the strategies proposed progress in an incremental fashion by including the processors in several phases in view of minimizing the processing time. The strategies take distinct advantage in utilizing faster processors earlier in the computation and progressively including other fast processors as time progresses. This special design is quite akin to linear network infrastructure as each load fraction has to percolate up and traverse several links in reaching its destination. Experiments were carried out to reveal the performance of the proposed strategies under several influencing factors. The performance of a strategy that serves as an upper bound (pureDLT) was also analyzed. The simulation shows that our strategies have gained a significant speed-up against the pureDLT.

One of the immediate problems that may need attention is in the choice of initial distribution of load among different phases. It would be interesting to derive an effective partitioning scheme for these phases, taking into consideration the size of the load, estimated parameters, dynamically as opposed to a fixed partitioning scheme proposed here.

We then addressed scheduling divisible loads in resource unaware multi-level tree networks. We considered two different cases of interests - SNP case and DNP case. In DNP case, the mild fluctuation of the link and processor speeds was considered. In such an environment, nodes can participate and leave the system at any time, which poses considerable challenge in prudently selecting the resources for efficient scheduling in the sense of minimizing the overall processing time. To this end, we proposed load distribution strategies to cater to the unknown and time varying network parameter cases to derive the best possible load distribution. The main contribution is in terms of designing strategies to efficiently exploit the probing technique and tune it to multi-level networks. We designed and analyzed two different strategies - static (for unknown but constant parameters) and dynamic (for unknown and time-varying parameters) and studied their performances. We demonstrated the strengths and several salient features of our algorithms in a systematic fashion using some illustrative examples. The examples show that the proposed approach is robust and resilient and easily adaptable to network fluctuations. The algorithm is shown to have a tracking capability, a property that is important for such dynamic environments. This property attempts to reuse fast processors efficiently, whenever available. On the other hand, the algorithms also prudently avoid using nodes that may cause bottleneck for

processing. However, one may notice the dynamic strategy may not be suitable for an extremely unstable network. Both the strategies presented in this study contribute significantly to the ultimate applicability of the divisible load distribution strategies, available in the literature, to realistic networks.

Two important issues of scheduling divisible load(s) in arbitrary networks were discussed. We argued that due to the difficulty in controlling, the probing technique may not be suitable in this case because of its centralized nature. In addition it is very costly for the root to prob the speed of a far away link or processor in a large-scale network. To this end, a reporting technique was suggested, which seems to be more suitable in large-scale arbitrary networks. Further, we investigated the performance of various spanning tree routing strategies for scheduling divisible loads utilizing the RAOLD-OS scheduling algorithm. The performance of these strategies have been evaluated for wide range of arbitrary graphs with varying connectivity and processor densities and analyzed the results. Our simulations study shows that the SPT routing strategy offers a better trade-off between time complexity and performance while RST renders better trade-off between performance and robustness. We also proposed an EST strategy which delivers best time performance, however with large time complexity.

We finally addressed the problem of scheduling multi-source divisible loads in arbitrary networks. This study considered a very generic graph/network with heterogeneous processing nodes and links, and considered each aspect of the problem dimension by analyzing the effects of several key parameters - network size (number

of nodes/scalability of the network), rate of arrival of loads, rate of processing of the loads, number of sources, etc. We proposed a novel graph partitioning scheme GP. GP solves the fundamental problem of scheduling loads in arbitrary networks - that is, from which source and which route a node should receive loads. Further, GP works very efficiently in the sense that it combines partitioning network and generating shortest path spanning tree. Then, based on GP, two novel scheduling strategies, SSS and DSS, were proposed. SSS applies to the static case where no new loads will arrive in the network, while DSS applies to the dynamic case where loads arrive randomly. We have shown that by using a reporting scheme, the strategies can be easily adapted to the resource unaware case, and compared to the resource aware case only a constant, one-time pre-scheduling overhead is introduced. We also studied the dynamic behavior of DSS using queuing theory, and our analysis revealed the upper bound of each load's average waiting time and each source's average queue length. Both SSS and DSS have shown a "hybrid" property of superposition and network partitioning. Our simulation has verified the effectiveness and usefulness of SSS and DSS. Further, the simulation has revealed a very interesting characteristic of SSS, that is, in loosely-coupled networks, as long as the number of sources is less than some threshold, increasing number of sources will not increase the total processing time. This characteristic can be explained by a "Nearest Nodes Dominance" effect. Our simulation also shows that, for DSS, the theoretical bound derived for each source's average queue length is very useful to predict the actual average queue length in loosely-coupled networks, but it may not be tight for tightly-coupled networks. This is because that a region's average computation capacity in tightly-coupled networks

can be much larger than the computation capacity of this region's critical domain.

This is the first attempt to consider scheduling multi-source divisible loads on arbitrary networks. We believe that this study is a very timely contribution to the DLT domain, as it represents a generalization of the problem. One of the key contributions of this study is in the graph partitioning approach and resource sharing across domains. This introduces the possibility of dynamic power tapping of idle resources. A possible extension to this work is to study the dynamic behavior of tightly-coupled networks more precisely. Further, because of the complexity of the problem, a dedicated network is considered here. One may attempt to incorporate the time-varying nature of the speeds of links and processors into problem formulation, and study the multi-source scheduling problem in a real dynamic environment.

# Bibliography


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

[2] R.Agrawal and H.V. Jagadish, "Partitioning Techniques for Large-Grained Parallelism", *IEEE Transanction on Computers*, Vol. 37, No. 12, pp. 1627-1634, 1988.

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

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

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

[6] Robertazzi, T., "Ten Reasons to Use Divisible Load Theory", *Computer*, Vol 36, 5, May 2003.

[7] Bharadwaj V., and S. Ranganath, “Theoretical and Experimental Study on Large Size Image Processing Applications using Divisible Load Paradigm on Distributed Bus Networks”, *Image and Vision Computing*, Elsevier Publishers, USA, Vol. 20, No. 13-14, pp. 917-936, December 2002.

[8] Chan, S.K., 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*, Vol. 58, pp. 71-79, 2001.

[9] Ko, K. and T. G. Robertazzi, “Record Search Time Evaluation”, *Proceedings of the Conference on Information Sciences and Systems*, Princeton University, Princeton, N.J., March 2000.

[10] Blazewicz, J., M. Drozdowski, and M. Markiewicz, “Divisible Task Scheduling - Concept and Verification”, *Parallel Computing*, Elsevier Science, Vol. 25, pp. 87-98, January 1999.

[11] Drozdowski, M., and P. Wolniewicz, “Experiments with Scheduling Divisible Tasks in Clusters of Workstations”, Euro-Par 2000, LNCS 1900, Springer-Verlag, pp. 311-319, 2000.

[12] D. Ghose and H. J. Kim, “Computing BLAS level-2 operations on workstation clusters using the divisible load paradigm”, *Mathematical and Computer Modeling*, Vol. 41, pp. 49-71, Jan 2005.

[13] K. Van der Raadt, Yang Yang, and Casanova, H., ”Practical Divisible Load Scheduling on Grid Platforms with APST-DV”, *Parallel and Distributed Pro-*

*cessing Symposium*, 2005.

[14] Bharadwaj, V., and G. Barlas, “Access Time Minimization for Distributed Multimedia Applications”, Special Issue in *Multimedia Tools and Applications*, Kluwer Academic Publishers, Vol. 12, No. 2/3, November 2000.

[15] G. Barlas and B. Veeravalli, “Optimized Distributed Delivery of Continuous-Media Documents over Unreliable Communication Links”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 10, pp. 982-994, Oct 2005.

[16] Balafoutis E., M. Paterakis, P. Triantafillou, G. Nerjes, P. Muth, and G. Weikum, “Clustered Scheduling Algorithms for Mixed-Media Disk Workloads in a Multimedia Server”, *Cluster Computing*, Kluwer Academic Publishers, Vol. 6, No. 1, pp. 75-86, January 2003.

[17] Wong H.M. and B. Veeravalli, “Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach”, *IEEE Transactiosn on Information Technology in BioMedicine*, Vol. 9, No. 4, pp. 489-501, December 2005.

[18] Suresh, S., Mani, V., Omkar, S.N. and Kim, H.J., “Parallel Video Processing using Divisible Load Scheduling Paradigm”, *Korean Journal of Broadcast Engineering*, 2005.

[19] Teo Tse Chin, Bharadwaj V, and Jia Jingxi, ”Handling Large-Size Discrete Wavelet Transform on Network-Based Computing Systems: Parallelization via Divisible Load Paradigm”, under revision of *Journal of Parallel and Distributed Computing*, 2008.

[20] Veeravalli, B., and Viswanadham, N., "Sub-optimal solutions using integer approximation techniques for scheduling divisible loads on distributed bus networks", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 30, No. 6, Nov 2000.

[21] K. Hwang, "Advanced Computer Architecture - Parallelism, Scalabiltty, Programmabtlity", McGraw-Hill and MIT Press, New York, 1993

[22] J.Sohn and T. G.Robertazzi, "Optimal Time-varying Load Sharing for Divisible Loads", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 34, No. 3, pp. 907-924, July 1998.

[23] D. Ghose, H. J. Kim, and T. H. Kim, "Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 10, pp. 897-907, October 2005.

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

[25] Blazewicz, J., and M. Drozdowski, "Scheduling Divisible Jobs on Hypercubes", *Parallel Computing*, Vol. 21, No. 12, pp. 1945-1956, December 1995.

[26] Sohn, J., and T.G. Robertazzi, "Optimal Divisible Job Load Sharing for Bus Networks", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 32, No. 1, pp. 34-40, January 1996.

[27] Mani V., "An Equivalent Tree Network Methodology for Efficient Utilization of

Front-Ends in Linear Network", *Cluster Computing*, Kluwer Academic Publishers, Vol. 6, No. 1, pp. 57-62, January 2003.

[28] Hung, J.T. and Robertazzi, T.G., "Divisible Load Cut Through Switching in Sequential Tree Networks", *IEEE Transactions on Aerospace and Electronic Systems*, vol. 40, no. 3, pp. 968-982, July 2004.

[29] Cheng, Y.C., and T. G. Robertazzi, "Distributed Computation for a Tree Network with Communication Delays", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 26, No. 3, pp. 511-516, 1990.

[30] Robertazzi, T. G., "Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 29, pp. 1216-1221, October 1993.

[31] Bharadwaj, V., D. Ghose, and V. Mani, "Optimal Sequencing and Arrangement in Distributed Single-Level Networks with Communication Delays", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 9, pp. 968-976, 1994.

[32] Ghose, D., and V. Mani, "Distributed Computation with Communication Delays: Asymptotic Performance Analysis", *Journal of Parallel and Distributed Computing*, Vol. 23, No. 3, pp. 293-305, December 1994.

[33] L.M. Ni, P.K. McKinley, "A survey of wormhole routing techniques in direct networks", *Computer*, Vol. 26, pp. 62-76, 1993.

[34] Blazewicz, J., M. Drozdowski, F. Guinand, and D. Trystram, "Scheduling a Divisible Task in a Two-dimensional toroidal Mesh", *Discrete Applied Mathematics*,

Vol. 94, No. 1-3, pp. 35-50, June 1999.

[35] Li, K., "Improved Methods for Divisible Load Distribution on k-Dimensional Meshes using Pipelined Communications", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 12, pp. 1250-1261, Dec. 2003.

[36] Taeyoung Lim and T.G. Robertazzi, "Efficient Parallel Video Processing Through Concurrent Communication on a Multi-port Star Network", *2006 Conference on Information Sciences and Systems*, 2006.

[37] Bataineh, S., and T. G. Robertazzi, "Bus-Oriented Load Sharing for a Network of Sensor Driven Processors" *IEEE Transactions on Systems, Man, and Cybernetics* Vol. 21, No. 5, Sept/Oct 1991.

[38] Bataineh, S., and T. G. Robertazzi, "Distributed Computation for a Bus Network with Communication Delays", *Proceedings of the 1991 Conference on Information Sciences and Systems*, The Johns Hopkins University, Baltimore MD, pp. 709-714, March 1991.

[39] Sohn, J., and T. G. Robertazzi, "Optimal Load Sharing for a Divisible Job on a Bus Network", *Proceedings of the 1993 Conference on Information Sciences and Systems*, The Johns Hopkins University, Baltimore MD, pp. 835-840, March 1993.

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

[41] Bataineh, S. and T.G. Robertazzi, "Ultimate Performance Limits for Networks

of Load Sharing Processors", *Proceedings of the 1992 Conference on Information Sciences and Systems*, Princeton N.J., pp. 794-799, March 1992.

[42] Blazewicz, J., and M. Drozdowski, "The Performance Limits of a Two-dimensional Network of Load Sharing Processors", *Foundations of Computing and Information Sciences*, Vol. 21, No. 1, pp. 3-15, 1996.

[43] Drozdowski, M., "Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems", Poznan University of Technology Press, Series Monographs, No. 321, Poznon, Poland, 1997.

[44] Drozdowski, M., and W. Glazek, "Scheduling Divisible Loads in a Three-dimensional Mesh of Processors", *Parallel Computing*, Vol. 25, No. 4, pp. 381-404, April 1999.

[45] Li, K., "Speedup of Parallel Processing of Divisible Loads on k-Dimensional Meshes and Tori", *The Computer Journal*, Vol. 46, No. 6, pp. 625-631, 2003.

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

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

[48] Jui, T. H., H. J. Kim, and T. G. Robertazzi, "Scalable Scheduling in Parallel

Processors", *Proceedings of the 2002 Conference on Information Sciences and Systems*, Princeton University, pp. 20-22, March 2002.

[49] Beaumont, O., Casanova, H., Legrand, A., Robert, Y., and Yang, Y.,"Scheduling divisible loads on star and tree networks: results and open problems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 3, Mar 2005.

[50] Li, K., "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.

[51] Yao, J. and Veeravalli, B., "Design and Performance Analysis of Divisible Load Scheduling Strategies on Arbitrary Graphs", *Cluster Computing*, Vol. 7, No. 2, pp.841-865, 2004.

[52] P. Byrnes and L. A. Miller, "Divisible load scheduling in distributed computing environments: complexity and algorithms", Technical Report MN-ISYE-TR-06-006, University of Minnesota Graduate Program in Industrial and Systems Engineering, 2006.

[53] Darin England, Bharadwaj Veeravalli, and Jon Weissman, "A Robust Spanning Tree Topology for Data Collection and Dissemination in Distributed Environments", *IEEE Transaction on Parallel and Distributed System*, Vol.18, No.5, pp.608-620, 2007.

[54] Li, X., V. Bharadwaj, and C. C. Ko, "Divisible Load Scheduling on Single-level Tree Networks with Buffer Constraints", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 36, 4, pp. 1298-1308, October 2000

[55] Drozdowksi, M., and P. Wolniewicz, “Divisible Load Scheduling in Systems with Limited Memory”, *Cluster Computing*, Kluwer Academic Publishers, Vol. 6, No. 1, pp. 19-30, January 2003.

[56] Bharadwaj, V., and Jingnan, Y.,“Divisible load scheduling strategies on distributed multi-level tree networks with communication delays and buffer constraints”, *Computer Communications*, Vol. 27, pp.93-110, 2004.

[57] Drozdowski, M. and Wolniewicz, P., “Performance Limits of Divisible Load Processing in Systems with Limited Communication Buffers”, *Journal of Parallel and Distributed Computing*, Vol. 64, No. 8, pp. 960-973, 2004.

[58] Li, X., V. Bharadwaj, and C. C. Ko, “Divisible Load Scheduling in a Hypercube Cluster with Finite-size Buffers and Granularity Constraints”, *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2001*, Brisbane, Australia, 2001.

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

[60] Bharadwaj, V., X. Li, and C. C. Ko, “On the Influence of Start-up Costs in Scheduling Divisible Loads on Bus Networks”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 12, pp. 1288-1305, December 2000.

[61] M. Drozdowski, P. Wolniewicz, “Optimum divisible load scheduling on heterogeneous stars with limited memory”, *European Journal of Operational Research*, Vol. 172, No. 2, pp. 545-559, 2006.

[62] Arnaud Legrand, Yang Yang and Henri Casanova, "NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs", Technical Report CS2005-0818, CSE, UCSD, 2005.

[63] Bataineh, S., and M. Al-Ibrahim, "Effect of Fault-tolerance and Communication Delay on Response Time in a Multiprocessor System with a Bus Topology", *Computer Communications*, Vol. 17, pp. 843-851, 1994.

[64] Bharadwaj, V., H. F. Li, and T. Radhakrishnan, "Scheduling Divisible Loads in Bus Networks with Arbitrary Processor Release Times", *Computer Math. Applic.*, Vol. 32, No. 7, pp. 57-77, 1996.

[65] Veeravalli, B. and Min, W.H., "Scheduling Divisible Loads on Heterogeneous Linear Daisy Chain Networks with Arbitrary Processor Release Times", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 3, pp. 273-288, March 2004.

[66] Bharadwaj, V., and G. Barlas, "Scheduling Divisible Loads with Processor Release Times and Finite Size Buffer Capacity Constraints", Special Issue on Divisible Load Scheduling in *Cluster Computing*, Kluwer Academic Publishers, Vol. 6, No. 1, pp. 63-74, January 2003.

[67] Charcranoon, S., T. G. Robertazzi, and S. Luryi, "Parallel Processor Configuration Design with Processing/Transmission Costs" *IEEE Transactions on Computers* , Vol. 40, No. 9, pp. 987-991, September 2000.

[68] Charcranoon, S., Robertazzi, T.G., and Luryi, S., "Load Sequencing for a Parallel Processing Utility", *Journal of Parallel and Distributed Computing*, Vol. 64,

pp. 29-35, 2004.

[69] Moges, M., Ramirez, L.A., Gamboa, C. and Robertazzi, T,G., "Monetary Cost and Energy Use Optimization in Divisible Load Processing", *Proc. of the 2004 Conference on Information Sciences and Systems*, Princeton University, March 2004

[70] M.Drozdowski, M.Lawenda, "The combinatorics in divisible load scheduling", *Foundations of Computing and Decision Sciences*, Vol.30, No.4, pp.297-308, 2005.

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

[72] Bharadwaj, V., D. Ghose, and V. Mani, "Multi-installment Load Distribution in Tree Networks With Delays", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 31, No. 2, pp. 555-567, April 1995.

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

[74] D. Altilar, Y. Paker, "Optimal scheduling algorithms for communication constrained parallel processing", *Euro-Par 2002*, LNCS 2400, Springer Verlag, pp. 197C206, 2002.

[75] Olivier Beaumont, Arnaud Legrand, and Yves Robert, "Scheduling divisible workloads on heterogeneous Platforms", *Parallel Computing*, Vol. 29, pp. 1121-

1152, Sep 2003.

[76] Wolniewicz, P., "Multi-installment Divisible Job Processing with Communication Startup Cost", *Foundations of Computing and Decision Sciences*, Vol. 27, No.1, 43-57, 2002.

[77] Yang Yang, van der Raadt, K., Casanova, H., "Multiround Algorithms for Scheduling Divisible Loads", *IEEE Trans on Parallel and Distributed Systems*, Vol. 16, No. 11, pp. 1092- 1102, Nov 2005.

[78] T. Hagerup, "Allocating Independent Tasks to Parallel Processors: An Experimental Study", *Journal of Parallel and Distributed Computing*, Vol. 47, pp. 185-197, 1997.

[79] Yang Yang, Casanova, H., "RUMR: Robust Scheduling for Divisible Workloads", *IEEE International Symposium on High Performance Distributed Computing*, pp. 114-123, Jun 2003.

[80] Sohn, J. and T. G. Robertazzi, "A Multi-Job Load Sharing Strategy for Divisible Jobs on Bus Networks", *Proceedings of the 1994 Conference on Information Sciences and Systems*, Princeton University, Princeton NJ, March 1994.

[81] Bharadwaj, V., 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.

[82] Wong H.M. and B. Veeravalli, and Gerassimos, B, "Design and Performance Evaluation of Load Distribution Strategies for Multiple Divisible Loads on Heterogeneous Linear Daisy Chain Networks", *Journal of Parallel and Distributed*

*Computing*, pp. 1558-1577, 2005.

[83] M. Moges and T. Robertazzi, “Grid Scheduling Divisible Loads from Multiple Sources via Linear Programmin”, *International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, 2004.

[84] M.Moges, T. Robertazzi, and D. Yu, “Divisible Load Scheduling with Multiple Sources: Closed Form Solutions”, *2005 Conf. on Information Sciences and Systems*, Baltimore, MD, 2005.

[85] T. Lammie and T. Robertazzi, “A Linear Daisy Chain with Two Divisible Load Sources”, *2005 Conf. on Information Sciences and Systems*, Baltimore, MD, 2005.

[86] T.G. Robertazzi and Dantong Yu, “Multi-Source Grid Scheduling for Divisible Loads”, *2006 Conf. on Information Sciences and Systems*, 2006.

[87] Xiaolin, L., and Veeravalli, B., “A Processor-set Partitioning and Data Distribution Algorithm for Handling Divisible Loads from Multiple Sites in Single-Level Tree Networks”, to appear in *Cluster Computing*, 2009.

[88] B. Awerbuch and F. Leighton,“A simple local-control approximation algorithm for multicommodity flow”, *Proc. of the 34th IEEE Symp. on Foundations of Computer Science*, pp. 459-468, 1993.

[89] B. Awerbuch and F. Leighton, “Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks”, *Proc. of the 26th ACM Symp. on Theory of Computing*, pp. 487-496, 1994.

[90] James A. Broberg, Zhen Liu, Cathy H. Xia, and Li Zhang, "A multicommodity flow model for distributed stream processing", *SIGMETRICS*, 2006.

[91] Cathy H. Xia, James A. Broberg, Zhen Li, and Li Zhang, "A multicommodity flow model for distributed stream processing", *DISC*, 2006.

[92] Donald E. Knuth, "The Art of Computer Programming," Vol. 1, Addison-Wesley, Boston, 1997.

[93] T. H. Cormen, C. E. Leiserson, R.L. Rivest and C. Stein, " Introduction to Algorithms", 2nd Edition, MIT Press, September 2001, ISBN 0-262-03293-7.

[94] Dimitri Bertsekas and Robert Gallager, "Data Networks", Prentice Hall, New Jersey, 1992.

[95] Little,J, "A Proof of the Queueing Formula $L = \lambda W$", *Oper.Res.J*, 18:172-174, 1961.

# List of Publications

1. Jingxi Jia and Bharadwaj V, "Resource Unaware Computing - A Distributed Strategy for Divisible Load Processing on Linear Daisy Chain Networks", *The 14th IEEE International Conference on Networks (ICON)*, Singapore, September 2006.

3. Jingxi J, Bharadwaj V, and Debasish, G, "Adaptive Load Distribution Strategies for Divisible Load Processing on Resource Unaware Multi-level Tree Networks", *IEEE Transactions on Computers*, Vol. 65, No. 7, 2007.

2. Bharadwaj Veeravalli and Jingxi Jia, "Design, Analysis, and Performance Evaluation of an Efficient Resource Unaware Scheduling Strategy for Processing Divisible Loads on Distributed Linear Daisy Chain Networks", the *Proceedings of the 15th Annual IEEE International Conference on High Performance Computing (HIPC 2008)*, Bangalore, India, 2008.

4. Teo Tse Chin, Bharadwaj V, and Jingxi Jia, "Handling Large-Size Discrete Wavelet Transform on Network-Based Computing Systems: Parallelization via Divisible Load Paradigm", *Journal of Parallel and Distributed Computing*, vol.69, no.2, 2009.

5. Jingxi Jia, Bharadwaj Veeravalli and Jon Weissman, "Scheduling Multi-source Divisible Loads on Arbitrary Networks", accepted to appear in *IEEE Trans-*

*action on Parallel and Distributed System*, 2009.