DESIGN, ANALYSIS AND APPLICATION OF DIVISIBLE LOAD SCHEDULING STRATEGIES IN LINEAR DAISY CHAIN NETWORKS WITH SYSTEM CONSTRAINTS

WONG HAN MIN

(*B.Eng.(Hons.), University of Nottingham, United Kingdom*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

First of all, I would like to express my sincere appreciation and thanks to my supervisor, Assistant Professor Dr. Bharadwaj Veeravalli for his guidance, support and stimulating discussions during the course of this research. He has certainly made my research experience at the National University of Singapore, an unforgettable one.

Special thanks to my devoted parents, who provided me with never ending supports, encouragements and the very academic foundations that make everything possible. Not to forget my excellent brother who had greatly influenced me in everything I do with his attitude of perfection in everything he does.

Many thanks also to all my fellow lab-mates in Open Source Software Lab for the help and support during all the brain-storming periods, solving technical and analytical problems, throughout the research.

Finally, I would like to thank the university for the facilities and financial support that make this research a success.

Contents

Acknowledgements **i**

List of Figures **vi**

List of Tables **x**

Summary **xi**

1 Introduction **1**

1.1 Related Work 2

1.2 Issues to Be Studied and Main Contributions 4

1.3 Organization of the Thesis 6

2 System Modelling **7**

2.1 Divisible Loads 7

2.2 Linear Daisy Chain Network Architecture 8

2.3 Mathematical Models and Some Definitions 9

2.3.1 Processor model 9

2.3.2 Communication link model 10

2.3.3 Some notations and definitions 10

3 Load Distribution Strategies for Multiple Divisible Loads 11

3.1 Problem Formulation 12

3.2 Design and Analysis of a Load Distribution Strategy 15

3.2.1 Case 1: $L_n C_{1,n} \leq T(n-1) - t_{1,n}$ (Single-installment strategy) 19

3.2.2 Case 2: $L_n C_{1,n} > T(n-1) - t_{1,n}$ (Multi-installment strategy) 19

3.3 Heuristic Strategies 25

3.4 Simulation and Discussions of the results 32

3.4.1 Simulation experiments 33

3.4.2 Discussions of results 34

3.5 Concluding Remarks 44

4 Load Distribution Strategies with Arbitrary Processor Release Times 47

4.1 Problem Formulation 48

4.2 Design and Analysis of a Load Distribution Strategy 51

4.2.1 Identical release times 52

4.2.2 Calculation of an optimal number of installments 59

4.2.3 Non-identical release times 60

4.2.4 Special Cases 70

4.3 Heuristic strategies 71

4.4 Discussions of the Results 76

4.5 Concluding Remarks 80

5 Aligning Biological Sequences: A Divisible Load Scheduling Approach 81

5.1 Preliminaries and Problem Formulation 84

5.1.1 Smith-Waterman Algorithm 84

5.1.2 Trace-back process 86

5.1.3 Problem Formulation 88

5.2 Design and Analysis of Parallel Processing Strategy 90

5.2.1 Load distribution strategy 92

5.2.2 Distributed post-processing : Trace-back process 96

5.3 Heuristic Strategy 98

5.3.1 Idle time insertion 99

5.3.2 Reduced set processing 100

5.3.3 Hybrid strategy 101

5.4 Performance Evaluation and Discussions 103

5.4.1 Effects of communication links speeds and number of processors 104

5.4.2 Performance evaluation against the bus network architecture 107

5.5 Concluding Remarks 112

6 Conclusions and Future Work **115**

Bibliography **119**

Author's Publications **127**

Appendix **128**

List of Figures

2.1 Linear network with m processors with front-ends and $(m-1)$ links. 8

3.1 Timing diagram for the single-installment strategy when the load L_n can be completely distributed before $T(n-1)$ 16

3.2 Timing diagram showing a collision-free front-end operation between the load L_{n-1} and L_n for a $m = 6$ system. Exclusively for this timing diagram, the diagonally shaded area below the each processor's axis indicate the period when the front-end is busy 17

3.3 Timing diagram for the multi-installment strategy when the distribution of the load $L_{2,n}$ is completed before the computation process for load $L_{1,n}$ 21

3.4 Flow-chart diagram illustrating the workings of Heuristic A 27

3.5 Example illustrating the working style of Heuristic A 29

3.6 Timing diagram for Heuristic B when the value for δT is large 31

3.7 Timing diagram for Heuristic B when the value for δT is small 31

3.8 Average processing time when the loads are unsorted 34

3.9 Average processing time when the loads are sorted using SLF policy 35

3.10 Average processing time when the loads are sorted using LLF policy 35

3.11 Timing diagram for Heuristic A when the loads are sorted(SLF or LLF) . . . 36

3.12 Timing diagram for Heuristic A when the loads are unsorted 36

3.13 Timing diagram for Heuristic A when heuristic strategy is used in between two optimal distributions 37

3.14 Timing diagram for EXA1 38

3.15 Timing diagram for EXA2 39

3.16 Timing diagram showing a large unutilized CPU time when large δT is used . 40

3.17 Timing diagram showing a better performance with small δT 40

3.18 Performance gain of multiple loads distribution strategy compared with single load distribution strategy 43

4.1 Timing diagram for a load distribution strategy when all the load can be communicated before τ. 52

4.2 Timing diagram showing a collision free front-end operation between installments $n-1$ and n. The numbers inside the block denote the installment number. 54

4.3 Timing diagram for conservative strategy. 61

4.4 Flow chart illustrating Heuristic A. 73

4.5 Example for Heuristic B: Arbitrary release times. The numbers appearing in communication blocks of P_1 denote the installment number. 75

4.6 Flow chart showing the entire scheduling of a divisible load by a scheduler at P_1. 76

5.1 Illustration of the computational dependency of the element (x,y) in the **S** matrix 86

5.2 Linear network with m processors interconnected by $(m-1)$ communication links . . . 88

5.3 Distribution pattern for matrices **S**, **h**, and **f** . . . 91

5.4 Timing diagram when $m = 6$ . . . 93

5.5 Distributed trace-back process between P_i and P_{i-1} . . . 97

5.6 Timing diagram when **S** is not required to be sent to P_m . . . 98

5.7 Timing diagram for the *idle time insertion* heuristic when $m = 6$ . . . 99

5.8 Effect of communication link speed and number of processors on the speed-up when **S** is required to be sent to P_m . . . 105

5.9 Effect of communication link speed and number of processors on the speed-up when **S** is not required to be sent to P_m . . . 106

5.10 Extreme case when the condition (5.13) is at the verge of being satisfied . . . 107

5.11 Bus network architecture with m processors . . . 108

5.12 Timing diagram of the distribution strategy when $m = 5$, and $Q = 5$ . . . 108

5.13 Effect of communication link speed and number of processors on the speed-up when **S** is required to be sent to P_m, in bus network . . . 110

5.14 Effect of communication link speed and number of processors on the speed-up when **S** is not required to be sent to P_m, in bus network . . . 110

5.15 Effect of communication link speed and large number of processors on the speed-up when **S** is not required to be sent to P_m . . . 111

5.16 Effect of communication link speed and large number of processors on the speed-up when **S** is not required to be sent to P_m, in bus network 112

List of Tables

5.1 Example 5.1: Trace-back process 87

Summary

Network based computing system has proven to be a power tool in processing large computationally intensive loads for various applications. In this thesis, we consider the problem of design, analysis and application of load distribution strategies for divisible loads in linear networks with various real-life system constraints. We utilize the mathematical model adopted by Divisible Load Theory (DLT) literature in the design of our strategies. We investigate several influencing real-life scenarios and systematically derive strategies for each scenario.

In the existing DLT literature for linear networks, it is always assumed that only a single load is given to the system for processing. Although the load distribution strategy for single load can be directly implemented for scheduling multiple loads by considering the loads individually, the total time of processing all the loads will not be an optimum. When designing the load distribution strategy for multiple loads, the distribution and the finish time of the previous load have to be carefully taken into consideration when scheduling the current load as to ensure that no processors are left unutilized. We derive certain conditions to determine whether or not an optimum solution exists. In case an optimum solution does not exist, we propose two heuristic strategies. Using all the above strategies, we conduct four different rigorous simulation experiments to track the performance of strategies under several real-life situations.

In real-life scenario, it may happen that the processors in the system are busy with other computation task when the load arrives. As a result, the processors will not able to process the arriving load until they have finished their respective tasks. The time instant after which the processor is available, is referred as *release time.* We design a load distribution strategy by taking into account the release times of the processors in such a way that the entire processing time of the load is a minimum. We consider two generic cases in which all processors have identical release times and when all processors have arbitrary release times. We adopt both the single and multi-installment strategies proposed in the DLT literature in our design of load distribution strategies, wherever necessary, to achieve a minimum processing time. Finally, when optimal strategies cannot be realized, we propose two heuristic strategies, one for the identical case, and the other for non-identical release times case, respectively.

Finally, as to complete our analysis on distribution strategies in liner networks, we consider the problem of designing a strategy that is able to fully harness the advantages of the independent links in linear networks. We investigate the problem of aligning biological sequences in the field of bioinformatics. For first time in the domain of DLT, the problem of aligning biological sequences is attempted. We design multi-installment strategy to distribute the tasks such that a high degree of parallelism can be achieved. In designing our strategy, we consider and exploit the advantage of the independent links of linear networks.

Various future extensions are possible for the problems addressed in this thesis. We address several promising extensions at the end of this thesis.

Chapter 1

Introduction

Parallel and distributed computing systems have proven to be a power tool in processing large computationally intensive loads for various applications such as large scale physics experiments [1], biological sequence alignment [2], image feature extraction [3], Hough transform [4], etc. These loads, which are classified under *divisible loads*, are made up of smaller portions that can be processed independently by more than one processors. The theory of scheduling and processing of divisible loads, referred to as *divisible load theory* (DLT), exists since 1988 [5], has stimulated considerable interest among researchers in the field of parallel and distributed systems.

In DLT literature, the loads are assumed to be very large in size, homogeneous, and are arbitrarily divisible. This means that each partitioned portion of the load can be independently processed on any processor on the network and each portion demands identical processing requirements. DLT adopts a linear mathematical modelling of the processor speed and communication link speed parameters. In this model, the communication time delay is assumed to be proportional to the amount of load that is transferred over the channel, and the computation time is proportional to the amount of load assigned to the processor. The primary objective in the research of DLT is to determine the *optimal fractions* of the entire load to be assigned to each of the processors such that the total processing time of the entire load is a minimum. A collection of all the research contributions in DLT until 1996 can be found in the monograph [6] and a recent report consolidates all the published contributions till date (2003) in this domain [7]. Now we present a brief survey on some of significant contributions in this area relevant to the problem addressed in this thesis. Readers are referred to [8, 9] for an up-to-date survey.

1.1 Related Work

In the domain of DLT, the primary objective is to determine the load fractions to be assigned to each processor such that the overall processing time of the entire load is minimal. In all the research so far in this domain, it has been shown that in order to obtain an optimal processing time, it is necessary and sufficient that all the processors participating in the computation must stop computing at the same time instant. This condition is referred to as an *optimality criterion* in the literature. An analytic proof of the assumption for optimal load distribution for bus networks also appears in [10]. Studies in [11] analyzed the load distribution problem on a variety of computer networks such as linear, mesh and hypercube. Scheduling divisible loads in three dimensional mesh have also been studied [12] and was recently improved by Glazek in [13] by considering distributing the load in multiple stages. Barlas [14] presented an important result concerning an optimal sequencing in a tree network by including the results collection phase. Load partitioning of intensive computations of large matrix-vector products in a multicast bus network was investigated in [15].

To determine the ultimate speedup using DLT analysis, Li [16] conducted an asymptotic analysis for various network topologies. The paper [17] introduced simultaneous use of communication links to expedite the communication and the concept of *fractal hypercube* on the basis of *processor isomorphism* to obtain the optimal solution with fewer processors is proposed. Several practical issues addressed in conventional multiprocessor scheduling problems in the literature were also attempted in the domain of DLT. These studies include, handling multiple loads [18], scheduling divisible loads with arbitrary processor release times in bus networks [19], use of affined delay models for communication and computation for scheduling divisible loads [20, 21], and scheduling with the combination constraints of processor release times and finite-size buffers [23]. Kim [24] presented the model for store-and-bypass communication which is capable of minimizing the overall processing time. A recent works also considered the problem of scheduling divisible load in real-time [29] and systems with memory hierarchy [26]. The proposed algorithms in the literature were tested using experiments on real-life application problems. In [28] rigorous experimental implementation of the matrix-vector products on PC clusters as well as on a network of workstations (NOWs) were carried out and in [27] several other applications such as *pattern search*, *file compression*, *joining operation in relational databases*, *graph coloring and genetic search* using the divisible load paradigm. Experiments have also been performed on Multicast workstation clusters [29].

Extension of the DLT approach to other areas such as multimedia was attempted in [30]. In the domain of multimedia, the concept of DLT was cleverly exploited to retrieve a long duration movie from a pool of servers to serve a client site. In a recent paper, DLT is used in designing mixed-media disk scheduling algorithm for multi-media server [31]. Implementation of DLT have also been applied to the Grid for processing large scale physic experimental data was considered in [1]. We shall now discuss our contribution in the next section.

1.2 Issues to Be Studied and Main Contributions

In this thesis, we consider design and analysis of load distribution strategy on linear networks. Linear networks consist of set of processors interconnected in a linear daisy chain fashion. The network can be considered as a subset of other much complex network topologies such as mesh, grid and tree. As a result, strategies and solutions that are designed for linear networks can be easily mapped/modified into these network topologies to solve much complex problems. Although linear networks have a much complex pipelined communication pattern that may induce a relatively large communication delay, the independent communication links between processors in linear networks may offer significant advantages, depending on the underlying applications. For example, in image feature extraction application [32], adjacent processors are required to exchange boundary information and thus a linear network is a natural choice. The independent links offer flexibility in the scheduling process as communications can be done concurrently.

In the DLT literature, extensive studies have been carried out for the linear network topology in determining the optimal load distribution strategy to minimize the overall processing time. In all the works so far, it is always assumed that only a single load is given to the system for processing. Nevertheless, in most practical situations, this may not be true always as there may be cases where more than one load is given to the system for processing. This is especially obvious in a grid like environment where multiple loads are given to the networked computation system for processing. Designing a load distribution strategy for distributing multiple loads is a challenging task as the condition for the previous loads have to been taken into consideration when processing the next load. The optimal distribution for scheduling a single load in linear networks using single-installment strategy [5] and closed-form solutions [33] are derived in the literature. Although the load distribution strategy for single load can be directly implemented for scheduling multiple loads by considering the loads individually, the total time of processing all the loads will not be optimum. We design a load distribution strategy for handling multiple loads by taking into consideration of the distribution pattern of the previous load to ensure that no processors and available communication times are left unutilized. We derive certain conditions to determine whether or not an optimum solution exists. In case an optimum solution does not exist, we will resolve to heuristic strategies.

In handling multiple loads in real-life scenario, it may happen that the processors in the system are busy with other computation task when the load arrives. As a result, the processors will not able to process the arriving load until they have finished their respective tasks. A similar situation was considered in the literature in [19] for a bus network architecture, which consists of only a single communication link. Nevertheless, when the similar situation is applied to linear networks, the problem is by no means a trivial task as linear networks have a pipelined communication pattern involving $(m-1)$ links, where m is the number of processors in the system. Further, in the case of linear networks, adopting multi-installment strategy for load distribution is very complex as there is a scope for "collision" among the adjacent front-end operations, if communication phase is not scheduled carefully. In solving this problem, we systematically consider different possible cases that can arise in the design of a load distribution strategy. As done in the literature, we consider two possible cases of interest, namely identical release times and non-identical release times. We design single and multiple installment distribution strategies for both cases. We derive important conditions to determine if these strategies can be used. If these conditions cannot be satisfied, we resolve to heuristic strategies. We also propose few heuristic strategies for both the case of identical release times and non-identical release times.

Although the linear networks have a complex pipelined communication pattern that may incur large communication delay, the independent communication links between processors in linear networks may offer significant advantages. We investigate some real-life applications and design a load distribution strategy that is able to harness these advantages. Specifically, we consider the problem of aligning biological sequences in the field of Bioinformatics. We design a distribution strategy that offer high degree of parallelism and clearly show the advantages that the linear networks offer.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2, we introduce the system model adopt in DLT and general definitions and notations used throughout this thesis.

In Chapter 3, we investigate the problem of scheduling multiple divisible loads in linear networks. We design and conduct load distribution strategies to minimize the processing time of all the loads submitted to the system.

In Chapter 4, we consider the scenario where each processor has a release time after which only it can be used to process the assigned load. As done in the literature, we consider two possible cases of interest, namely identical release times and non-identical release times. We derive conditions for both cases to check if optimal solution exist and resolve to heuristic strategies when these conditions are violated.

In Chapter 5, we design load distribution strategy for the problem of aligning sequences in the field of Bioinformatics.

In Chapter 6, we conclude the research works done and envision the possible future extensions.

Chapter 2

System Modelling

In this chapter, we shall give a brief introduction our system model that is used in solving the problems concerned. This model is widely used in the literature and details can be found in [6, 7]. We will also introduce the terminology, definitions, and notations that are used throughout the thesis.

2.1 Divisible Loads

In general, computation data, or loads (or jobs), can be classified into two categories, namely divisible and indivisible loads. Indivisible loads are independent loads, of different sizes, that cannot be further subdivided and hence must be processed by a single processor. There has been intensive works done on scheduling indivisible loads such as [35, 36, 37], to quote a few. Scheduling these loads are known to be NP-complete and hence only resolvable with heuristics strategies that yield sub-optimal solutions.

On the other hand, divisible loads are loads that can be divided into smaller portions such that they can be distributed to more than one processors to be process to achieve a faster

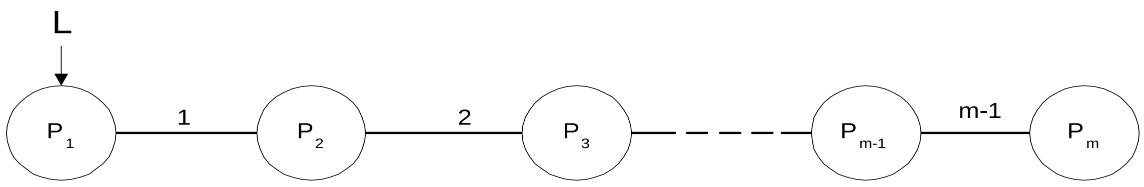


Figure 2.1: Linear network with m processors with front-ends and $(m-1)$ links.

overall processing time. Large linear data files, such as those found in image processing, large experimental processing, and cryptography are considered as divisible loads. Divisible loads can be further categorize into modularly and arbitrary divisible loads. Modularly divisible loads are divisible that can only be divided into smaller fixed size loads, based on the characteristic of the load, while arbitrary divisible loads can be divided into smaller size loads of any sizes.

2.2 Linear Daisy Chain Network Architecture

A linear daisy chain network architecture consists of m numbers of processors connected with $(m-1)$ numbers of communication links, as illustrated in Fig. 2.1. The processors in the system may or may not be equipped with front-ends. Front-ends are actually a co-processors which off loads the communication duties of a processor. Thus, a processor that has a front-end can communicate and compute concurrently. However, note that, the front-end of each processor cannot send and receive data simultaneously. The linear networks considered in this thesis is generally heterogenous and all processors are assumed to be equipped with front-ends. The heterogeneities considered are the computation and communication speed.

2.3 Mathematical Models and Some Definitions

In the DLT literatures, linear mathematical model is used to model the processors speed and communication links speed parameters. In this model, the communication time delay is assumed to be proportional to the amount of load that is transferred over the channel, and the computation time is proportional to the amount of load assigned to the processor [6]. Rigorously experiments have been done to verify the accuracy of this model [11, 28]. We adopt this model in solving the problems concerned in this thesis.

2.3.1 Processor model

Processor computation speed is modelled by the time taken for the individual processor P_i to compute a unit load. This parameter is denoted as w_i, and is defined as

$$w_i = \frac{\text{Time taken by } P_i \text{ to process a unit load}}{\text{Time taken by a standard processor to process a unit load}} \tag{2.1}$$

The speed of P_i is inversely proportional to w_i and the standard processor, which serve as a reference, will have $w_i = 1$. To specify the time performance, a common reference denoted as T_{cp} is defined as

$$T_{cp} = \text{Time taken to process a unit load by the standard processor} \tag{2.2}$$

Thus, $w_i T_{cp}$ represent the time taken by P_i to process a unit load. Example, if given a fraction α_i of the total load size of L_n to P_i to be process, the total time taken by P_i to process this load will be $\alpha_i L_n w_i T_{cp}$.

2.3.2 Communication link model

Communication link speed is modelled by the time taken for the individual link l_i to communicate a unit load. This parameter is denoted as z_i, and is defined as

$$z_i = \frac{\text{Time taken by } l_i \text{ to process a unit load}}{\text{Time taken by a standard link to communicate a unit load}} \tag{2.3}$$

Similar to the processor model, the speed of l_i is inversely proportional to z_i. The standard communication link, which serve as a reference, will have $z_i = 1$. To specify the time performance, a common reference for communication link is denoted as T_{cm} is defined as

$$T_{cm} = \text{Time taken to communicate a unit load by the standard link} \tag{2.4}$$

Thus, $z_i T_{cm}$ represent the time taken by l_i to communicate a unit load. Example, if given a fraction α_i of the total load size of L_n to be communicated over the link l_i, the total communication delay of this load fraction will be $\alpha_i L_n z_i T_{cm}$.

2.3.3 Some notations and definitions

We shall now introduce the terminology, definitions, and notations that are used throughout this thesis.

m	The total number of processors in the system
P_j	The processor j, where $j = 1, 2, ..., m$
l_i	The communication link connecting processors P_i and P_{i+1}, $i = 1, ..., m-1$
w_i	The inverse of the computation speed of the processor P_i.
T_{cp}	Time taken by the standard processor ($w_i = 1$) to compute a unit load.
z_i	The inverse of the communication speed of the link l_i.
T_{cm}	Time taken by a standard link ($z_i = 1$) to communicate a unit load.

Chapter 3

Load Distribution Strategies for Multiple Divisible Loads

In this chapter, we consider the problem of scheduling multiple divisible loads in heterogeneous linear daisy chain networks. Our objective is to design efficient load distribution strategies that a scheduler can adopt so as to minimize the total processing time of all the loads given for processing. The optimal distribution for scheduling a single load in linear networks using single-installment strategy [33] and multi-installment strategy [6] are derived in the literature. Although the load distribution strategy for single load can be directly implemented for scheduling multiple loads by considering the loads individually, the total time of processing all the loads will not be an optimum. Scheduling multiple loads in bus networks has been considered in [18] and a recent paper [40] presents some improved multiple loads distribution strategies for bus networks. In [40], rigorous simulation experiments are carried out to show the performance superiority of multi-installment strategy over single-installment strategy. Designing a multiple-load distribution strategy for linear networks is by no means a trivial task as the loads distribution basically has a pipelined communication pattern. This poses considerable challenge in designing strategies which maximizes the utilization of processor available times, front-end available times, and communication link times, respectively.

The organization of this chapter is as follows. In Section 3.1 we present the problem formulation and the terminology, definitions and notations that are used throughout the chapter. In Section 3.2, we will then present the design and analysis of our proposed strategy. In Section 3.3, we propose two heuristic strategies and present some illustrative examples. Later, in Section 3.4, we discuss the performance of the strategies proposed through rigorous simulation experiments. Finally, in Section 3.5, we conclude and discuss possible extensions to this work.

3.1 Problem Formulation

In this chapter, the loads for processing are assumed to arrive at one of the farthest end processors, referred to as *boundary processors*, say P_1 or P_m. Without loss of generality, we assume that the loads arrive at P_1. Further, we assume that the loads to be processed are resident in the buffer of the processor P_1. The process of load distribution is carried out by a scheduler residing at processor P_1. In general, the load distribution strategy is as follows.

The processor P_1 (which has a scheduler) keeps a load portion for itself and then sends the remaining portion to P_2. Processor P_2 upon receiving the portion from P_1, will keep a portion of the load for itself for processing and communicates the remaining load to P_3 and so on. Note that, as soon as a processor receives the load from its predecessor, it starts processing its portion and also starts communicating the remaining load to its successor. It should be noted that as far as the loads to be processed are concerned, the all the processors are single tasking machines, i.e., no two loads share the CPUs at the same instant in time. We use the *optimality criterion* mentioned in Chapter 1 in the design of an optimal distribution strategy. Also, it maybe noted that, in the design of optimal distribution strategy, we may be attempt to use multi-installment strategy [38].

As mentioned, we consider the case where multiple loads were given to the system, stored in the buffer of P_1, to be processed. We assume that N loads are resident in the buffer of P_1. When designing the load distribution strategy for multiple loads, the distribution and the finish time of the $(n-1)$-th load are taken into consideration when scheduling the n-th load as to ensure that no processors are left unutilized. In this chapter, we shall present the scheduling strategy for the n-th load, $2 \leq n \leq N$, by assuming that load distribution and the finish time of the $(n-1)$-th load are known to P_1.

We shall now introduce an index of terminology, definitions and notations that are used throughout the chapter.

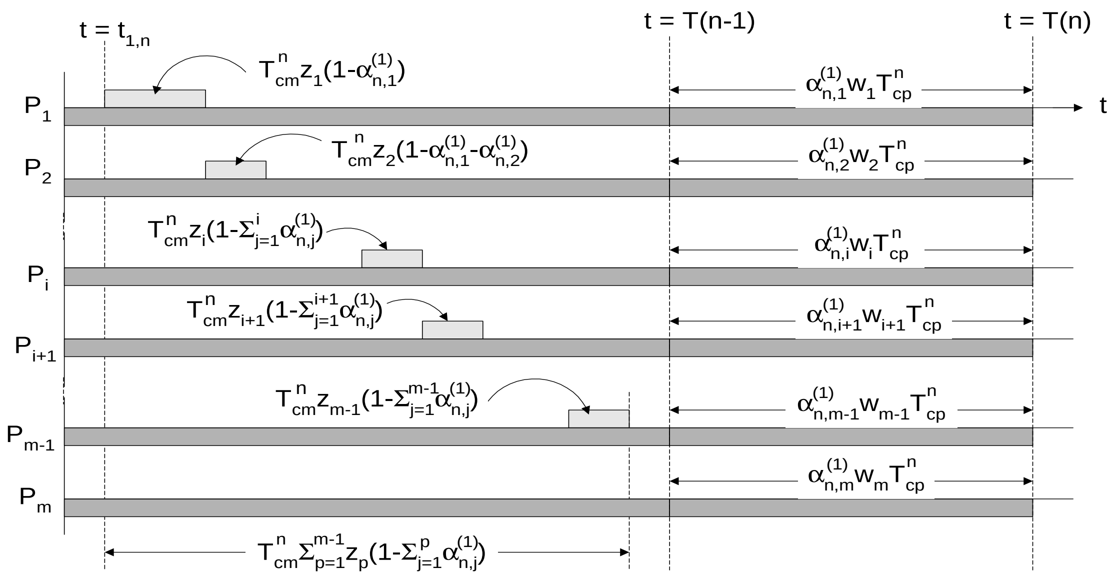
T_{cp}^{n}	Time taken by the standard processor ($w_i = 1$) to compute the n-th load, of size, L_n, where $T_{cp}^{n} = T_{cp}L_n$
T_{cm}^{n}	Time taken by a standard link ($z_i = 1$) to communicate the n-th load, of size, L_n, where $T_{cm}^{n} = T_{cm}L_n$.
N	The number of loads that is stored the buffer of P_1.
L_n	Size of the n-th load, where $1 \leq n \leq N$.
$L_{k,n}$	Portion of the n-th load, L_n, assigned to the k-th installment for processing.
$\alpha_{n,i}^{(k)}$	The fraction of the load assigned to P_i for processing the total load $L_{k,n}$, where $0 \leq \alpha_{n,i}^{(k)} \leq 1,\ \forall i = 1, ..., m$ and $\sum_{i=1}^{m} \alpha_{n,i}^{(k)} = 1$
$t_{k,n}$	This is the time instant at which the communication for the load to be distributed ($L_{k,n}$) for the k-th installment is initiated.
$C_{k,n}$	This is the total communication time of the k-th installment for the n-th load, of size L_n, when $L_{k,n} = 1$, where, $C_{k,n} = \frac{T_{cm}^{n}}{L_n} \sum_{p=1}^{m-1} z_p(1 - \sum_{j=1}^{p} \alpha_{n,j}^{(k)})$
$E_{k,n}$	This is the total processing time of P_m for the k-th installment for the n-th load, of size L_n, when $L_{k,n} = 1$, where, $E_{k,n} = \alpha_{n,m}^{(k)} w_m T_{cp}^{n} \frac{1}{L_n}$
$T(k, n)$	This is referred to as the *finish time* of the k-th installment for the n-th load, of size L_n, and it is defined as the time instant at which the processing of the k-th installment, for the n-th load, of size L_n, ends.
$T(n)$	This is referred to as the *finish time* of the n-th load, of size L_n, and it is defined as the time instant at which the processing of the n-th load ends. Where $T(n) = T(Q, n)$ if Q is the total number of installments require to finish processing the n-th load. And $T(N)$ is the finish time of the entire set of loads resident in P_1.

3.2 Design and Analysis of a Load Distribution Strategy

In this section, we shall design and analyze the load distribution strategies for processing multiple loads. For the analysis of the load distribution strategy when there is only one load, i.e., for $N = 1$, the readers are referred to [39, 33]. In this section, as a means of generalization, we shall consider the load distribution strategies for processing two adjacent loads, say the $(n-1)$-th and the n-th load. For the ease of understanding, we denote hereafter, the n-th load by its size L_n, as well as the load portion from n-th load assigned to the k-th installment by its size $L_{k,n}$. Example, the load L_x and $L_{y,z}$ has the size of L_x and $L_{y,z}$ respectively. Here we shall consider scheduling the load L_n by assuming that the distribution and $T(n-1)$ are known to P_1. It may be noted that one of the issues that need to be taken into consideration while scheduling L_n is that the load distribution, $L_n\alpha_{n,i}^{(1)}, i = 1, ..., m$, should be communicated to the respective processors $P_i, i = 1, ..., m$ before $T(n-1)$ so that no processors will be left un-utilized. Nevertheless, since the load L_n can be of any size, there may be a situation wherein the load L_n is very large that the load fractions may not able to reach all the respective processors before $T(n-1)$. As a result, we need to deal with two different distinct cases as follows.

Consider the timing diagram shown in Fig. 3.1. In this figure, the communication time is shown above the x-axis whereas the computation time is shown below the axis. This timing diagram corresponds to the case when the load L_n can be completely distributed before $T(n-1)$. For this distribution strategy, we will first derive the exact amount of load portions to be assigned to each processor. From the timing diagram shown in Fig. 3.1, we have,

$$\alpha_{n,i}^{(1)} w_i T_{cp}^n = \alpha_{n,i+1}^{(1)} w_{i+1} T_{cp}^n, \ i = 1, ..., m-1 \tag{3.1}$$



t = t_{1,n}
t = T(n-1)
t = T(n)
t
P_1
P_2
P_i
P_{i+1}
P_{m-1}
P_m
$T^n_{cm} z_1(1-\alpha^{(1)}_{n,1})$
$T^n_{cm} z_2(1-\alpha^{(1)}_{n,1}-\alpha^{(1)}_{n,2})$
$T^n_{cm} z_i(1-\Sigma^i_{j=1}\alpha^{(1)}_{n,j})$
$T^n_{cm} z_{i+1}(1-\Sigma^{i+1}_{j=1}\alpha^{(1)}_{n,j})$
$T^n_{cm} z_{m-1}(1-\Sigma^{m-1}_{j=1}\alpha^{(1)}_{n,j})$
$T^n_{cm}\Sigma^{m-1}_{p=1} z_p(1-\Sigma^p_{j=1}\alpha^{(1)}_{n,j})$
$\alpha^{(1)}_{n,1} w_1 T^n_{cp}$
$\alpha^{(1)}_{n,2} w_2 T^n_{cp}$
$\alpha^{(1)}_{n,i} w_i T^n_{cp}$
$\alpha^{(1)}_{n,i+1} w_{i+1} T^n_{cp}$
$\alpha^{(1)}_{n,m-1} w_{m-1} T^n_{cp}$
$\alpha^{(1)}_{n,m} w_m T^n_{cp}$

Figure 3.1: Timing diagram for the single-installment strategy when the load L_n can be completely distributed before $T(n-1)$

We can express each of the $\alpha^{(1)}_{n,i}$ in terms of $\alpha^{(1)}_{n,m}$ as,

$$\alpha^{(1)}_{n,i} = \alpha^{(1)}_{n,m}\frac{w_m T^n_{cp}}{w_i T^n_{cp}} = \alpha^{(1)}_{n,m}\frac{w_m}{w_i} \tag{3.2}$$

Using the fact that $\sum_{i=1}^{m}\alpha^{(1)}_{n,i} = 1$, we obtain,

$$\alpha^{(1)}_{n,m} = \frac{1}{1+\sum_{i=1}^{m-1}\frac{w_m}{w_i}} \tag{3.3}$$

Thus, using (3.3) in (3.2) we obtain,

$$\alpha^{(1)}_{n,i} = \left(\frac{1}{1+\sum_{p=1}^{m-1}\frac{w_m}{w_p}}\right)\frac{w_m}{w_i},\ i = 1,2,...,m \tag{3.4}$$

Note that the actual load that is assigned to each P_i is $L_n\alpha^{(1)}_{n,i}$. Then, we have to determine the time instant, $t_{1,n}$, at which P_1 shall start distributing the load $L_{1,n}$. The load distribution for the load $L_{1,n}$ starts at the time instant when P_1 finish delivering the load portion $L_{Q,n-1}(1-\alpha^{(1)}_{n-1,1})$ to P_2 (assuming the load L_{n-1} requires Q number of installments to be distributed) and will incur a "collision" with the front-end of P_2 as it is still busy sending respective load to P_3. Similarly by initiating communication of the load when the front-end of P_2 is available

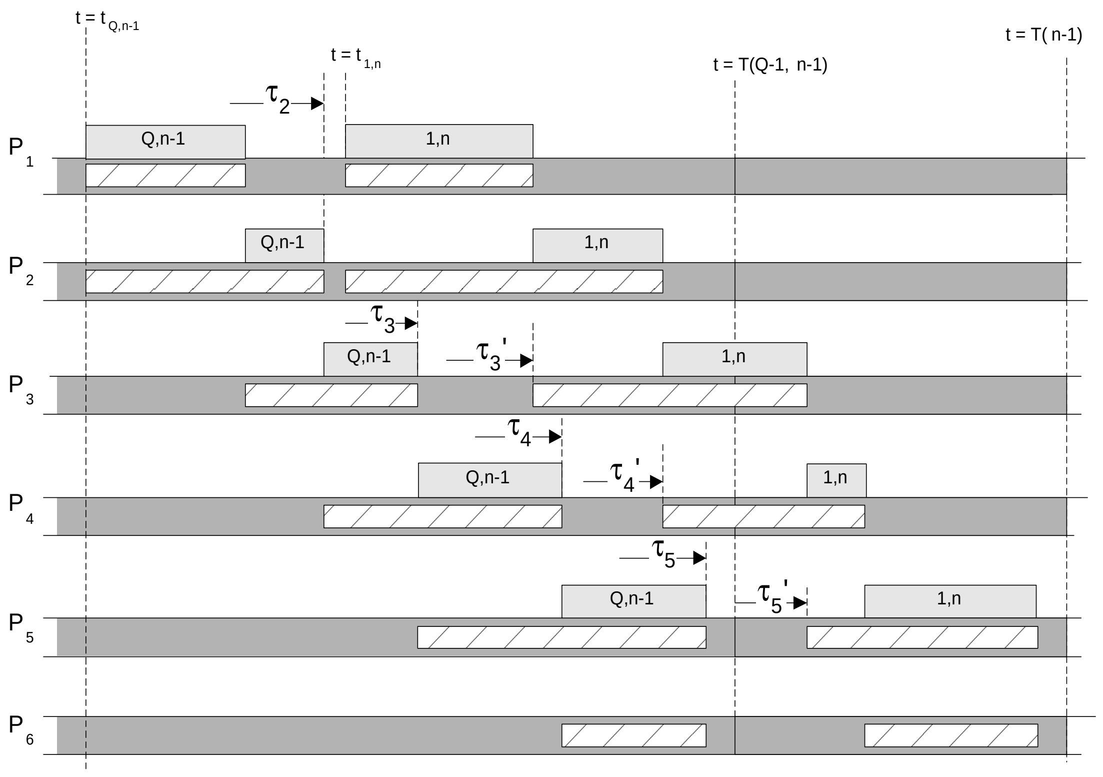


Figure 3.2: Timing diagram showing a collision-free front-end operation between the load L_{n-1} and L_n for a $m = 6$ system. Exclusively for this timing diagram, the diagonally shaded area below the each processor's axis indicate the period when the front-end is busy

may still cause similar collisions among the front-ends of other processors and this process may continue until processor P_{m-1}. As a result, we need to determine t_n, the starting time that will guarantee a collision-free front-end operation for all processors to communicate to their respective successors.

Before we describe the strategy in general, we shall consider a network with $m = 6$ and describe the strategy between two adjacent loads L_{n-1} and L_n as shown in the timing diagram in Fig. 3.2. From this diagram, we observe that, for the distribution of $L_{Q,n-1}$, the front-end of P_2 is occupied until τ_2, while the front-end of P_3, P_4, and P_5 is occupied until τ_3, τ_4 and τ_5 respectively. On the other hand, for the distribution of $L_{1,n}$, the front-end of P_2 will start receiving the load at $t_{1,n}$, while the front-end of P_3, P_4, and P_5 will start receiving at τ'_3, τ'_4

and τ_5', respectively (for the given $t_{1,n}$). Note that, $t_{1,n}$ in Fig. 3.2 is slightly shifted to right for the purpose of easy explanation. Now, in order to have a collision free operation for the front-end of P_2, i.e., $t_{1,n} \geq \tau_2$, we have,

$$t_{1,n}^{(2)} \geq t_{Q,n-1} + T_{cm}^{n-1} \frac{L_{Q,n-1}}{L_{n-1}} \sum_{p=1}^{2} z_p (1 - \sum_{j=1}^{p} \alpha_{n-1,j}^{(Q)}) \tag{3.5}$$

The superscript of $t_{1,n}$ denotes the index of the processor from which we consider a collision free operation. Similarly, for a collision free operation for the front-end of P_3, i.e., $\tau_3' \geq \tau_3$, we have

$$t_{1,n}^{(3)} + T_{cm}^{n} \sum_{p=1}^{1} z_p (1 - \sum_{j=1}^{p} \alpha_{n,j}^{(1)}) \geq t_{Q,n-1} + T_{cm}^{n-1} \frac{L_{Q,n-1}}{L_{n-1}} \sum_{p=1}^{3} z_p (1 - \sum_{j=1}^{p} \alpha_{n-1,j}^{(Q)}) \tag{3.6}$$

Similar equations can be obtained when considering collision free front-end operations for P_4 and P_5, respectively. Hence, in order to have a collision-free front-end operation for this system, we need to determine $t_{1,n}$ that can satisfy these following conditions, $t_{1,n} \geq \tau_2$, $\tau_3' \geq \tau_3$, $\tau_4' \geq \tau_4$ and $\tau_5' \geq \tau_5$. Note that, we need not consider a collision free operation for front-end of P_1 and P_6 (P_m) as can be seen from Fig. 3.2, the front-end for P_1 and P_6 are already taken into consideration while we are considering collision free operations for the front-end for P_2 and P_5 (P_{m-1}) respectively. Hence, for a m processors system, we will have $t_n^{(i)},\ i = 2, ..., m-1$. In general, to obtain a distribution for a collision free front-end operation for a m processors system, we must use the following $t_{1,n}$,

$$t_{1,n} = max\{t_{1,n}^{(i)}\},\ i = 2, ..., m-1 \tag{3.7}$$

where, for $i = 2, ..., m-1$

$$t_{1,n}^{(i)} = t_{Q,n-1} + L_{Q,n-1} X_{Q,n-1}^{(i)} - L_n Y_{1,n}^{(i)} \tag{3.8}$$

and

$$X_{k,n}^{(i)} = \frac{T_{cm}^{n}}{L_n} \sum_{p=1}^{i} z_p (1 - \sum_{j=1}^{p} \alpha_{n,j}^{(k)}) \quad , \quad Y_{k,n}^{(i)} = \frac{T_{cm}^{n}}{L_n} \sum_{p=1}^{i-2} z_p (1 - \sum_{j=1}^{p} \alpha_{n,j}^{(k)}) \tag{3.9}$$

Note that the value obtained from (3.7) guarantees a collision free scenario, as all the load that was percolating down the network for the previous load would have been completed before any processor communicates the next load.

As mentioned earlier, we may have two conditions where the load L_n may or may not be able to be communicated to all processors before $T(n-1)$. Thus, before we schedule the load L_n, following condition is first verified.

$$L_n C_{1,n} \le T(n-1) - t_{1,n} \tag{3.10}$$

The left hand side of the above expression is the total communication time needed to communicate the load portion $L_n \alpha_{n,i}^{(1)}$ to $P_i,\ i = 1, ..., m$ respectively where $\alpha_{n,j}^{(1)},\ j = 1, ..., m$ in $C_{1,n}$ are as defined in (3.4). On the other hand, the right hand side of the above expression is the total available time for communication before $T(n-1)$ where $t_{1,n}$ are as defined in (3.7).

3.2.1 Case 1: $L_n C_{1,n} \le T(n-1) - t_{1,n}$ (Single-installment strategy)

This is the case when (3.10) is satisfied. When (3.10) is satisfied, we can distribute the load L_n in a single installment and the optimal distribution is given by (3.4) while the finish time for L_n in this case is $T(n) = T(n-1) + L_{1,n}E_{1,n}$, where $L_{1,n} = L_n$.

3.2.2 Case 2: $L_n C_{1,n} > T(n-1) - t_{1,n}$ (Multi-installment strategy)

This is the case when the entire load L_n cannot be communicated to all the processors in a single installment, e.g., when (3.10) is violated. This prompts us to use multi-installment strategy where the load L_n is divided into smaller fractions and distributed in multiple installments.

For this case, at the first installment, there are two issues to be considered in the design of the strategy. Firstly, as done in the case of single-installment strategy, we have to consider collision-free operations among the front-ends of the system. Secondly, we have to determine the exact amount of loads for the load $L_{1,n}$, where $L_{1,n} < L_n$, such that it can be completely distributed before $T(n-1)$. Hence, starting from $t_{1,n}$, we have the following condition to be satisfied.

$$t_{1,n} + L_{1,n}C_{1,n} \leq T(n-1) \tag{3.11}$$

where $\alpha_{n,j}^{(1)},\ j = 1, ..., m$ in $C_{1,n}$ is as defined in (3.4) and $t_{1,n}$ is as defined in (3.7), that is $t_{1,n} = \max\{t_{1,n}^{(i)}\}, i = 2, ..., m-1$. Hence, in order to satisfy (3.11) we have the following condition,

$$t_{1,n}^{(i)} + L_{1,n}C_{1,n} \leq T(n-1)\ ,\ i = 2, ..., m-1 \tag{3.12}$$

Solving (3.12), with equality relationship, and (3.8) for a collision-free front-ends operation, for $i = 2, .., m-1$, we have,

$$t_{1,n}^{(i)} = \frac{C_{1,n}\left(t_{Q,n-1} + L_{Q,n-1}X_{Q,n-1}^{(i)}\right) - T(n-1)Y_{1,n}^{(i)}}{C_{1,n} - Y_{1,n}^{(i)}} \tag{3.13}$$

where $X_{k,n}^{(i)}$ and $Y_{k,n}^{(i)}$ are as defined in (3.9). We can then calculate $t_{1,n}$ which is defined in (3.7), but for the multi-installment strategy, we obtain $t_{1,n}^{(i)}$ from (3.13) instead of (3.8). After we have determined $t_{1,n}$, we can obtain $L_{1,n}$ by solving (3.11) with equality relationship. That is,

$$L_{1,n} = \frac{T(n-1) - t_{1,n}}{C_{1,n}} \tag{3.14}$$

The finish time of the first installment will then be $T(1,n) = T(n-1) + L_{1,n}E_{1,n}$. For the second installments onwards, the procedure is very similar to first installment. As shown in the timing diagram of Fig. 3.3, we attempt to complete the distribution of $L_{2,n}$ before $L_{1,n}$ is fully processed, e.g., before $T(1,n)$. In general, we attempt to complete the distribution of $L_{k,n}$ before $T(k-1,n)$. Thus, we have a condition, which is similar to (3.11), where the load $L_{k,n}$ has to be such that, the total communication time for $L_{k,n}$, starting from time $t = t_{k,n}$

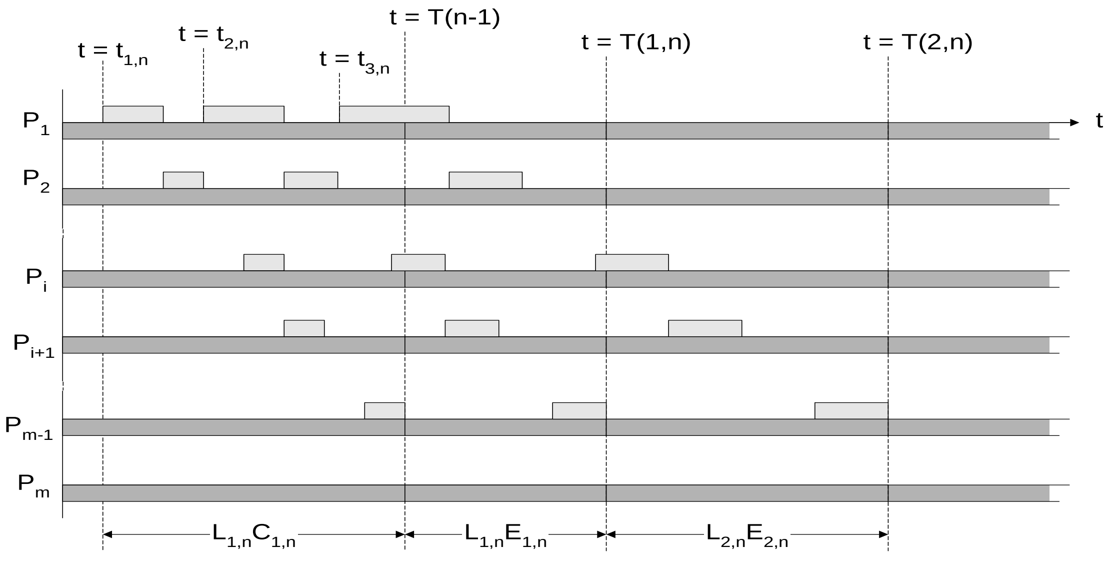


Figure 3.3: Timing diagram for the multi-installment strategy when the distribution of the load $L_{2,n}$ is completed before the computation process for load $L_{1,n}$

is less than the finish time of the load $L_{k-1,n}$, that is,

$$t_{k,n} + L_{k,n}C_{1,n} \leq T(k-1,n) \tag{3.15}$$

where $T(k-1,n) = t_{k-1,n} + L_{k-1,n}(C_{1,n} + E_{1,n})$. Note that we have replaced $C_{k,n}$ with $C_{1,n}$ in (3.15) since $\alpha_{n,j}^{(k)}$, the proportions in which the load $L_{k,n}$ will be distributed among m processors remain identical in every installment, where $\alpha_{n,j}^{(1)}$ is given by (3.4). This condition also applies to the $C_{k-1,n}$ and $E_{k-1,n}$ within $T(k-1,n)$ in (3.15).

Similar to the first installment, we have to consider a collision-free front-end operation between the distribution of $L_{k-1,n}$ and $L_{k,n}$. As a result, we have the following condition, which is similar to (3.8), for $i = 2, ..., m-1$,

$$t_{k,n}^{(i)} = t_{k-1,n} + L_{k-1,n}X_{1,n}^{(i)} - L_{k,n}Y_{1,n}^{(i)} \tag{3.16}$$

where $X_{1,n}^{(i)}$ and $Y_{1,n}^{(i)}$ are as defined in (3.9). Similar to the replacement of $C_{k,n}$(with $C_{1,n}$) in (3.15), we use $X_{1,n}^{(i)}$ and $Y_{1,n}^{(i)}$ instead of $X_{k-1,n}^{(i)}$ and $Y_{k,n}^{(i)}$ respectively in (3.16) because $X_{k,n}^{(i)}$ and $Y_{k,n}^{(i)}$ will remain constant for every installment of L_n.

Solving (3.15) and (3.16), for $i = 2, ..., m-1$, we have,

$$t_{k,n}^{(i)} = t_{k-1,n} + L_{k-1,n}H(i) \tag{3.17}$$

where,

$$H(i) = \left(\frac{X_{1,n}^{(i)}C_{1,n} - Y_{1,n}^{(i)}(C_{1,n} + E_{1,n})}{C_{1,n} - Y_{1,n}^{(i)}} \right) \tag{3.18}$$

We denote,

$$H = max\{H(i)\},\ \forall i = 2, ..., m-1 \tag{3.19}$$

For a collision-free front-end operation, we must have $t_{k,n} = max\{t_{k,n}^{(i)}\},\ i = 2, ..., m-1$. Hence, for a collision-free front-end operation, we have,

$$t_{k,n} = t_{k-1,n} + L_{k-1,n}H \tag{3.20}$$

It may be noted that in (3.20), the value of H may be pre-computed as it involves determining the values of $H(i)$ for all i, which are essentially constants. Thus, (3.20) can be used to compute the values of $t_{k,n}$, for $k > 1$ by using the previous values and the value of H. Now, after we obtained the value $t_{k,n}$, $L_{k,n}$ can then be calculated by solving (3.15) with equality relationship with respect of $L_{k,n}$. That is,

$$L_{k,n} = \frac{T(k-1,n) - t_{k,n}}{C_{1,n}} \tag{3.21}$$

The finish time of each installment is given by, $T(k,n) = T(k-1,n) + L_{k,n}E_{1,n}$. The above procedure will determine the start times of the installments and the amount of loads that needs to be assigned in each installment. We will repeat the above procedure until the last installment. The last installment can be determined by calculating the number of installments required to process the entire load L_n, which we will discuss in the next section. Now, suppose we assume that Q installments are required to process the entire load L_n, then for the last installment, we have

$$L_{Q,n} = L_n - \sum_{p=1}^{Q-1} L_{p,n} \tag{3.22}$$

Since we already obtained the amount of loads for the load $L_{Q,n}$, we can then calculate $t_{Q,n}$ by the following equation,

$$t_{Q,n} = max\{t_{Q,n}^{(i)}\},\ i = 2, ..., m-1 \tag{3.23}$$

where $t_{Q,n}^{(i)}$ is as defined (3.16) with Q in the places of k. Then, the finish time for the load L_n is, $T(n) = T(Q-1, n) + L_{Q,n}E_{1,n}$ Now, a final question that is left unanswered in our analysis so far is on the number of installments require to distribute the entire load L_n, which we address in the following section.

Calculation of an optimal number of installments

Here, we will present a strategy to calculate an optimal number of installments required, if it exists, to process the entire load L_n. We will derive some important conditions to ensure that the load L_n will be processed in a finite number of installments. We shall now assume that we need Q installments to distribute the entire load L_n to be processed and determine the conditions under which such a value of Q may exist. To derive this value of Q, we start by solving (3.20) and (3.21) to obtain a relationship between $L_{k-1,n}$ and $L_{k,n}$. Thus, $L_{k,n}$ is given by,

$$L_{k,n} = \frac{L_{k-1,n}}{C_{1,n}}B \tag{3.24}$$

where $B = C_{1,n} + E_{1,n} - H$. Using the above relationship, we can express each of $L_{k,n},\ k = 2, ..., Q$ in terms of $L_{1,n} = \frac{T(n-1)-t_{1,n}}{C_{1,n}}$, as,

$$L_{k,n} = \left(\frac{T(n-1) - t_{1,n}}{B}\right)\left(\frac{B}{C_{1,n}}\right)^k \tag{3.25}$$

Note that since the load $L_{k,n}$ is the fraction of the load L_n, and if Q is the last installment, it is obvious that $\sum_{j=1}^{Q} L_{j,n} = L_n$. Hence, we have,

$$L_n = \left(\frac{T(n-1) - t_{1,n}}{B}\right)\left(\frac{B}{C_{1,n}}\right)\sum_{j=0}^{Q-1}\left(\frac{B}{C_{1,n}}\right)^j \tag{3.26}$$

from which we obtain,

$$L_n = \left(\frac{T(n-1)-t_{1,n}}{B}\right)\left(\frac{B}{C_{1,n}}\right)\left(\frac{(\frac{B}{C_{1,n}})^Q - 1}{(\frac{B}{C_{1,n}}) - 1}\right) \tag{3.27}$$

Simplifying the above expression, we obtain,

$$Q = \left(\frac{ln\left(\frac{T(n-1)-t_{1,n}+L_n(B-C_{1,n})}{T(n-1)-t_{1,n}}\right)}{ln\left(\frac{B}{C_{1,n}}\right)}\right) \tag{3.28}$$

Now, from the above expression, for $Q > 0$, $T(n-1) - t_{1,n} + L_n(B - C_{1,n}) > 0$, where B is as defined above. Equivalently, we have the following relationship to be satisfied.

$$T(n-1) - t_{1,n} > L_n(H - E_{1,n}) \tag{3.29}$$

The above condition must be satisfied in order to obtain a feasible value of Q. Thus, when the above condition is satisfied, we distribute the load in Q installments. However, if the above condition is violated and no feasible value of Q may exist. When this happens, it means that continuous processing of the load will not be possible, which results in processor under utilization. In this case, we use heuristic strategies to complete the distribution of the load. We shall present two heuristic strategies and illustrative examples in the next section.

Note that, if a system has parameters such that $B = C_{1,n}$(i.e. $H = E_{1,n}$), the system will always satisfy (3.29). Nevertheless, Q cannot be obtained from (3.28). For such cases, we note that, from (3.25), $L_{k,n},\ k = 1, ..., Q$ will remain constant, hence Q can be calculated with the following equation,

$$Q = \frac{L_n}{L_{1,n}} \tag{3.30}$$

Lemma 3.1: When the condition (3.29) is violated, $L_{k,n} > L_{k+1,n}$.

Proof: When (3.29) is violated, we have,

$$T(n-1) - t_{1,n} \leq L_n(H - E_{1,n}) \tag{3.31}$$

Substituting $H = E_{1,n} + C_{1,n} - B$, we obtain,

$$T(n-1) - t_{1,n} \leq L_n(C_{1,n} - B) \tag{3.32}$$

Rearranging (3.32), we have,

$$C_{1,n} \geq \frac{T(n-1) - t_{1,n} + L_n B}{L_n} \tag{3.33}$$

Since $T(n-1) - t_{1,n} > 0$ and $L_n > 0$, we have,

$$\frac{B}{C_{1,n}} < 1 \tag{3.34}$$

Substituting the above condition into (3.25), observe $L_{k,n} > L_{k+1,n}$. Hence the proof.

3.3 Heuristic Strategies

In this section, we shall present two heuristic strategies that can be used to distribute the load when conditions for continuous processing of the load cannot be satisfied, i.e., when (3.29) is violated.

Heuristic A: In this heuristic, we attempt to distribute the entire load L_n in a single installment. We first partition the processors into two groups as those which receive their data before time $T(n-1)$ and those receive their data after time $T(n-1)$. We shall call the former group as set S, and the processors in this set satisfy the following condition for $i = 1, ..., m$

$$t_{1,n} + \sum_{p=1}^{i-1}(1 - \sum_{j=1}^{p} \alpha_{n,j}^{(1)}) z_p T_{cm}^n \leq T(n-1) \tag{3.35}$$

First, we assume that initially $S = \{P_1\}$. Hence, we will have the following relationship between $L_n\alpha_{n,i}^{(1)}$ and $L_n\alpha_{n,i+1}^{(1)}$, for $i = m-1, m-2, ..., 2$

$$\alpha_{n,i}^{(1)} w_i T_{cp}^n = \sum_{j=i+1}^{m} \alpha_{n,j}^{(1)} z_i T_{cm}^n + \alpha_{n,i+1}^{(1)} w_{i+1} T_{cp}^n \tag{3.36}$$

Note that the set of equations obtained from (3.36) is constant for a particular system hence can be pre-determined. Using (3.36), we can relate $L_n\alpha_{n,i}^{(1)}$, $i = 2, ..., m-1$, with respect to $L_n\alpha_{n,m}^{(1)}$ and using the fact that $\sum_{i=1}^{m} \alpha_{n,i}^{(1)} = 1$, we obtain,

$$\alpha_{n,1}^{(1)} + \sum_{i=2}^{m} \alpha_{n,i}^{(1)} = 1 \tag{3.37}$$

Expressing each of the $\alpha_{n,i}^{(1)},\ i = 2, ..., m$ in (3.37) with (3.36), we determine $\alpha_{n,1}^{(1)}$ as a function of $\alpha_{n,m}^{(1)}$. Hence, we obtained all the $\alpha_{n,i}^{(1)}, i = 1, ..., m$ with respect to $\alpha_{n,m}^{(1)}$. Then, since we know all the processors in the set S start computing at $T(n-1)$, we define a condition wherein the processing time for P_1, starting from $T(n-1)$, is the equal to the total communication time till P_m plus the processing time of P_m. That is,

$$T(n-1) + \alpha_{n,1}^{(1)} w_1 T_{cp}^n = t_{1,n} + L_n \left(C_{1,n} + E_{1,n}\right) \tag{3.38}$$

Note that, $t_{1,n}$ in the above equation is still unknown as it depends on the values of $\alpha_{n,j}^{(1)},\ j = 1, ..., m$ which is yet to be calculated. Hence, initially, we shall use values of $\alpha_{n,j}^{(1)},\ j = 1, ..., m$ calculated from (3.4) to approximate the value of $t_{1,n}$ using (3.7).

Solving (3.38) using all the $\alpha_{n,i}^{(1)}, i = 1, ..., m$ found previously, we can then calculate $\alpha_{n,m}^{(1)}$. With $\alpha_{n,m}^{(1)}$ known, all other $\alpha_{n,i}^{(1)}, i = 1, ..., m-1$ can be immediately calculated. Substituting these sets of $\alpha_{n,i}^{(1)},\ i = 1, ..., m$ into (3.9), we can then find a better approximation for $t_{1,n}$ using (3.7), we denote this $t_{1,n}$ as $t_{1,n}^*$. This $t_{1,n}^*$ can then be use as $t_{1,n}$ in (3.38) to solve for another set of $\alpha_{n,i}^{(1)},\ i = 1, ..., m$, which will be used to find $t_{1,n}$ again. This cycle will continue until the value of $t_{1,n}^* = t_{1,n}$. We will then use this value of $t_{1,n}$ for rest of the procedure. Note that, $t_{1,n}^*$ and $t_{1,n}$ may not be exactly equal to each other but both values will be almost identical after a few iteration.

Now, we use (3.35) to identify a set of processors that can be included in the set S together with P_1. After identifying the set S, we use the following set of recursive equations to determine the exact load portions to be assigned to the processors. Note that, for all processors in S, we use the following equation to determine the load portion $L_n\alpha_{n,i}^{(1)},\ P_i \in S$ with respect to $L_n\alpha_{n,1}^{(1)}$.

$$L_n\alpha_{n,i}^{(1)} = L_n\alpha_{n,1}^{(1)}\frac{w_1}{w_i} \tag{3.39}$$

Then, solving (3.39) for the processors in S, (3.36) for other processors and together with (3.37) and (3.38), we get another set of $L_n\alpha_{n,i}^{(1)},\ i = 1, ..., m$. These are the load fractions that

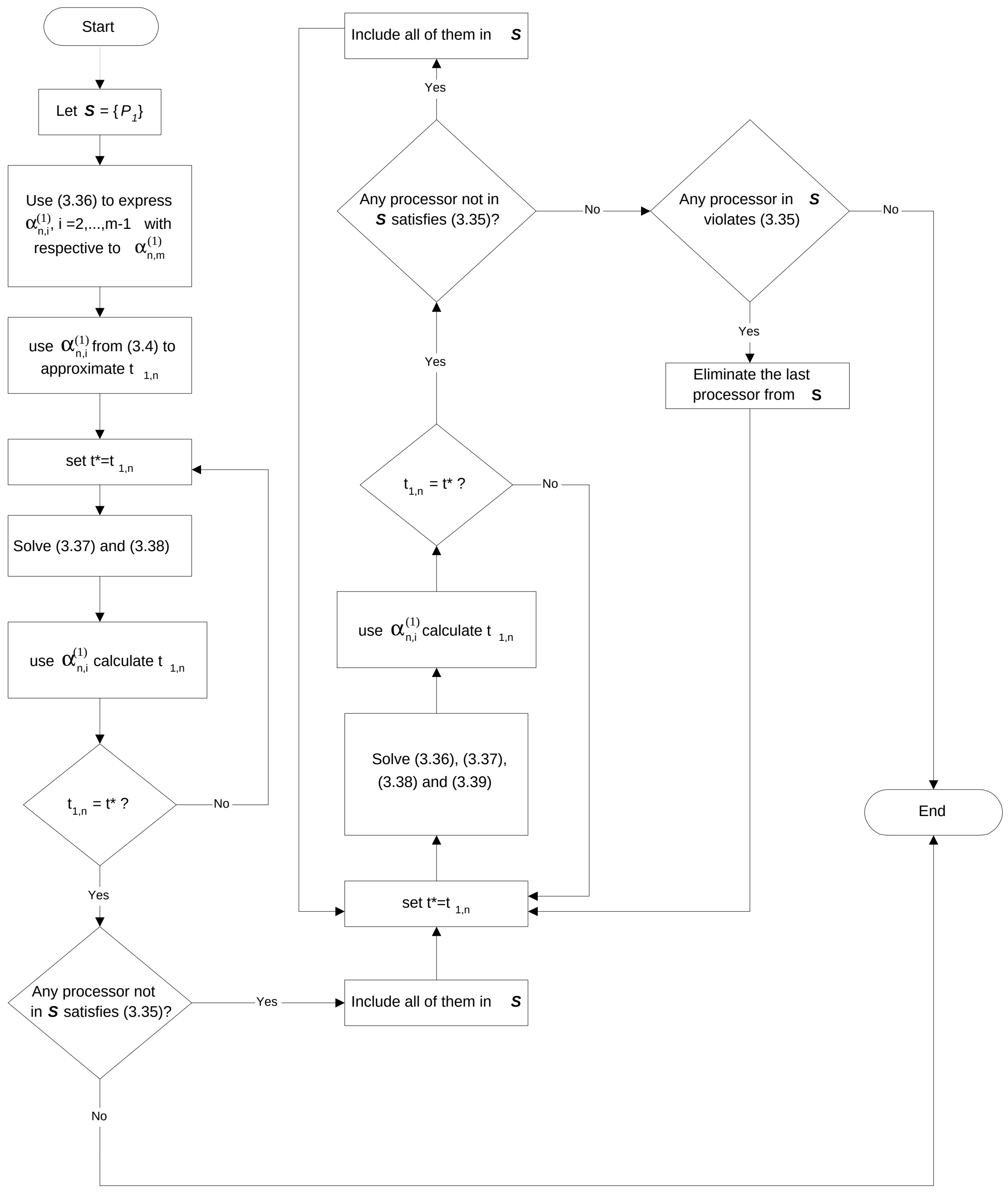

Start
Let $S = \{P_1\}$
Use (3.36) to express $\alpha^{(1)}_{n,i}$, i =2,...,m-1 with respective to $\alpha^{(1)}_{n,m}$
use $\alpha^{(1)}_{n,i}$ from (3.4) to approximate $t_{1,n}$
set $t^*=t_{1,n}$
Solve (3.37) and (3.38)
use $\alpha^{(1)}_{n,i}$ calculate $t_{1,n}$
$t_{1,n} = t^*$?
No
Yes
Any processor not in S satisfies (3.35)?
Yes
No
Include all of them in S
set $t^*=t_{1,n}$
Solve (3.36), (3.37), (3.38) and (3.39)
use $\alpha^{(1)}_{n,i}$ calculate $t_{1,n}$
$t_{1,n} = t^*$?
No
Yes
Any processor not in S satisfies (3.35)?
Yes
Include all of them in S
No
Any processor in S violates (3.35)
Yes
Eliminate the last processor from S
No
End

Figure 3.4: Flow-chart diagram illustrating the workings of Heuristic A

are to be assigned to the processors in the system. Note that although (3.39) assumes that the processors in S start at $T(n-1)$, the exact communication delays are not accounted. Therefore, as a last step, we need to verify whether or not all the processors in S indeed start at $T(n-1)$ using (3.35). Thus, in case, if some processors in S violate (3.35), we will then eliminate the last processors in S and repeat the above procedure until all processors in S satisfy (3.35). On the other hand, if all the processors in S satisfy (3.35) it is guaranteed that all the processors in S can indeed start from $T(k-1)$. Further, at this stage, as an incentive, it may happen that few processors that do not belong to S may satisfy (3.35) and we include all these processors in the set S. The flowchart in Fig. 3.4 and the following example illustrates the workings of Heuristic A.

Example 3.1 (Heuristic A): Consider a linear network system with the parameters, $m=6$, $T_{cp}^{n}=5$, $T_{cm}^{n}=1$, $w_1=2$, $w_2=1$, $w_3=3$, $w_4=2$, $w_5=2$, $w_6=1$, $z_1=1$, $z_2=2$, $z_3=1$, $z_4=3$, $z_5=2$ and $L_n=380$. Also, for the $(n-1)$-th load, $\alpha_{n-1,1}^{(1)}=0.2594$, $\alpha_{n-1,2}^{(1)}=0.3989$, $\alpha_{n-1,3}^{(1)}=0.0874$, $\alpha_{n-1,4}^{(1)}=0.1057$, $\alpha_{n-1,5}^{(1)}=0.0612$, $\alpha_{n-1,6}^{(1)}=0.0874$ and $T(n-1)=1.0341\times 10^3$.

Initially we assume $S=\{P_1\}$. Then, we use (3.36) to obtain $\alpha_{n,2}^{(1)}=4.57\alpha_{n,6}^{(1)}$, $\alpha_{n,3}^{(1)}=1.00\alpha_{n,6}^{(1)}$, $\alpha_{n,4}^{(1)}=1.21\alpha_{n,6}^{(1)}$ and $\alpha_{n,5}^{(1)}=0.70\alpha_{n,6}^{(1)}$. Using these values in (3.37), we obtain $L_n\alpha_{n,1}^{(1)}=L_n-8.84L_n\alpha_{n,6}^{(1)}$. Using the value of $\alpha_{n,i}^{(1)}$, $i=1,...,m$ found using (3.4), we first approximate $t_{1,n}=7.45\times 10^2$. Now, with $t_{1,n}$ and all $L_n\alpha_{n,i}^{(1)}$, $i=1,...,6$ in respect to $L_n\alpha_{n,6}^{(1)}$ known, we can immediately obtain the value $L_n\alpha_{n,6}^{(1)}=35.23$ by solving (3.38). Other values, $L_n\alpha_{n,i}^{(1)}$, $i=1,...,5$ can then be calculated.

With this set of $\alpha_{n,i}^{(1)}$, $i=1,...,6$, we calculate the new $t_{1,n}=6.93\times 10^2$. We then set $t_{1,n}^{*}=t_{1,n}=6.93\times 10^2$ and repeat the above procedure. We found that $t_{1,n}^{*}=t_{1,n}$, hence we will use this $t_{1,n}$ for the remaining calculation. Next, we check this set of $\alpha_{n,i}^{(1)}$, $i=1,...,6$ and observed that P_1, P_2 and P_3 satisfy (3.35). Hence, we include P_2 and P_3 in S and then use

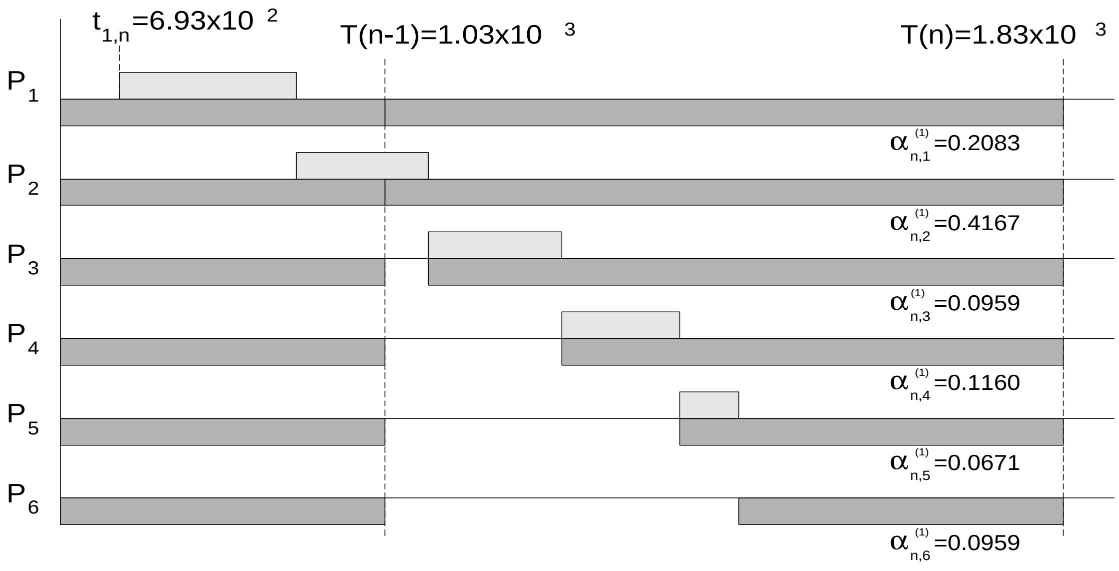


Figure 3.5: Example illustrating the working style of Heuristic A

(3.39) to get $L_n\alpha^{(1)}_{n,2} = 2L_n\alpha^{(1)}_{n,1}$ and $L_n\alpha^{(1)}_{n,3} = 0.67L_n\alpha^{(1)}_{n,1}$.

We repeat the above procedure again and we still obtain $t^*_{1,n} = t_{1,n}$. We then check the new values for the set of $\alpha^{(1)}_{n,i}$, $i = 1, ..., 6$ and found that P_3 has violated (3.35), hence we remove it from S. Repeating the above procedure, we found that $t_{1,n}$ remains the same and only P_1 and P_2 satisfy (3.35), and hence the result. The finish time is given by, 1.83×10^3 unit and Fig. 3.5 illustrates the final solution.

Heuristic B: For this heuristic strategy, we attempt to use multiple installments to distribute the load by intentionally introducing an additional delay such that (3.29) can be satisfied. From (3.29), we notice that one of the reasons for the violation of (3.29) is that there is insufficient communication time for L_n, i.e., $T(n-1) - t_{1,n}$ is small. Hence, in order to satisfy (3.29), we introduce an additional delay δT to $T(n-1)$ such that (3.29) can be satisfied. We choose a value of δT such that,

$$T(n-1) - t_{1,n} + \delta T > L_n(H - E_{1,n}) \tag{3.40}$$

The procedure for this heuristic is almost identical to the multi-installment strategy discussed in Section 3.2.2 except that δT is added to $T(n-1)$ in (3.11) and (3.14). Hence, we have the following two equations to replace (3.11) and (3.14), respectively.

$$\tau_n + L_{1,n}C_{1,n} \leq T(n-1) + \delta T \tag{3.41}$$

$$t_{1,n} + L_{1,n}C_{1,n} = T(n-1) + \delta T \tag{3.42}$$

Note that, although δT is the additional delay introduced to the system, having the smallest possible δT may not guarantee the best solution. For example, using a smaller value of δT may minimize the amount of processor time wasted (due to the delay introduced) and also may minimize $T(n)$. Nevertheless, having a small value of δT also means increasing of the number of installments required (increase complexity) to process the load L_n while decreasing the total available communication time for the next load L_{n+1}. The reason behind this is that, when (3.29) is initially violated, it means that $L_{k,n}$ is decreasing in size in each iteration (proven in Lemma 3.1). Also note that, from (3.41), when the value of δT is small, the amount of loads $L_{1,n}$ will be less as well. Hence, the amount of loads $L_{k,n}$ for each of the following installments will be subsequently lesser, thus requiring more installments to finish processing the load L_n. In the final installment for L_n, a small amount of load implies less processing time for that installment, which implies smaller total available communication time for L_{n+1} according to (3.11). Therefore, the value of δT needs to be tuned accordingly, that is considering the trade-off between a larger $T(n)$ for less complexity and allowing more communication time for the next load. The timing diagram in Fig. 3.6 and Fig. 3.7 illustrates the effect of having a small and large value of δT for the case of a 6 processors system.

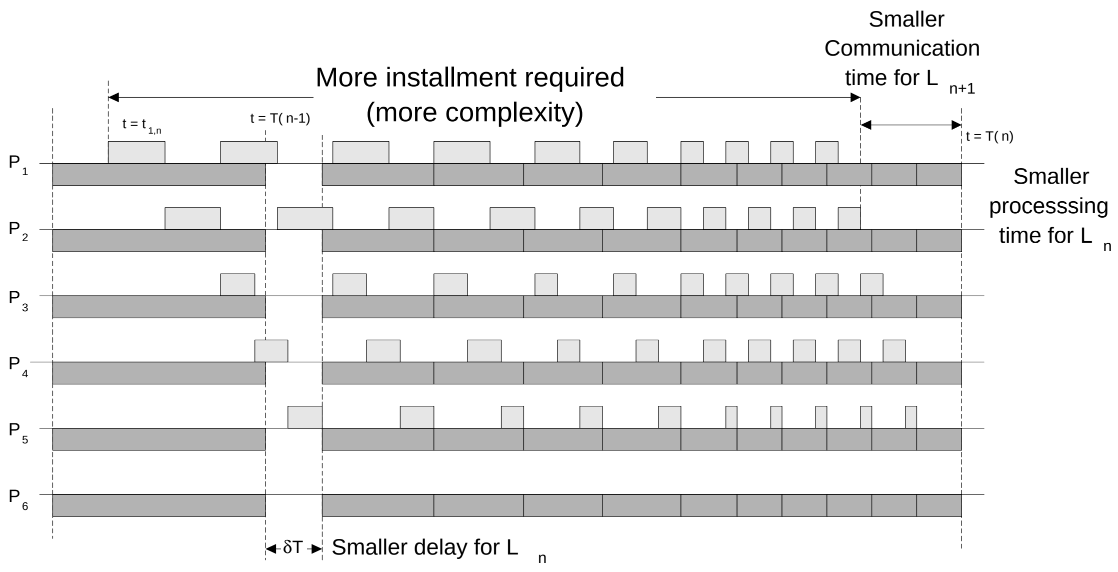


Figure 3.6: Timing diagram for Heuristic B when the value for δT is large

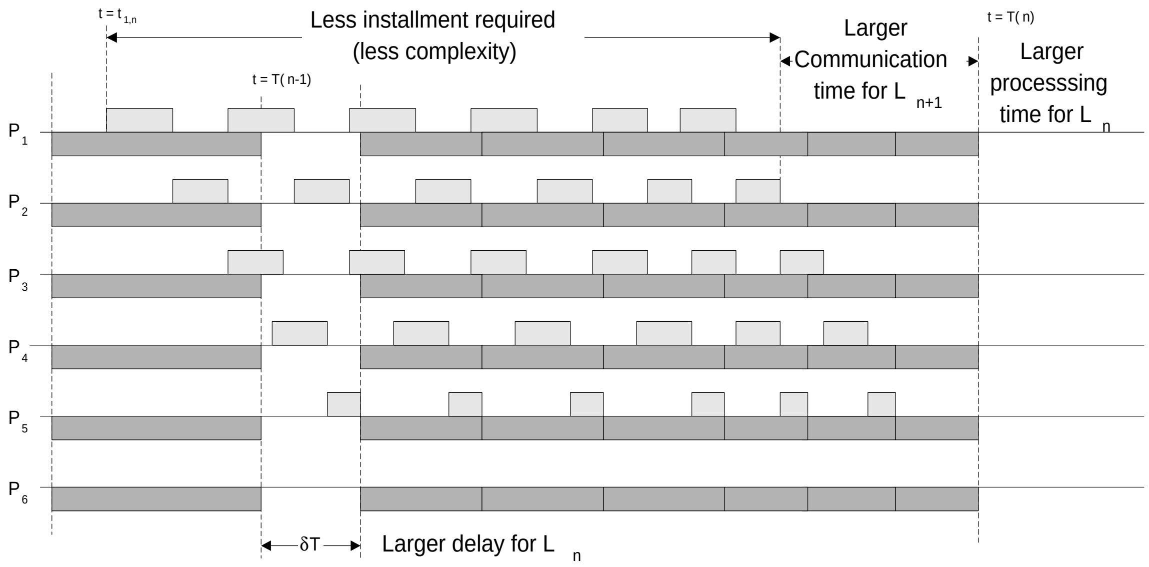


Figure 3.7: Timing diagram for Heuristic B when the value for δT is small

3.4 Simulation and Discussions of the results

As stated in Section 3.1, we assume that the set of loads is initially resident in the buffer of P_1. Hence, we have an option of sorting the given loads according to its size (either in ascending or descending order) before distributing them. Hence, we consider three cases in our simulation experiments where the given set of loads is (a) Unsorted (b) Sorted by smallest load first (SLF) (c) Sorted by largest load first (LLF).

To test the performance of both the heuristic strategies A and B proposed in Section 3.3, we have done rigorous simulations based on the 3 cases mentioned above. Further, we have also designed four distinct simulation experiments to identify the best configuration suitable for the two of the heuristics strategies proposed. We will present the distribution strategies used in these simulation experiments in the next subsection.

In our simulation study, we emulate a homogenous linear network with $m = 20$ processors, with $w_i = 1$ and $z_i = 1$, $\forall i = 1, ..., 20$. We carry out simulations to study the impact of the processor speeds on the performance of the heuristic strategies. We let $T^i_{cp} = T^j_{cp}, \forall i \neq j$ and assume that the communication time is directly proportional to the load size, that is $T^n_{cm} = 1 \times L_n$ sec $\forall n = 1, 2..., N$. Simulations are carried out for the value of $T^n_{cp} = 4L_n, 6L_n..., 40L_n$ secs, respectively. For each value of T^n_{cp}, 100 simulation test runs are carried out and we consider an average processing time. The system is given a set of $N = 50$ loads where L_n are uniformly distributed in the range $[100, 100000]$MBytes. For the first load, L_1, obviously we use an optimal distribution strategy presented in [5, 33]. For the remaining loads, we will use the strategies of the four simulation experiments to be presented in the next subsection.

3.4.1 Simulation experiments

In this section, we shall introduce different distribution strategies that are used in our four simulation experiments. By carrying out these experiments, we will later show that it is possible to identify the right combination while making a choice of a particular strategy. We denote these simulation experiments as EXA1, EXB1, EXA2, and EXB2. Note that while A and B denote the strategies discussed in Section 3.3, the numbers 1 and 2 in EXA1, EXB1, EXA2, and EXB2, denote two distinct possible choices in using the respective strategies.

Experiment using Heuristic A, #1 (EXA1) - In this experiment, we attempt to test the performance of Heuristic A if (3.10) and (3.29) cannot be satisfied. The following distribution strategy is used. First, condition (3.10) is verified. If (3.10) is satisfied, the load will then be distributed in a single installment. On the other hand, if (3.10) is violated, we will then verify whether (3.29) is satisfied or not. Multi-installment strategy will be used to distribute the load, if (3.29) is satisfied while Heuristic A will be used if (3.29) is violated.

Experiment using Heuristic B, #1 (EXB1) - In this experiment, we will implement Heuristic B, using a smallest possible value for δT, when (3.10) and (3.29) cannot be satisfied. The distribution strategy is similar to EXA1 but Heuristic B will be used rather than Heuristic A if (3.29) is violated.

Experiment using Heuristic A, #2 (EXA2) - In this experiment, we try to exploit the advantage of Heuristic A which renders more communication time for the next load. In this experiment Heuristic A is used for all the loads $L_2, L_3, ..., L_{49}$ whether or not (3.10) or (3.29) is satisfied for these loads. For the last load, L_{50}, distribution strategy of EXA1 will be implemented, since we do not have any further loads to process.

Experiment using Heuristic B, #2 (EXB2) - In this experiment, we want to examine the effect of using a larger value for δT in Heuristic B. The distribution strategy used is similar

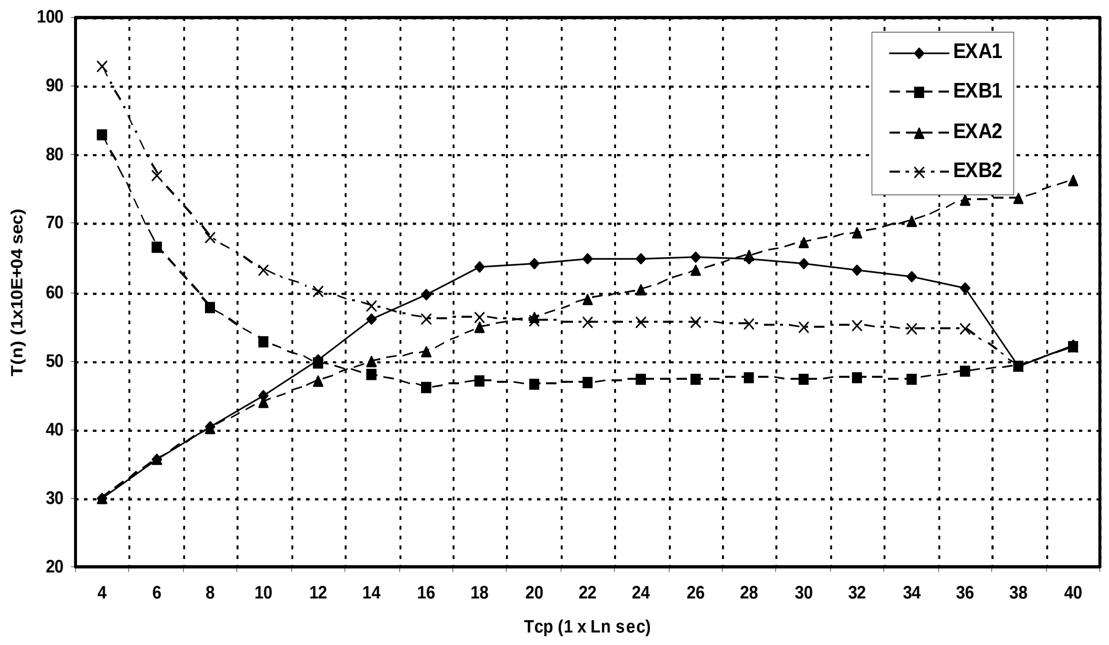


Figure 3.8: Average processing time when the loads are unsorted

to EXB1 but a larger value of δT is used.

Fig. 3.8, 3.9 and 3.10 are the results obtained from these simulation experiments.

3.4.2 Discussions of results

In this section, we discuss the results obtained from our simulation experiments. From Fig. 3.8, 3.9 and 3.10, we see that the processing time for EXA1, EXB1, and EXB2 yield more-or-less the same processing time after $T_{cp}^n = 38L_n$. The reason is that for $m = 20$, we have $H < E_{1,n}$, $n = 1, ..., 50$, when $T_{cp}^n > 38L_n$ (the exact value is $T_{cp}^n \geq 37L_n$). When $H < E_{1,n}$, we can see that (3.29) will always be satisfied for all the loads hence guaranteeing an optimum solution for all loads, for all the three strategies. Since in EXA2, optimal distribution is not used whether or not (3.10) or (3.29) is satisfied, optimal processing time cannot be achieved.

An interesting observation can be made at this stage is that Heuristic A (EXA1 and EXA2) tends to perform better in general, if the loads are sorted (either by SLF or LLF). This is

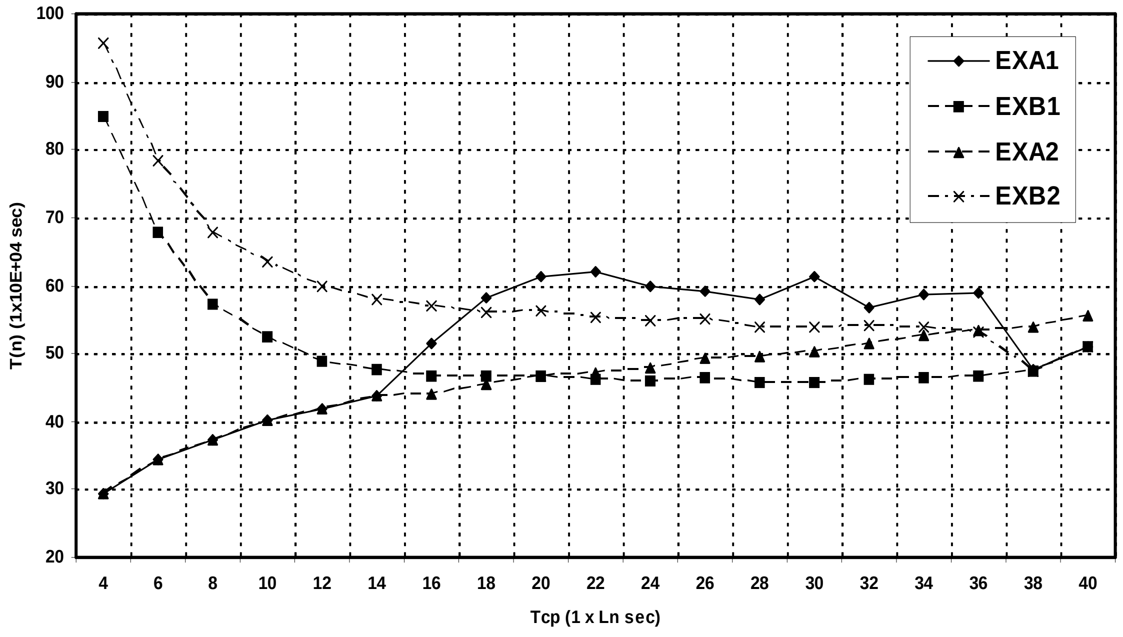


Figure 3.9: Average processing time when the loads are sorted using SLF policy

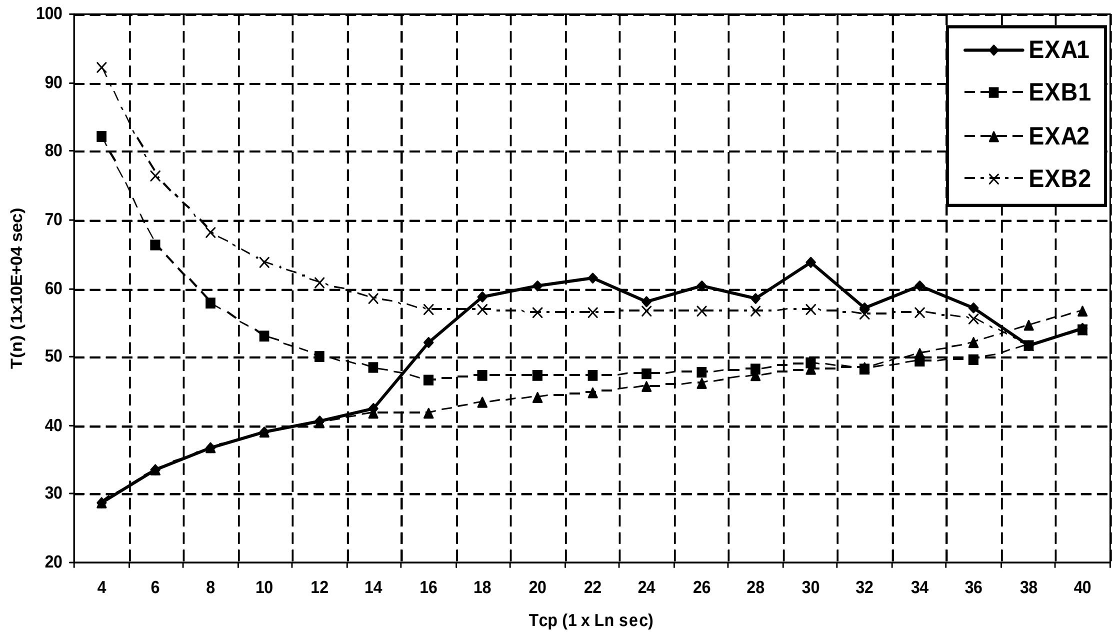


Figure 3.10: Average processing time when the loads are sorted using LLF policy

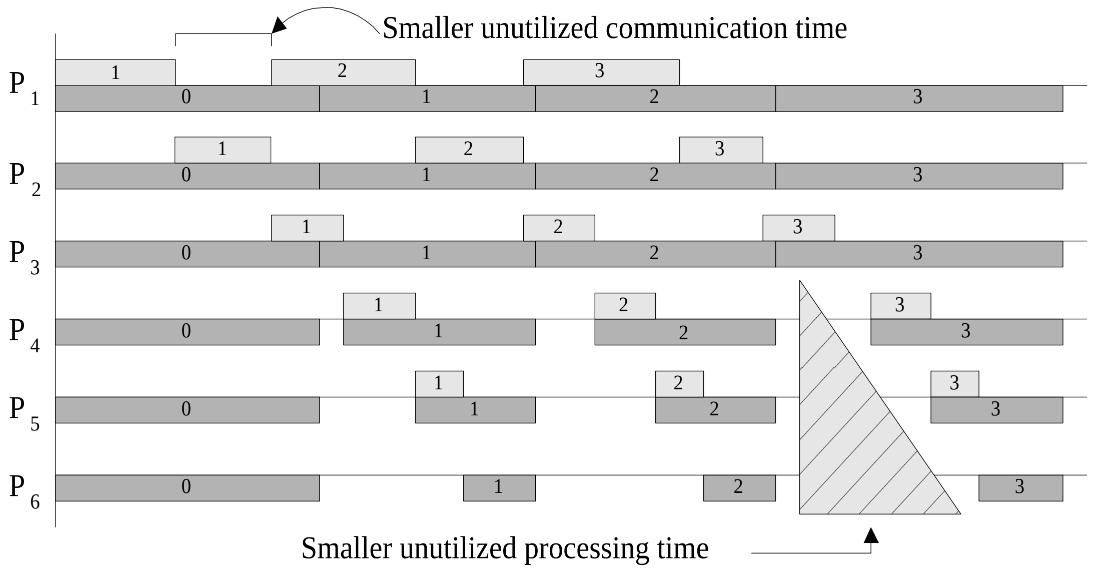

Smaller unutilized communication time
P 1
P 2
P 3
P 4
P 5
P 6
Smaller unutilized processing time

Figure 3.11: Timing diagram for Heuristic A when the loads are sorted(SLF or LLF)

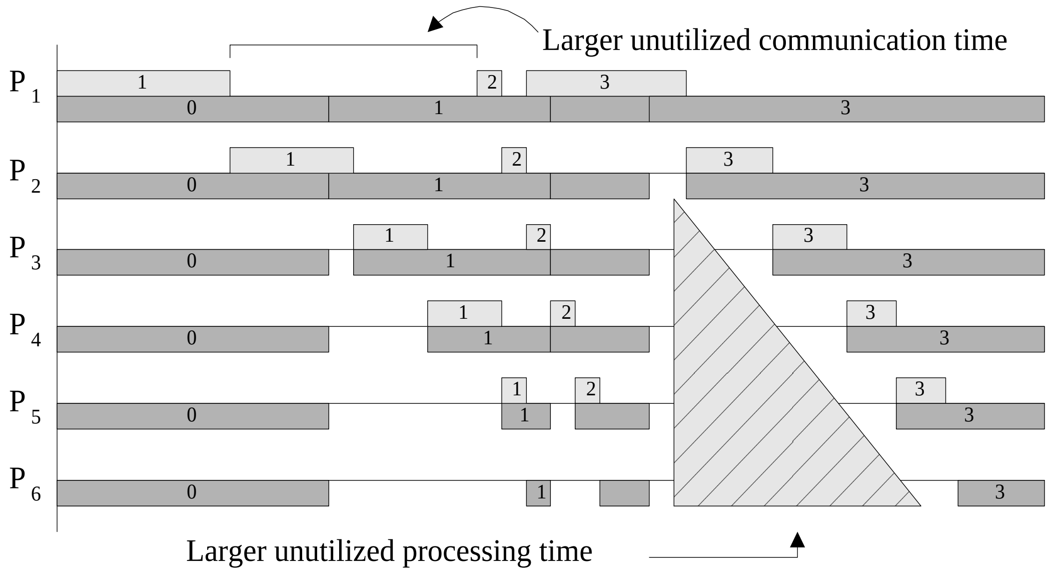

Larger unutilized communication time
P 1
P 2
P 3
P 4
P 5
P 6
Larger unutilized processing time

Figure 3.12: Timing diagram for Heuristic A when the loads are unsorted

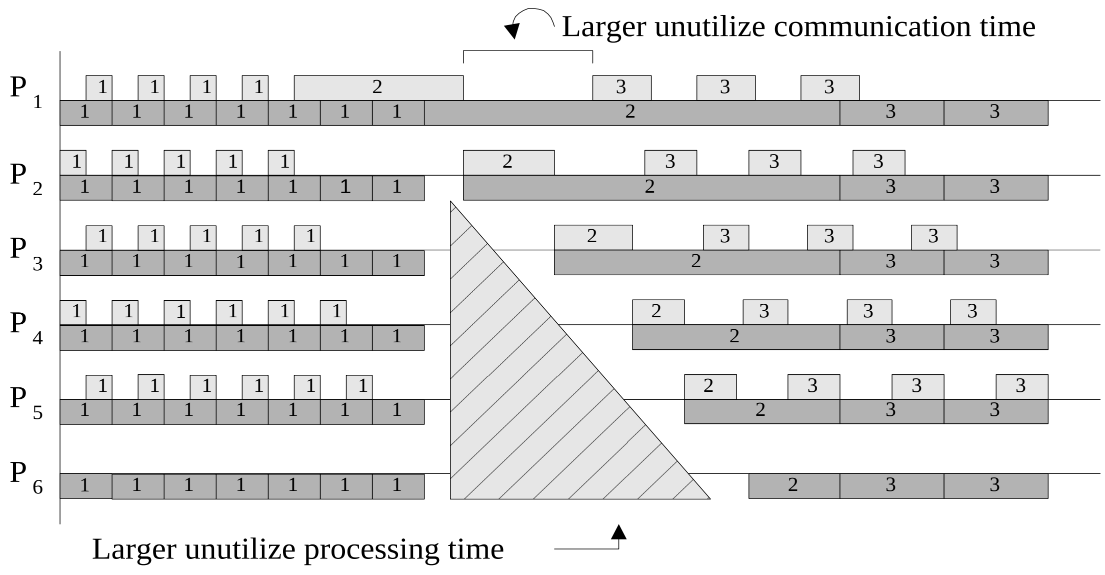

Larger unutilize communication time
P 1
P 2
P 3
P 4
P 5
P 6
Larger unutilize processing time

Figure 3.13: Timing diagram for Heuristic A when heuristic strategy is used in between two optimal distributions

because when the size of the adjacent loads are more-or-less similar, the advantage of Heuristic A, which renders more time to communicate the next load, can be fully exploited. On the other hand, when the sizes of the loads are random(unsorted), large differences in size between adjacent loads will not permit availing this advantage. This is illustrated in Fig. 3.11 and 3.12. As can be seen in the figures, when there is a large difference in size between the adjacent loads, more communication time will be left unutilized and similarly, processors will go idle for more time. The same phenomenon mentioned above also occurs when multi-installment strategy is used after or before Heuristic A. This is illustrated in Fig. 3.13. As a result, our earlier claim that the performance superiority for sorted loads may not be realized when multi-installment strategy is implemented. This can be seen in Fig. 3.9 and 3.10 when there is a sudden deterioration in performance at $T^n_{cp} \approx 14L_n$ for EXA1. The reason is that, when processor speed increases, the probability of satisfying (3.29) also increases. For our system settings and load size variations, (3.29) will be satisfied approximately from $T^n_{cp} = 14L_n$ onwards. Hence, the performance of EXA1 on sorted loads will drop to a level similar to the

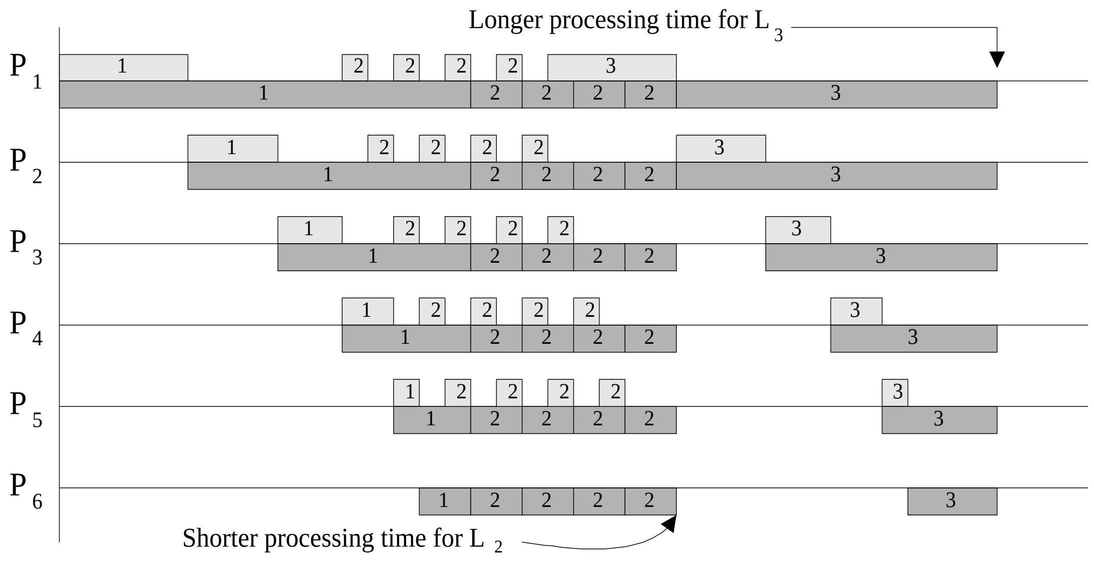

Longer processing time for L_3
P_1
P_2
P_3
P_4
P_5
P_6
Shorter processing time for L_2

Figure 3.14: Timing diagram for EXA1

performance of EXA1 for unsorted loads when the probability of satisfying (3.29) increases up to a certain value.

Notice that from figures 3.9 and 3.10, EXA2 does not suffer a large drop in performance at $T^n_{cp} \approx 16L_n$ as exhibited by EXA1. This is because, in EXA2, single or multi-installment strategies will not be implemented even if conditions allow. Further, we see that EXA2 performs much better when compared to EXA1 for sorted loads in the range $14L_n \leq T^n_{cp} < 38L_n$ although, the optimal distribution is used in EXA1. This is due to the fact that the optimal distribution can significantly deteriorate the performance of Heuristic A, as stated above. This is illustrated in figures 3.14 and 3.15. Note that in Fig. 3.14, a shorter processing time is achieved using an optimal distribution for the 2-nd load, however since the optimal distribution has left less available communication time for the 3-rd load, the overall processing time for these 3 loads is longer. In contrast, from Fig. 3.15, we observe that although the 2-nd load has a longer processing time the overall processing time for these 3 loads is much shorter.

As for Heuristic B, we can see that from figures 3.8, 3.9 and 3.10, EXB1 always gives better

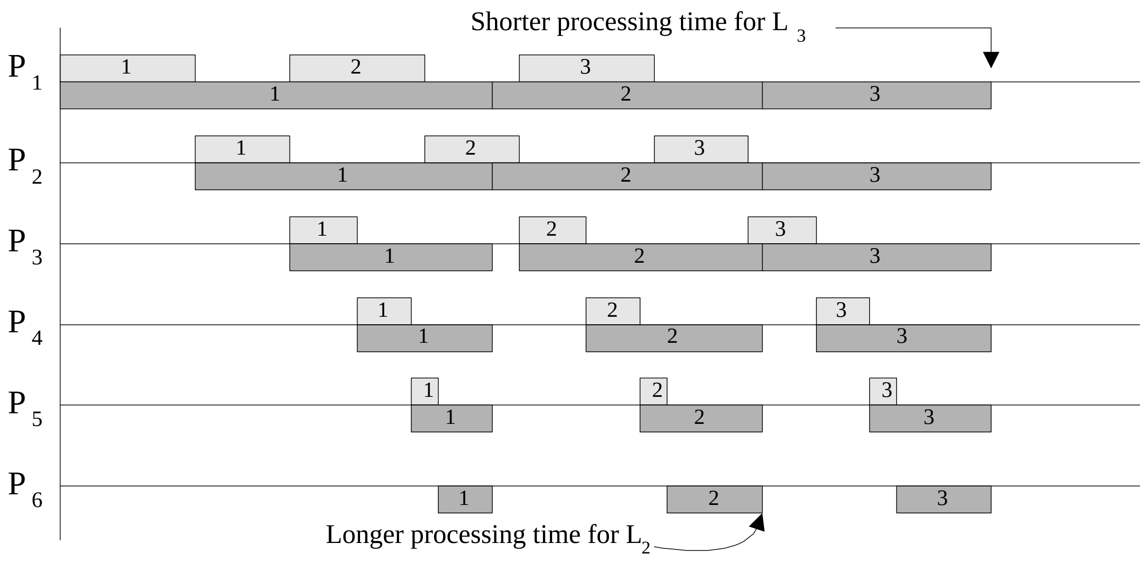

Shorter processing time for L 3
P 1
P 2
P 3
P 4
P 5
P 6
Longer processing time for L2

Figure 3.15: Timing diagram for EXA2

performance when compared to EXB2. This means that, for Heuristic B, better performance can be achieved using smaller values for δT. We conclude that, although EXB2 allows a larger scope to satisfy (3.10) or (3.29), the processing time saved by using an optimal distribution is smaller when compared to the processor time wasted by using a larger δT value. This is illustrated in Fig. 3.16 and Fig. 3.17. Another interesting observation that can made from figures 3.8, 3.9 and 3.10 is the influence of T_{cp}^{n} on processing time. In general, one would expect that as T_{cp}^{n} value increases, the processing time must be increasing, as T_{cp}^{n} fundamentally quantifies the computational time of processors. However, as opposed to this normal behavior, we observe that smaller T_{cp}^{n} values have an effect of increasing the processing time when Heuristic B is used. We prove this rigorously below in Theorem 3.1. Before that we state an important lemma that will be used in the proof of the theorem.

Lemma 3.2: For homogenous systems, $H(i)$ is a non-increasing function of T_{cp}^{n} for any i.

Proof: From (3.18), we have,

$$H(i) = \frac{X_{1,n}^{(i)} C_{1,n} - Y_{1,n}^{(i)}(C_{1,n} + E_{1,n}}{C_{1,n} - Y_{1,n}^{(i)}} \tag{3.43}$$

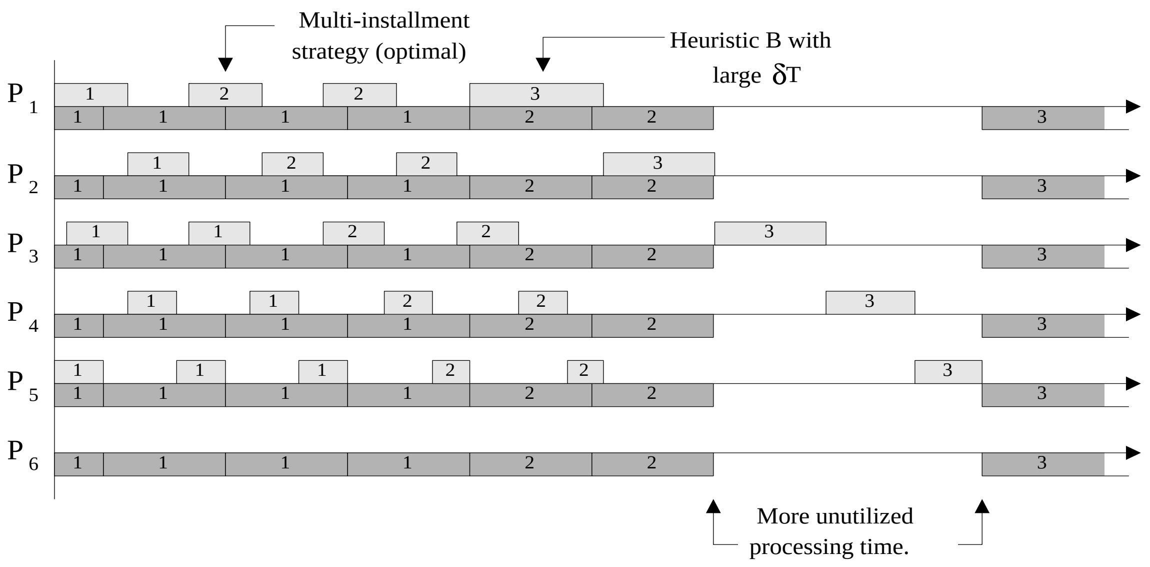


Figure 3.16: Timing diagram showing a large unutilized CPU time when large δT is used

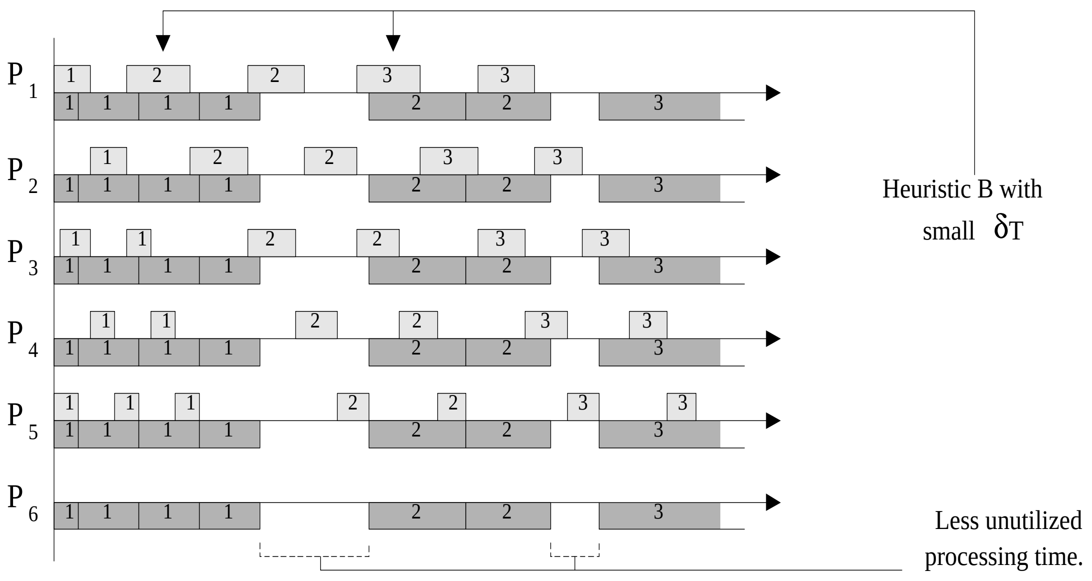


Figure 3.17: Timing diagram showing a better performance with small δT

For homogenous systems, we have $w_i = w_1, \forall i$ and $z_i = z_1, \forall i$. Hence, we can express $X_{1,n}^{(i)}, Y_{1,n}^{(i)}$, and $C_{1,n}$ as,

$$\begin{aligned} X_{1,n}^{(i)} &= \left(\frac{m(m-1)}{2} - \frac{(m-i)(m-i-1)}{2}\right) \alpha_{n,1}^{(1)} z_1 T_{cm}^n \frac{1}{L_n} \\ Y_{1,n}^{(i)} &= \left(\frac{m(m-1)}{2} - \frac{(m-i+1)(m-i+2)}{2}\right) \alpha_{n,1}^{(1)} z_1 T_{cm}^n \frac{1}{L_n} \\ C_{1,n} &= \frac{m(m-1)}{2} \end{aligned}$$

Hence, (3.43) can be expressed as,

$$\begin{aligned} H(i) &= \left(\frac{(m-i+1)(m-i+2)-(m-i)(m-i-1)}{(m-i+1)(m-i+2)}\right)\left(\frac{m(m-1)}{2}\right) \alpha_{n,1}^{(1)} z_1 T_{cm}^n \frac{1}{L_n} \\ &- \frac{m(m-1)-(m-i+1)(m-i+2)}{(m-i+1)(m-i+2)} \alpha_{n,1}^{(1)} w_1 T_{cp}^n \frac{1}{L_n} \end{aligned} \tag{3.44}$$

The above can be then simplified as,

$$H(i) = \left(\frac{m(m-1)(2(m-i)+1)}{(m-i+1)(m-i+2)}\right) \alpha_{n,1}^{(1)} z_1 T_{cm}^n \frac{1}{L_n} - \left(\frac{m(m-1)}{(m-i+1)(m-i+2)} - 1\right) \alpha_{n,1}^{(1)} w_1 T_{cp}^n \frac{1}{L_n} \tag{3.45}$$

Differentiate the above equation with respect to T_{cp}^n, we have,

$$\frac{\delta H(i)}{\delta T_{cp}^n} = \left(1 - \frac{m(m-1)}{(m-i+1)(m-i+2)}\right) \alpha_{n,1}^{(1)} w_1 \frac{1}{L_n} \tag{3.46}$$

For $H(i)$ to be a non-increasing function of T_{cp}^n, the following condition must be satisfied, that is,

$$\frac{m(m-1)}{(m-i+1)(m-i+2)} \geq 1 \tag{3.47}$$

In order to violate the above condition, i must be as small as possible. Nevertheless, since the minimum value allowable for i is 2, the above condition is always satisfied. Hence the proof. □

Theorem 31: Consider two identical linear network systems, say System-A and System-B, that processes a set of loads $L_1^A, ..., L_q^A, L_x$ and $L_1^B, ..., L_q^B, L_y$ respectively, where $L_i^A = L_i^B$, $\forall i = 1, ...q$. Now, whenever $L_x = L_y$ and $T_{cm}^x = T_{cm}^y$, if $T_{cp}^x < T_{cp}^y$, then $T(x) \geq T(y)$ when using Heuristic B for processing L_x and L_y in the respective systems.

Proof of Theorem 3.1:

From (3.40), we have the equation for δT as

$$\delta T > L_n(H - E_{1,n}) - T(n-1) + t_{1,n} \tag{3.48}$$

We denote the δT for T_{cp}^x and T_{cp}^y as $\delta T^{(x)}$ and $\delta T^{(y)}$, respectively. We also denote the H for both cases as $H^{(x)}$ and $H^{(y)}$, respectively. With equality relationship, we have,

$$\begin{aligned}
\delta T^{(x)} &= L_x H^{(x)} - L_x E_{1,x} - T(q) + t_{1,x} \\
\delta T^{(y)} &= L_y H^{(y)} - L_y E_{1,y} - T(q) + t_{1,y}
\end{aligned}$$

Since $t_{1,x}$ and $t_{1,y}$ are dependent on $T(q), L_x$, and L_y we have, $t_{1,x} = t_{1,y}$. Hence,

$$\delta T^{(x)} - \delta T^{(y)} = L_x H^{(x)} - L_x E_{1,x} - L_y H^{(y)} + L_y E_{1,y} \tag{3.49}$$

From Figure 3.6 and 3.7, we can see that, for Heuristic B, $T(N) = T(n-1) + \delta T + L_n E_{1,n}$, hence we have,

$$T(x) - T(y) = \delta T^{(x)} + L_x E_{1,n} - \delta T^{(y)} - L_y E_{1,y} \tag{3.50}$$

Using (3.49) in (3.50), we have

$$T(x) - T(y) = L_x H^{(x)} - L_y H^{(y)} \tag{3.51}$$

Since we have $L_x = L_y$, $T_{cp}^x < T_{cp}^y$ and H is a non-increasing function with respect to T_{cp}^n (from Lemma 3.2), we can see from (3.51) that $T(x) \geq T(y)$. Hence the proof. $\square$

The main reason behind such an anomalous behavior is due to the fact that for smaller T_{cp}^n values, the δT to be used in Heuristic B (in order to satisfy (3.29)) is larger than that of a value to be used when T_{cp}^n is large. Consequently, the amount of processor time wasted (due to idling) will be more in the former case than in the latter case. This results in an increased processing time for smaller T_{cp}^n values.

From the results of our simulations, we see that, in general, for better performance, experiment

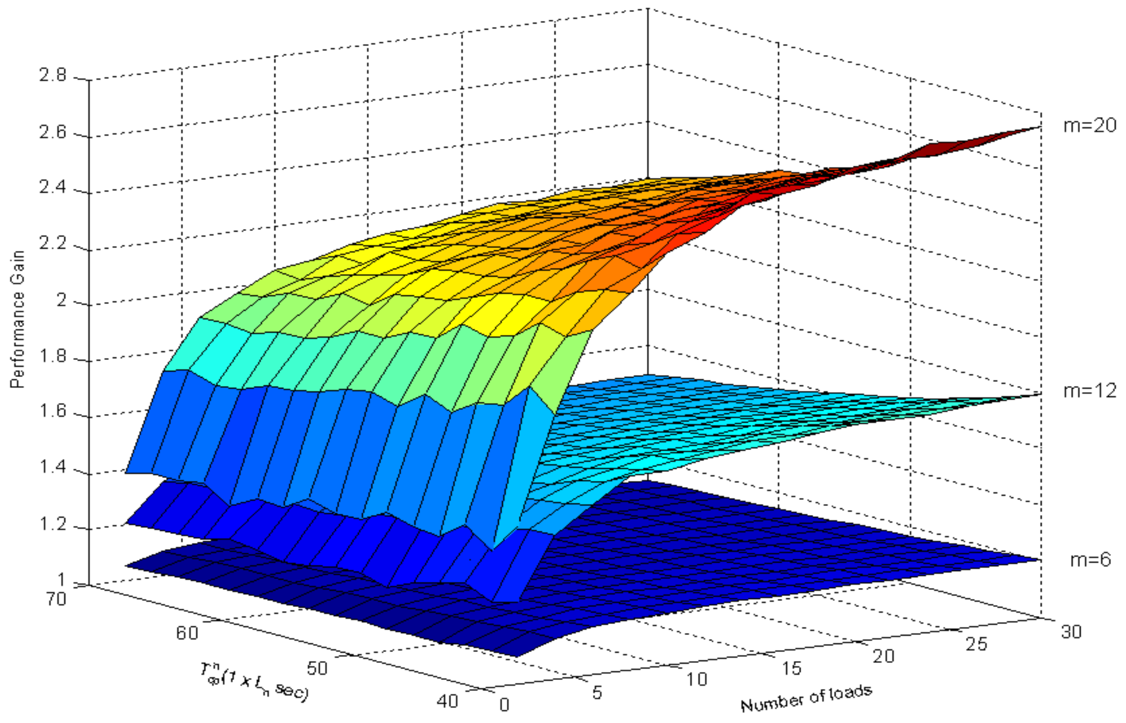


Figure 3.18: Performance gain of multiple loads distribution strategy compared with single load distribution strategy

described in EXA2 is recommended for $T^n_{cp} < 38L_n$, e.g., when (3.29) cannot be satisfied for all loads. Further, optimal distribution can be used when $T^n_{cp} \geq 38L_n$. In general, an optimal distribution can be used when (3.29) can always be satisfied.

Finally, as a natural curiosity, we attempt to use single installment strategy [5, 33] to process all the loads one after other, to reflect the exact performance gain that can be achieved when compare to using the multiple loads strategies discussed in this chapter. In order to compare an optimal distribution for multiple loads strategy with the single load strategy, in the following experiments, we set the parameters of a homogenous system as $T^n_{cm} = 1 \times L_n, \forall n$ sec and $T^n_{cp} = 40L_n, 42L_n, ..., 70L_n$ sec, respectively, as this range yields an optimal distribution for our system settings and parameter ranges mentioned earlier. The simulation is carried out with number of loads, $N = 2, 4, ..., 30$ with three different system each consisting of

$m = 6, 12, 20$, processors respectively. The results of our rigorous simulation is shown in Fig. 3.18. The performance gain can be quantified by the following ratio, defined as,

$$\text{Performance Gain} = \frac{T_{single}(m, N)}{T_{multi}(m, N)} \tag{3.52}$$

where $T_{single}(m, N)$ and $T_{multi}(m, N)$ are the respective total processing times following the single load distribution strategy and the total processing time following the multiple loads distribution strategy, with m processors and N loads. From this figure, we clearly notice that the performance gain increases as the number of loads increases, which shows the benefit of using the multiple loads distribution strategy in dealing with multiple loads. Also, from these plots we can observe that for large values of T_{cp}^{n}, the performance gain deteriorates. This is due to the fact that the processor idling time (wasted time) in the case of single installment strategy is lower when the ratio of communication time to the processing time is smaller. We also observe that the gain achieved increases when m increases, for a given T_{cp}^{n} value.

3.5 Concluding Remarks

In this chapter, the problem of scheduling multiple divisible loads in linear networks is addressed. Previous work [5, 33] for scheduling divisible loads in linear networks assumes that there is only a single load submitted to the system. While this strategy can be applied for scheduling multiple loads by considering one load at a time, the finish time of processing all the loads need not be an optimum. As a result, there is a need for designing distribution strategies specifically for handling multiple loads in linear networks. As mentioned in the Section 3.1, the problem of scheduling multiple loads was addressed for bus networks [18, 40]. Nevertheless, designing load distribution strategies for linear networks is a challenging problem as the data distribution has a pipelined communication pattern involving $(m - 1)$ links, as opposed to a bus network which has a single communication link.

In this chapter, a set of loads is assumed to be resident in the buffer of P_1 (which has a scheduler) and wait to be distributed. We have designed and conducted load distribution strategies to minimize the processing time of all the loads submitted to the system. Since the front-end of each processor cannot send and receive data simultaneously, we have considered the possibility of a "collision" among the front-ends while distributing the loads. In this chapter, we derived a set of conditions that will guarantee a collision-free front-end operation among the adjacent loads for both single and multi-installment strategy. In the case where multi-installment strategy is used, it may happen that a feasible number of installments does not exist and we resolve this situation by using heuristic strategies. Two heuristic strategies, referred to as A and B, are proposed in this chapter for such cases.

The choice of heuristic strategies depends on several issues. For Heuristic A, in Section 3.3, the load will be distributed in single installment. The advantage of distributing the load using a single installment is due to the low time complexity involved. However, for multi-installment strategy, we need to determine the $t_{k,n}$ and $L_{k,n}$ for every installment. Further, using single installment will also renders a considerably longer time for communication for the next load, which will decrease the probability that (3.29) will be violated. The disadvantage of this heuristic is that a considerable CPU time may be left unutilized. Heuristic B, which uses multi-installment strategy, will lower the unutilized CPU time if a small δT is used. Less unutilized CPU time implies less total processing time for the load. Nevertheless, using a small value for δT will increases the number of installments required, and hence, increases the complexity of the heuristic. Further, in the last installment, the total available communication time will also be less, and hence, increases the chances of using heuristic strategies for processing the next load.

We conducted four different simulation experiments (EXA1, EXB1, EXA2, EXB2) mentioned in Section 3.4.1 based on the two heuristic strategies mentioned above to identify the best combination suitable for our multiple loads distribution strategy. Since the processing loads are resident in P_1, we have the option of sorting the given loads with respect to their sizes. We run rigorous simulations for a homogeneous system to evaluate the performance of all these strategies under 3 different policies, that is when the loads are (a) Unsorted (b) Sorted with smallest load first (SLF) (c) Sorted with largest load first (LLF). The simulation results show that experiments described for EXA2 performs better in general, when (3.29) cannot be always be satisfied. Nevertheless, the performance of the general load distribution strategy can be further improved if Heuristic B with small value of δt (EXB1) is use when the probability of satisfying (3.29) is high enough, e.g., approximately $26L_n < T_{cp}^n < 38L_n$ in our simulation set-up. The region of value T_{cp}^n where EXB1 may be implemented to give better performance cannot yet be derived and can be considered as a future extension for this work. Finally, in this chapter, we ran simulations comparing the single load distribution strategy for linear networks with the multiple loads distribution strategy presented in this chapter. The results of these simulations show that significant performance gain can be achieved by using multiple loads distribution strategy, especially when there is a large number of loads to be processed.

Chapter 4

Load Distribution Strategies with Arbitrary Processor Release Times

In the domain of DLT, the primary objective is to determine the load fractions to be assigned to each processor such that the overall processing time of the entire load is minimal. Research efforts was then started focusing on including practical issues. Most of these works are based on the bus and single-level tree network topologies. These studies include, handling multiple loads [18], scheduling divisible loads with arbitrary processor release times [19], use of affined delay models for communication and computation for scheduling divisible loads [20, 21] and scheduling with finite-size buffers [22].

In this chapter, we consider the problem of scheduling arbitrarily divisible loads on linear daisy chain networks with processor release times. This means that, each processor has a release time after which it can be used to process the assigned load. For the first time in the domain of DLT, the problem of scheduling divisible loads on linear networks with processor release times are considered. Our objective is to design efficient load distribution strategies that a *scheduler* can adopt so as to minimize the total processing time of the entire load submitted for processing. We consider the case in which the load originates at the boundary processor (referred to as *boundary case* in the literature) and present a rigorous analysis on the strategies designed to obtain optimal processing time. A similar formulation was considered in the literature in [19] for a bus network architecture. Since a bus network architecture consists of a single communication link, the design of the load distribution strategy, although involves several phases, is fairly straightforward. However, the analysis presented in [19] gives considerable clues to solve the problem addressed in this chapter. It will be clear later from our analysis that the solution procedure for the case of linear networks is by no means a trivial task and offers considerable challenge in designing load distribution strategies for linear networks with arbitrary processor release times.

The organization of this chapter is as follows. Section 4.1 presents the network architecture and various conditions which this chapter will be base on. Also, this section will introduce all the terminology, definitions and notations that are used throughout the chapter. In Section 4.2, we will then present the strategy of solving the problem concerned. When our strategy does not yield an optimal solution, in Section 4.3, we propose few heuristic strategies. Here, we will also include some examples to demonstrate the working style of our strategies. Later, in Section 4.4, we will discuss all the contributions of the chapter. Finally, we will conclude this chapter in Section 4.5 pointing out to some possible extensions to the problem addressed in this chapter.

4.1 Problem Formulation

In this section, we shall introduce the network architecture first and formally define the problem we address. Fig. 2.1 shows a linear daisy chain network architecture consisting of m processors denoted as $P_1, ..., P_m$ connected with $(m-1)$ communication links, denoted as

$l_1, ..., l_{m-1}$. The first (or the last) processor in the chain, also known as the root processor, is assumed to receive the divisible load L to be processed, at time $t = 0$. When the processing load originates at one of these processors, this case is referred to as *boundary case*, whereas when the processing load originates at any other processor in the network, the case is referred to as *interior case*. In this chapter, we shall consider only the boundary case and assume that the processing load originates at processor P_1. In general, the load distribution strategy is as follows.

The processor P_1 (which has a scheduler) upon receiving the load L keeps a load portion $L\alpha_1$ for itself and then sends the remaining $L(1 - \alpha_1)$ portion to P_2. Processor P_2 will keep a portion of the load for itself for processing and communicates the remaining load to P_3 and so on. Note that a processor starts processing its portion only after receiving the entire load from its predecessor and also simultaneously starts communicating the remaining load to its successor. An optimal solution to this problem using an *optimality criterion*, mentioned in Section 1.1 was obtained and in this chapter, we use this criterion in the design of an optimal load distribution strategy. Also, it may be noted that, in the design of a load distribution strategy, we may be attempting to use multiple installments strategy [38]. In the case of multiple installments, the processing load is distributed to the processors in more than one installment. Thus, in this strategy, apart from determining *how many number of installments to be used* to distribute the entire load, we need to determine *how much load to distribute in each installment.* We shall discuss these details later.

In this chapter, we consider the problem of load distribution in linear networks when processor release times are non-zero and are arbitrary in nature. This means that, all the processors are engaged in some form of computation when the load arrives, say at time $t = 0$, and are not available for processing the load from time $t = 0$ onwards. Thus, we are confronted with a problem of designing a load distribution strategy that minimizes the processing time of the entire load by taking into account the arbitrary processor release times of the processors in the problem formulation. We assume that the root processor, i.e., P_1 knows the release times of all the processors in the network. It may be noted that these release times can also be estimates made by P_1 or the processors in the system can explicitly relay their expected release times to the boundary processor P_1 before the start of the load distribution. Note that the processors in the system, knowing the amount of current load to process, can estimate their release times. Thus, it should be clear at this stage that we are concerned with the design of load distribution strategies for scheduling divisible loads after knowing the release times of the processors and we do not address the problem of how these release times are estimated by the processors. This assumption is somewhat similar to the case of bus networks wherein a bus controller unit arbitrates the load distribution process and assumed to know the release times of all the processors in the system.

We shall now introduce the terminology, definitions, and notations that are used throughout the paper.

L	The total load to be processed
L_k	Portion of the load, of size, L assigned in the k-th installment to the processors for processing.
E_i	The time taken to compute a unit load on P_i, where $E_i = w_i T_{cp}$
C_i	The time taken to communicate a unit load over the link l_i, where $C_i = z_i T_{cm}$
$\alpha_{i,k}$	Fraction of the load assigned to P_i in the k-th installment, where $\sum_{i=1}^{m} \alpha_{i,k} = 1$. Note that the actual load assigned to processor P_i in the k-th installment then will be $\alpha_{i,k} L_k$
τ_i	This is the release time of a processor P_i and it is defined as the time instant P_i becomes available for processing the load.
t_k	This is the time instant at which the communication of the load to be distributed (L_k) for the k-th installment is initiated.
$T(q,k)$	This is referred to as the *finish time* of k-th installment and it is defined as the time instant at which the processing of the k-th installment, using q processors, ends.
$T_{process}$	This is referred to as the *processing time* of the entire load and it is defined as the time interval between the instant at which the load arrived to the system (at $t = 0$) and the time instant at which the entire load gets processed.

4.2 Design and Analysis of a Load Distribution Strategy

For the analysis of the single installment strategy when $\tau_i = 0$,$\forall i = 1, ..., m$, readers are referred to [5]. The formulated problem in Section 4.1 of this thesis can be categorized into 2 general cases. The first case is when the release times of all processors are identical and the second case is when the release times are arbitrary. In the following we carry out the design

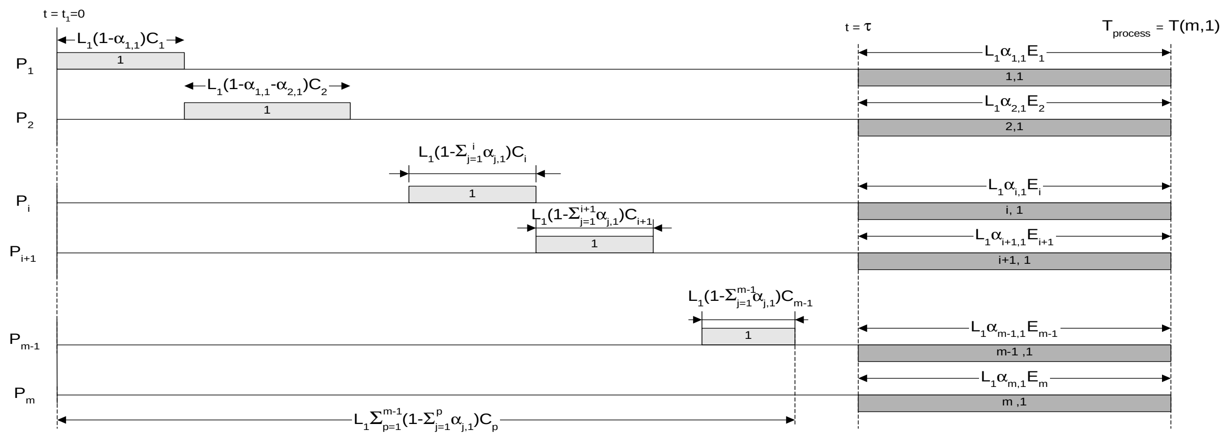


Figure 4.1: Timing diagram for a load distribution strategy when all the load can be communicated before τ.

and analysis for each of these cases separately.

4.2.1 Identical release times

We consider a scenario in which all the processor release times are identical, i.e., $\tau_i = \tau$, $\forall i = 1, ..., m$. Further, it may be noted that the load to be processed may be of any size and hence, the total communication time of the load, starting from time $t = 0$, may or may not exceed the release time τ of the processors. This means that some of the processors may receive their load portions after their release times while others may start exactly from their release times. Consequently, we have to deal with two different scenarios as follows.

Consider a timing diagram shown in Fig. 4.1. In all the timing diagrams used in this chapter, the communication process is shown above the time axis whereas the computation of the processors are shown below the time axis, of each processor. This timing diagram corresponds to the case when the release times of the processors are large enough to accommodate the communication of the entire load to all the processors before τ. Also, note that the load distribution is in such a way that all the processors start processing at time $t = \tau$ and stop at the same instant in time. Thus, we can distribute the entire load in just one installment. We will first derive the exact amount of load portions to be assigned to each processor following this distribution strategy. From the timing diagram shown in Fig. 4.1, we have,

$$\alpha_{i,1}E_i = \alpha_{i+1,1}E_{i+1}, \ i = 1, ..., m-1 \tag{4.1}$$

We can express each of the $\alpha_{i,1}$ in terms of $\alpha_{m,1}$ as,

$$\alpha_{i,1} = \alpha_{m,1}\frac{E_m}{E_i} \tag{4.2}$$

Using the fact that $\sum_{p=1}^{m} \alpha_{i,1} = 1$, we obtain,

$$\alpha_{m,1} = \frac{1}{1 + \sum_{p=1}^{m-1} \frac{E_m}{E_p}} \tag{4.3}$$

Thus, we obtain,

$$\alpha_{i,1} = \frac{\frac{E_m}{E_i}}{1 + \sum_{p=1}^{m-1} \frac{E_m}{E_p}}, \ i = 1, 2, ..., m \tag{4.4}$$

Note that the actual load that is assigned to each P_i is $\alpha_{i,1}L_1$, respectively, where $\alpha_{i,1}$ is given by the above equations and the optimal processing time in this case is given by $T_{process} = T(m, 1) = \tau + \alpha_{m,1}E_m$. Thus, when a load arrives at P_1 at $t = 0$, following condition is verified first.

Case A1: $\tau \geq L\left(\sum_{p=1}^{m-1}(1 - \sum_{j=1}^{p} \alpha_{j,1})C_p\right)$

In the right hand side of the above expression, $\alpha_{j,1}$ is given by (4.4). Note that, the expression on the right hand side is the total communication time of the entire load following the strategy shown in Fig. 4.1. Thus, when a given τ satisfies the above condition, the optimal distribution and the optimal processing time is given by (4.4) and $T_{process}$ derived above, respectively.

Case A2: $\tau < L\left(\sum_{p=1}^{m-1}(1 - \sum_{j=1}^{p} \alpha_{j,1})C_p\right)$

This is the case when the entire load cannot be communicated to all the processors in a

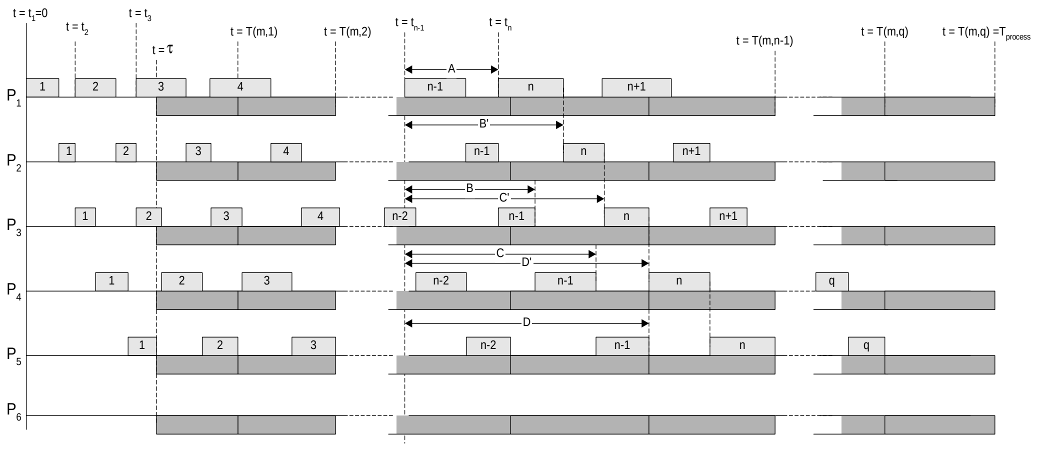


Figure 4.2: Timing diagram showing a collision free front-end operation between installments $n-1$ and n. The numbers inside the block denote the installment number.

single installment before τ, i.e., the load distribution given by (4.4) does not satisfy the above condition. This prompts us to use multiple installments strategy to distribute the load to achieve an optimal processing time. Designing a multiple installment strategy for the case of linear networks with non-zero processor release times is a challenging task. We follow exactly the strategy shown in Fig. 4.2 in the design of multiple installments strategy. That is, for n-th installment, starting from say, time $t = t_n$, we attempt to complete the distribution of a portion of the load L_n before the completion of processing of the $(n-1)$-th installment at time $T(m, n-1)$. There are two issues to be considered in the design of this strategy. Firstly, the number of installments to be used to distribute the entire load and secondly, the amount of load to be assigned in each installment must be derived [38]. Above all, the issue of whether or not an optimal solution exists may arise in certain cases. This is due to the fact that the processor speeds may be extremely faster than the link speeds thus causing the amount of load to be assigned in the consecutive installments to be of smaller and smaller magnitudes. Consequently, the above method of distributing the load may become infeasible. In such situations, we may need to rely on some heuristic strategies. We shall describe some of these strategies later.

Before we describe the multiple installment strategy in general, we shall consider a network comprising of 6 processors and describe the entire load distribution process between two adjacent installments $(n-1)$ and n, respectively, for the ease of understanding. Fig. 4.2 shows the load distribution process for this 6 processors case. Let us assume that the start time t_{n-1} and the amount of load to be distributed L_{n-1} for the $(n-1)$-th installment are known. We need to determine these quantities for the n-th installment. Let us assume that communication for the n-th installment is from $t = t_n$. Thus, the amount of load L_n that can be distributed should not exceed the finish time of the $(n-1)$-th installment, i.e., $T(6, n-1)$. Hence, starting from, say $t = t_n$, we have the following condition to be satisfied.

$$t_n + L_n \left(\sum_{p=1}^{5} (1 - \sum_{j=1}^{p} \alpha_{j,n}) C_p \right) \leq t_{n-1} + L_{n-1} \left(\sum_{p=1}^{5} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right) + L_{n-1}\alpha_{6,n-1}E_6 \quad (4.5)$$

The left hand side of (4.5) gives the amount of communication time needed for the load L_n to be assigned in the n-th installment, whereas the expression on the right hand side is the available communication time from time t_{n-1} onwards. Further, we observe the following from the timing diagram in Fig. 4.2. Starting from time t_{n-1}, for a time interval of A units, the front-end of P_2 will be busy in receiving the load from P_1 and distributing the remaining load to P_3. Thus, the earliest time at which the load distribution for n-th installment can start is only after $t = A$. Consequently, $t_n \geq A$. This is given as,

$$t_n \geq t_{n-1} + L_{n-1} \left(\sum_{p=1}^{2} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right) \quad (4.6)$$

Note that in (4.5) and in (4.6) the parameters t_n and L_n are unknown yet. Equations (4.5) and (4.6) can be solved together to yield t_n and L_n, respectively, using equality relationships. Let us represent the solution obtained above as a pair (t_n^2, L_n^2) and denote this tuple simply as sol_2. The superscript (and the subscript in the sol) in the tuple denotes the index of the processor form which we start comparing for a collision-free front-end operation. Note that L_n^2 gives the maximum amount of load that can be communicated within the available communication time. For consistency reasons, we denote $sol_1 = (t_n^1, L_n^1)$, with $t_n^1 = L_n^1 = 0$.

It may be noted that sol_2 may result in a scenario in which the operations of the front-ends of the successive processors may collide, as we have only considered the possibility of a collision-free front-end operation of P_2 with respect to P_1 and P_3. So, we now need to account for a collision-free operation of P_3's front-end. From the diagram, we observe from the start of reception of the load from P_2, the front-end of P_3 will be busy for an interval of $(B-A)$ units, where B is the time interval between t_{n-1} and finish time of communication of P_3. Thus, in order to have a collision-free front-end operation, we need to have $B' \geq B$, where B' is the start time of communication of n-th installment for P_3. This yields,

$$t_n + L_n(1-\alpha_{1,n})C_1 \geq t_{n-1} + L_{n-1}\left(\sum_{p=1}^{3}(1-\sum_{j=1}^{p}\alpha_{j,n-1})C_p\right) \tag{4.7}$$

Similar to procedure for obtaining sol_2 above, we can solve (4.5) and (4.7) to yield another pair given by $sol_3 = (t_n^3, L_n^3)$. Similarly, considering a collision free operation of the front-end of P_4, from the timing diagram, we have,

$$t_n + L_n\left(\sum_{p=1}^{2}(1-\sum_{j=1}^{p}\alpha_{j,n})C_p\right) \geq t_{n-1} + L_{n-1}\left(\sum_{p=1}^{4}(1-\sum_{j=1}^{p}\alpha_{j,n-1})C_p\right) \tag{4.8}$$

Following the same above steps, we can solve (4.5) and (4.8) to yield another pair given by $sol_4 = (t_n^4, L_n^4)$. Thus, following this procedure, we obtain $(m-2)$ tuples, each accounting for a collision-free front-end operation, starting from P_2 until processor P_{m-1}. On the whole, for obtaining a load distribution that results in a collision-free front-end operation within the available communication time, we must use the following value of t_n.

$$t_n = max\{t_n^i \mid t_n^i \in sol_i,\ i = 2,3,4,5\} \tag{4.9}$$

Note that the value obtained in (4.9) guarantees a collision-free scenario, as all the load that was percolating down the network from the previous installment would have been completed before any processor communicates the next installment to its successor. Thus, the value of L_n is given by the corresponding value in the tuple that yields a maximum t_n given by (4.9). In general, for a m processor system, we can generalize (4.5) as,

$$t_n + L_n \left(\sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_{j,n}) C_p \right) \leq$$
$$t_{n-1} + L_{n-1} \left(\sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right) + L_{n-1}\alpha_{m,n-1}E_m, \; n \neq 1 \tag{4.10}$$

Following the above argument for a collision-free front-end operation scenario, starting from P_2 till P_{m-1}, we have the following set of inequalities for the respective cases.

$$t_n + L_n \left(\sum_{p=1}^{i-2} (1 - \sum_{j=1}^{p} \alpha_{j,n}) C_p \right) \geq t_{n-1} + L_{n-1} \left(\sum_{p=1}^{i} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right), \; i = 2, ..., m-1 \tag{4.11}$$

Note that (4.11) gives rise to $(m-2)$ inequality relationships, one for each value of i. Each inequality relationship together with (4.10) needs to be solved to yield a solution sol_r, $r = 1, ..., m-2$, as described in detail for the 6 processors case above. The value of t_n that is to be used is then given by (4.9) (with $i = 2, 3, ..., m-1$) and the value of L_n is given by the corresponding value in the tuple that yields a maximum t_n . The above procedure can be simplified by first solving (4.10) and (4.11) for each value of t_n^i, $\forall i = 2, ..., m-1$. Thus, t_n^i is given by

$$t_n^i = t_{n-1} + L_{n-1} \frac{X^{(i)}M_c - Y^{(i)}(M_c + M_e)}{M_c - Y^{(i)}}, \; i = 2, ..., m-1 \tag{4.12}$$

where $M_c = \left(\sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_{j,1}) C_p \right)$, $M_e = \alpha_{m,1}E_m$ and

$$X^{(i)} = \left(\sum_{p=1}^{i} (1 - \sum_{j=1}^{p} \alpha_{j,1}) C_p \right) \text{ and } Y^{(i)} = \left(\sum_{p=1}^{i-2} (1 - \sum_{j=1}^{p} \alpha_{j,1}) C_p \right), \; i = 2, ..., m-1 \tag{4.13}$$

Note that we have replaced $\alpha_{j,n}$ and $\alpha_{j,n-1}$ with $\alpha_{j,1}$ since these are the proportions in which the load L_n will be distributed among m processors in n-th installment and remain identical in every installment. $\alpha_{j,1}$ is given by (4.4). We denote

$$H(i) = \left(\frac{X^{(i)}M_c - Y^{(i)}(M_c + M_e)}{M_c - Y^{(i)}} \right), \; i = 2, ..., m-1 \tag{4.14}$$

and

$$H = max\{H(i)\}, \forall \; i = 2, ..., m-1 \tag{4.15}$$

Then, the value of t_n given by (4.9) with $i = 2, ..., m-1$, can be expressed as,

$$t_n = t_{n-1} + L_{n-1}H \tag{4.16}$$

It may be noted that the above equation (4.16) can be used to compute the values of t_n by using the previous values and the value of H. The value of H may be pre-computed as it involves determining the values of $H(i)$ for all i, which are essentially constants. Although $H(i)$ is an expression comprising M_c, M_e, $X^{(i)}$ and $Y^{(i)}$ (which are functions of $\alpha_{i,1}$), these are the $\alpha_{i,1}$ that are derived using (4.4) and are fixed. Thus, parameters comprising H(i) are constants. The value of L_n can be calculated using t_n by using the following equation, which is obtained by solving (4.10) and (4.11).

$$L_n = \frac{(t_{n-1} - t_n) + L_{n-1}(M_c + M_e)}{M_c} \tag{4.17}$$

At this juncture, for any installment k, it may be verified that the value of H derived above remains identical and can be pre-computed as it is essentially a function of speed parameters of the system.

The above procedure described the process of determining the start times of the installments and the amount of load that needs to be assigned in each installment. It may be recalled that in each installment, we follow the strategy that is shown in Fig. 4.2. Thus, the amount of load a processor P_i will be assigned in the k-th installment is given by $\alpha_{i,k}L_k$, where $\alpha_{i,k} = \alpha_{i,1}$ is given by (4.4). The finish time k-th installment is then given by $T(m,k) = T(m,k-1) + L_k\alpha_{m,k}E_m$. Note that, from Fig. 4.2, the total amount of load that can be distributed in the first installment $L_1 = \frac{\tau}{M_c}$. As a final remark, we note the following. Suppose if we know the number of installments K to be used to distribute the entire load following the above strategy, then for the last installment K, the amount of load that is left unprocessed is given by $L_K = \left(L - \sum_{j=1}^{K-1} L_j\right)$. Using this value of L_K in (4.17) (with $n = K$), we can immediately obtain the value of t_K, the start time of the last installment, without following the above procedure. Now, a final question that is left unanswered with our analysis so far is on the number of installments required to distribute the entire load L, which we address in the following section.

4.2.2 Calculation of an optimal number of installments

This section presents a method used to calculate the optimal number of installments required, if it exists, to process the entire load L and also derive some important conditions to ensure that the load L will be able to be processed in a finite number of installments. We shall now assume that we need K installments to distribute the entire load L to be processed and determine the conditions under which such a value of K may exist. To derive this value of K, we start by solving (4.16) and (4.17) to obtain a relationship between L_{n-1} and L_n. Thus, L_n is given by

$$L_n = \frac{L_{n-1}}{M_c} B \tag{4.18}$$

where $B = M_c + M_e - H$. Using the above relationship, we can express each of $L_i,\ i = 2, ..., K$ in terms of $L_1 = \frac{\tau}{M_c}$, as,

$$L_i = \left(\frac{\tau}{M_c^i}\right) B^{i-1} = \left(\frac{\tau}{B}\right)\left(\frac{B}{M_c}\right)^i \tag{4.19}$$

Note that since L_i is the fraction of the load L, and if K is the last installment, it is obvious that $\sum_{j=1}^{K} L_j = L$. We note that,

$$\left(\frac{\tau}{B}\right)\left(\frac{B}{M_c}\right)\sum_{j=0}^{K-1}\left(\frac{B}{M_c}\right)^j = L \tag{4.20}$$

from which we obtain,

$$\left(\frac{\tau}{B}\right)\left(\frac{B}{M_c}\right)\left(\frac{(\frac{B}{M_c})^K - 1}{(\frac{B}{M_c}) - 1}\right) = L \tag{4.21}$$

Simplifying the above expression, we obtain,

$$K = \left(\frac{ln\left(\frac{\tau + L(B - M_c)}{\tau}\right)}{ln\left(\frac{B}{M_c}\right)}\right) \tag{4.22}$$

Now, from the above expression, for $K > 0$, $\tau + L(B - M_c) > 0$, where B is as defined above. Equivalently, we have the following relationship to be satisfied.

$$\tau > L(H - M_e) \tag{4.23}$$

The above condition must be satisfied in order to obtain a feasible value of K. Thus, when the above condition is satisfied, we distribute the load in K installments. However, it may happen that the above condition may be violated and no feasible value of K may exist. In this case, we use heuristic strategies, which shall be discussed later, to complete the distribution of the load. In the next section, we shall analyze the case when the processor release times are arbitrary.

4.2.3 Non-identical release times

In this section, we shall present the analysis for the case of non-identical processor release times. Let $I = \{1, 2, ..., m\}$ denote a set of indices of the processors P_1 to P_m. Also, let us denote $l = argmax\{\tau_j : j \in I\}$ and $s = argmin\{\tau_j : j \in I\}$, respectively. In other words, τ_l is the release time of the processor with the largest release time and its speed is E_l. Similarly, τ_s is the release time of the processor with the smallest (earliest) release time and its speed is E_s. Similar to the identical case, here too, we have the following two distinct cases to be analyzed. We first attempt to follow a *conservative* approach, referred to as a *conservative* strategy hereafter, for the first case. In case this conservative strategy cannot be used, we attempt a *multi-installment* approach.

Conservative strategy

In this strategy, we attempt to identify a maximal number of processors that can participate starting from their respective release times and use single-installment strategy to complete processing of the entire load. Consider a load distribution strategy shown in timing diagram in Fig. 4.3. Further, using this load distribution $\alpha_{1,1}, ...\alpha_{m,1}$, we observe that all the m processors start computing from their respective release times and stop computing at the same instant in time. This optimal load distribution can be derived from the timing diagram

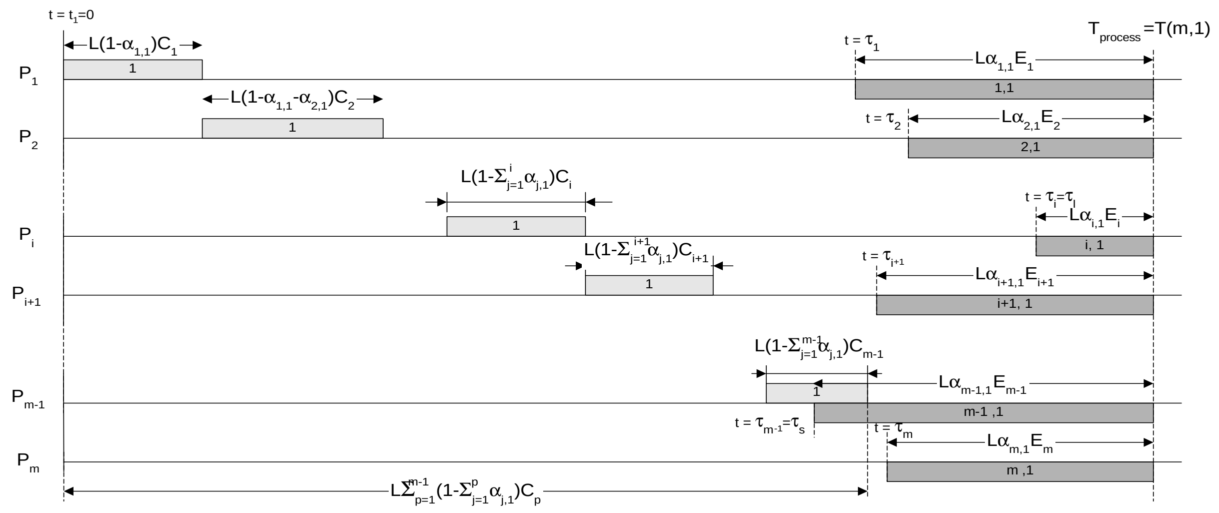


Figure 4.3: Timing diagram for conservative strategy.

as follows. From Fig. 4.3 we obtain the following equations.

$$L\alpha_{i,1} = \frac{\tau_l - \tau_i}{E_i} + \frac{E_l}{E_i} L\alpha_{l,1}, \;\; i = 1, ..., m \tag{4.24}$$

Together with the equation $\sum_{j=1}^{m} \alpha_{j,1} = 1$, we obtain,

$$L\alpha_{l,1} = \frac{L - \sum_{i=1}^{m} \frac{\tau_l - \tau_i}{E_i}}{\sum_{i=1}^{m} \frac{E_l}{E_i}} \tag{4.25}$$

Now, from (4.25) we observe that in order to utilize all the processors in the network a *necessary and sufficient* condition is given by,

$$L > \sum_{i=1}^{m} \frac{\tau_l - \tau_i}{E_i} \tag{4.26}$$

Thus, when the above condition (4.26) is violated by P_l, we eliminate P_l from participating in the computation and assign the total load to the remaining processors using the above strategy. Thus, we iteratively use the above condition until we obtain a *maximal* set of processors (m^*) to be used for computation. We refer to this maximal set of processors simply as a *qualifying set of processors*, hereafter. Note that in every iteration, while using the above condition (4.26), we replace m with the number of processors taking part in the computation in this current iteration. Further, also observe that, we need to determine l, the index of the processor that has the largest release time from the set of processors participating in the computation, in each iteration. The following example illustrates this procedure.

Example 4.1. Consider a linear network $m = 6$ processors, with the processing speeds given by, $E_1 = 5, E_2 = 10, E_3 = 5, E_4 = 10, E_5 = 5$, and $E_6 = 10$, respectively. Let the link speeds be $C_1 = 1, C_2 = 2, C_3 = 1, C_4 = 2, C_5 = 1$, respectively. All processors have arbitrary release times given by, $\tau_1 = 4.2, \tau_2 = 4.6, \tau_3 = 8.0, \tau_4 = 4.0, \tau_5 = 7.0, \tau_6 = 5.0$, respectively. Let the total load be $L = 1$.

First, we identify $\tau_s = \tau_4 = 4.0$ and $\tau_l = \tau_3 = 8.0$. We then note that the condition (4.26) is violated ($\sum_{i=1}^{6} \frac{\tau_l - \tau_i}{E_i} = 0.76 + 0.34 + 0 + 0.4 + 0.2 + 0.3 = 2 > L$) and hence, we eliminate P_3 from computation. Now, from the available set of processors, we note that $\tau_l = \tau_5 = 7.0$. We note that the condition (4.26) is still violated and hence, we eliminate P_5 from participating in computation. Next, we note that $\tau_l = \tau_6 = 5.0$. We now verify (4.26) and observe that it satisfies. Thus, the entire load is distributed to 4 processors $m^* = 4$ processors and these are P_1, P_2, P_4 and P_6, respectively. The optimal load distribution following the above strategy is given by, $\alpha_{1,1} = 0.44, \alpha_{2,1} = 0.18, \alpha_{4,1} = 0.24, \alpha_{6,1} = 0.14$. The processors P_2, P_4, and P_6, will receive their load portions at times $0.56, 1.70$, and 2.12, respectively (before their release times). Hence they can process their respective load portion starting from their release times. The optimal processing time in this example is given by, $T_{process} = T(4, 1) = 6.4$. □

The above example demonstrates the procedure to determine a set of qualifying processors and shows how the load is distributed among this set. It is important to realize that the conservative strategy need not guarantee that all the processors can start at their respective release times for processing their load fractions. In other words, we shall now show that conservative strategy need not produce an optimal solution as shown in Fig. 4.3. To demonstrate this fact, we note that the load fractions recommended by (4.24) do not consider the actual communication time of these load fractions to the respective processors. All that the above derivation of $\alpha_{j,1},\ j = 1, ..., m$ considers is only the computation part. Hence, even if (4.26) satisfies for some $r \in \{1, ..., m\}$ resulting in a maximal set of processors, there is no guarantee that each processor will receive its portion on or before its release time. Hence, the condition stated in (4.26) is not alone sufficient to assure a load distribution that generates an optimal solution. Hence, if $S = \{P_r\},\ r \in \{1, ..., m\}$ is the qualifying set of processors for computing the load after satisfying (4.26), the following set of conditions need to be verified for each $P_i \in S$.

$$\tau_i > L\left(\sum_{p=1}^{i-1}(1 - \sum_{j=1}^{p}\alpha_{j,1})C_p\right), P_i \in S \tag{4.27}$$

Thus, once (4.27) holds for each $P_i \in S$, the resulting solution is *optimal*. It may be verified that in Example 4.1, for the set of qualifying processors P_1, P_2, P_4, P_6, the above conditions given in (4.27) are satisfied and hence, yields an optimal solution. It is important to note that in the above expression, the number of processors participating r may not be physically adjacent to each other, and hence care must be taken while verifying the condition (4.27). Also, note that for processors not participating, $\alpha_{j,1} = 0$ and these processors only involve in communicating the load to their successors. However, following example demonstrates a scenario in which conservative strategy may not generate optimal load distribution as shown in Fig. 4.3.

Example 4.2. Consider a linear network in *Example 4.1*, however, let their release times be $\tau_1 = 1.05, \tau_2 = 1.15, \tau_3 = 2.0, \tau_4 = 3.0, \tau_5 = 1.75, \tau_6 = 1.25$, respectively. Let $L = 1$.

Using the above procedure to generate a set of qualifying processors, we have $\tau_l = \tau_3 = 2.0$ (P_4 has been eliminated from the set), which satisfies (4.26), as $\sum_{i=1}^{5}\frac{\tau_l - \tau_i}{E_i} = 0.19 + 0.085 + 0 + 0.05 + 0.075 = 0.4 < L$. Nevertheless, if we calculate the communication times for each of the load fractions to the respective processors, we observe that for $L\alpha_{5,1}$ to reach P_5, the total communication time will be $\left(\sum_{p=1}^{4}(1 - \sum_{j=1}^{p}\alpha_{j,1})C_p\right) = 0.66 + 1 + 0.35 + 0.7 = 2.71$, which is larger than the release time of P_5 ($t_5 = 1.75$). In other words, the load fraction arrives at P_5 only after τ_5, and hence, the processor time available in the time interval between τ_5 and the arrival of $L\alpha_{5,1}$ will be left unutilized. □

Thus, from this example we note that conservative strategy fails to generate an optimal solution as shown in the timing diagram Fig. 4.3.

Remarks: When condition in (4.27) is violated for a set of qualifying processors after satisfying condition (4.26), it is natural to attempt to eliminate a processor with release time τ_l and continue with the conservative strategy with $(r-1)$ processors. Note that we use single installment strategy in our conservative strategy proposed above. However, this may not lead to an optimal solution, because it may happen that the total processing time with $(r-1)$ processors using conservative strategy may be more than the total processing time with r processors using multi-installment strategy. This clearly motivates us to adopt a multi-installment strategy.

Multi-installment strategy

We first prove the following lemma that will be used in the design of this multi-installment strategy.

Lemma 4.1: Let $Q = \{P_i\},\ i \in \{1, ..., m\}$ denote a set of processors sorted in the order of increasing release times in the network, i.e., $\tau_j \leq \tau_{j+1},\ j \in Q$. Also, let $S \subseteq Q$, with $|S| = r,\ r \leq m$, denote a set of qualifying processors for computing the load after satisfying (4.26) and (4.27) such that $\tau_j \leq \tau_{j+1},\ \forall P_j \in S$. Obviously, $r = argmax\{\tau_j : P_j \in S\}$. Then, $T_{process} = T(r,1) \leq \tau_{r+1}$, where $P_{r+1} \in Q/S$.

Proof: When τ_l does not satisfy (4.26), we will have the following condition.

Rearranging the above, we obtain the following.

$$\left(\frac{L + \sum_{i=1}^{r} \frac{\tau_i}{E_i}}{\sum_{i=1}^{r} \frac{1}{r}}\right) < \tau_{r+1} \tag{4.28}$$

We shall prove this lemma by contradiction. Suppose that $T(r,1) > \tau_{r+1}$. Then we will have,

$$\tau_r + E_r L\alpha_r > \tau_{r+1}$$

Using (4.25) in the above equation, we obtain,

$$\left(\frac{L + \sum_{i=1}^{r} \frac{\tau_i}{E_i}}{\sum_{i=1}^{r} \frac{1}{r}}\right) > \tau_{r+1} \tag{4.29}$$

Comparing (4.28) and (4.29), we see a clear contradiction, thus proving the lemma. □

The significance of the lemma is as follows. When a set of processors qualify for processing a certain amount of load after satisfying (4.26) and (4.27) conditions, these processors are assigned respective load fractions given by (4.24). These processors start computing from their respective release times and complete at a time $T(r,1)$. The above lemma shows that this finish time will not exceed the earliest release time of a processor among the set of non-qualified processors. This result will be widely used in the design of our multi-installment strategy in a recursive fashion as it highlights the fact that the earliest release times of the processors that qualify in any particular installment are indeed equal to the finish times of the processors in the previous installment and may be at most equal to the earliest release time of a processor among the set of non-qualified processors.

We are now set to design the multi-installment strategy. As a first step, for the first installment, we need to determine the amount of load that can be assigned before time τ_s such that a set of processors start at their respective release times and stop computing at the same time instant. In order to do so, solve the following set of equations.

$$L_1\alpha_{i,1} = \frac{\tau_l - \tau_i}{E_i} + \frac{E_l}{E_i} L_1 \alpha_{l,1},\ i = 1, ..., m \tag{4.30}$$

$$\tau_s = \left(\sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_j) C_p L_1\right) \tag{4.31}$$

Note that (4.30) is identical to (4.24) except L is replaced with L_1. The right hand side of (4.31) is equal to the total communication time available to transmit a load L_1 to all the processors. Thus, the amount of load L_1 should be chosen in such a way that it can be communicated on or before time τ_s. Hence, we equate these two quantities. Using (4.30), we rewrite (4.31) as,

$$\tau_s = \sum_{j=2}^{m}\left(\sum_{i=j}^{m}(\frac{\tau_l - \tau_i}{E_i})C_{j-1}\right) + \sum_{j=2}^{m}\left(\sum_{i=j}^{m}(\frac{C_{j-1}E_l}{E_i})\right) L_1\alpha_{l,1} \tag{4.32}$$

From the above equation, we can obtain $L_1\alpha_{l,1}$

$$L_1\alpha_{l,1} = \frac{\tau_s - \sum_{j=2}^{m}\left(\sum_{i=j}^{m}(\frac{\tau_l-\tau_i}{E_i})C_{j-1}\right)}{\sum_{j=2}^{m}\left(\sum_{i=j}^{m}(\frac{C_{j-1}E_l}{E_i})\right)} \tag{4.33}$$

Note that the above equation (4.33) is similar to (4.25). Thus, from (4.33), we obtain a *necessary and sufficient* condition for utilizing all the m processors to participate in the computation, as follows.

$$\tau_s > \sum_{j=2}^{m}\left(\sum_{i=j}^{m}(\frac{\tau_l - \tau_i}{E_i})C_{j-1}\right) \tag{4.34}$$

Thus, as done in the conservative strategy, we recursively use this condition to eliminate the redundant processors and work with a subset of processors that qualify for computation. Hence, at the end of this process, all the processors in the qualified set will start at their respective release times and stop computing at the same time. From second installment onwards, we follow a similar methodology used for the case of identical release times, to compute the start-up times for load distribution in every installment satisfying a collision-free front-end operation, as follows.

We shall now describe the load distribution process between two adjacent installments $(n-1)$ and n, respectively, for the ease of understanding. Let us assume that the start time t_{n-1} and the amount of load to be distributed L_{n-1} for the $(n-1)$-th installment are known. We need to determine these quantities for the n-th installment. Let us assume that communication for the n-th installment is from $t = t_n$. Firstly, we have to ensure that the total communication time for the load L_n for the n-th installment should be completed on or before the completion of processing of L_{n-1}. Hence, we have the following condition to be satisfied.

$$t_n + \left(\sum_{p=1}^{m-1} (L_n - \sum_{j=1}^{p} \alpha_{j,n} L_n) C_p \right) \leq t_{n-1} + L_{n-1} \left(\sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right) + L_{n-1}\alpha_{s,n-1} E_s \quad (4.35)$$

Note that this equation is similar to (4.10) except that we use $L_{n-1}\alpha_{s,n-1}$ and E_s instead of $L_{n-1}\alpha_{m,n-1}$ and E_m, respectively. Secondly, in order to ensure a collision-free front end operation, similar to the procedure described for identical case, we will solve (4.35) and (4.11) (using equality relationships) for every $i = 2, ...m-1$, for L_n to obtain,

$$\left(\sum_{p=1}^{i-2} (L_n - \sum_{j=1}^{p} \alpha_{j,n} L_n) C_p \right) - \left(\sum_{p=1}^{m-1} (L_n - \sum_{j=1}^{p} \alpha_{j,n} L_n) C_p \right) \geq$$
$$L_{n-1} \left(\sum_{p=1}^{i} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p \right) - L_{n-1}(M_c + M_e), \ i = 2, ..., m-1 \quad (4.36)$$

where, $M_c = \sum_{p=1}^{m-1} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p$ and $M_e = \alpha_{s,n-1} E_s$. We rewrite (4.30) as,

$$L_n \alpha_{i,n} = \frac{\tau_l - \tau_i}{E_i} + \frac{E_l}{E_i} L_n \alpha_{l,n}, \ i = 1, ..., m \quad (4.37)$$

Substituting (4.37) into (4.36), and after some algebraic manipulations, we obtain, for $i = 2, ..., m-1$,

$$L_n \alpha_{l,n} \leq \frac{L_{n-1}\left(M_c + M_e - \left(\sum_{p=1}^{i} (1 - \sum_{j=1}^{p} \alpha_{j,n-1}) C_p\right)\right) - \sum_{j=i}^{m} \left(\sum_{p=j}^{m} (\frac{\tau_l - \tau_p}{E_p}) C_{j-1} \right)}{\sum_{j=i}^{m} \left(\sum_{p=j}^{m} (\frac{C_{j-1} E_l}{E_p}) \right)} \quad (4.38)$$

Similar to the identical release times case, for a collision-free front end operation, (4.38) will yield $(m-2)$ values of $L_n\alpha_{l,n}$, one for each $i = 2, ..., m-1$. If there is one or more negative values in the set, then this is implies that the load L_n is insufficient to cater to all the processors owing to their large release times. Hence, we exclude P_l from taking part in computation of the load L_n in this n-th installment and repeat this process with fewer number of processors as above. On the other hand, if all the $(m-2)$ $L_n\alpha_{l,n}$ values in the set are positive numbers, then we will choose the minimum $L_n\alpha_{l,n}$ value among the set, as the load to be distributed in the n-th installment. From this chosen $L_n\alpha_{l,n}$ value, we can then calculate other $\alpha_{j,n}$ from (4.37) immediately. Thus, assuming that there are, say r processors participating in the n-th installment, since $\sum_{i=1}^{r} \alpha_{i,n} = 1$, we can readily obtain L_n.

Important remark: It may be noted that if the remaining load $(L - \sum_{i=1}^{n-1} L_i < L_n)$, is less than the calculated load using the above procedure, we will use the following equation to find the $\alpha_{l,n}$ for the remaining load.

$$L_n \alpha_{l,1} = \frac{L - \sum_{i=1}^{n-1} L_i - \sum_{i=1}^{m} \frac{\tau_l - \tau_i}{E_i}}{\sum_{i=1}^{m} \frac{E_l}{E_i}} \tag{4.39}$$

Note that (4.39) is similar to (4.25). If (4.39) gives a negative value for $\alpha_{l,n}$, then we will exclude P_l from participating, and work with less number of processors until we obtain a positive $\alpha_{l,n}$. Then, t_n, the starting time of the n-th installment, can be calculated using (4.35). Thus, the above procedure determines the exact start time t_n of the load distribution for the n-th installment (assuming t_{n-1} is known) and the individual load fractions $\alpha_{j,n}$ $j = 1, ..., r$ (assuming r processors are participating in this n-th installment).

As carried out for identical release times case, it will be of immense interest to determine the number of installments that should be used to complete the processing of the entire load. However, deriving this is by no means a simple task, as the number of processors that can participate in any installment is not known in advance, owing to their arbitrary release times. Thus, we propose the following methodology to "sense" whether or not to carry out the multi-installment strategy, or to use any heuristic method to complete the processing of the entire load. Let us consider the n-th installment wherein only r processors are qualified in processing the load L_n and denote this set of r processors as R. Suppose we assume that only processors in R will be involved in processing all the load L (starting from n-th installment), then we can calculate the number of installments required as done in the case of identical release times analysis. First of all, we can calculate $\alpha_{i,n+1}$ for all the processors in R as follows.

$$\alpha_{i,n+1} = \frac{1}{E_i \sum \frac{1}{E_i}}, \ P_i \in R \tag{4.40}$$

The above equation is similar to (4.4), but only involves the processors in R. For other processors, which is not in R, we have, $\alpha_{j,n+1} = 0, \ P_j \notin R$. Hence, all the $\alpha_{i,n+1}, \ i = 1, ..., m$

can be obtained. Note that since $\alpha_{j,n} \neq \alpha_{j,n+1},\ j = 1, ..., m$, instead of (4.14), we will have,

$$H_n(i) = \left(\frac{X^{(i)}_{(n)} M_c - Y^{(i)}(M_{c(n)} + M_{e(n)})}{M_c - Y^{(i)}} \right) \tag{4.41}$$

where, $X^{(i)}_{(n)}, M_{c(n)}$ and $M_{e(n)}$ is as defined in the case of identical release times case, using $\alpha_{i,n},\ i = 1, ..., m$ from the n-th installment. Similarly, $Y^{(i)}$ and M_c is as defined, using $\alpha_{i,n+1},\ i = 1, ..., m$ from $(n+1)$-th installment. Since that the H in between the n and $(n+1)$-th installments is a special case, we denote it as,

$$H_{(n)} = max\{H_n(i)\}, \forall i = 2, ..., m-1 \tag{4.42}$$

Similar to the case of identical release times, the values of $H_{(n)}(i)$ are essentially constants. Although $H_{(n)}(i)$ is an expression comprising $M_c,\ M_{c(n)},\ M_{e(n)},\ X^{(i)}_{(n)}$ and $Y^{(i)}$ (which are functions of $\alpha_{i,n}$ and $\alpha_{i,n+1}$), these are the $\alpha_{i,n}$ and $\alpha_{i,n+1}$ that are derived using (4.40) and are fixed. From $(n+1)$-th installment onwards, H will be given by (4.15) and it is constant as explained earlier. Similarly, $\alpha_{i,n+1},\ i = 1, ..., m$ is given by (4.40) and will be constant from $(n+1)$-th installment onwards. We now generalize the load L_{n+i} for all installment after the n-th installment as,

$$L_{n+i} = \frac{L_n(M_{c(n)} + M_{e(n)} - H_{(n)})}{(M_c + M_e - H)} \left(\frac{(M_c + M_e - H)}{M_c} \right)^i \tag{4.43}$$

which is similar to (4.19) in which only the τ has been replaced with $L_n(M_{c(n)} + M_{e(n)} - H_{(n)})$. Following similar steps, if $(n+K)$ installments are needed to finish processing the load, then to get a feasible K, we have the following condition to be satisfied.

$$L_n(M_{c(n)} + M_{e(n)} - H_{(n)}) > (L - \sum_{p=1}^{n} L_p)(H - M_e) \tag{4.44}$$

Using the condition (4.44), we can determine a feasible value for the number of installments to process the load, starting from n-th installment onwards. Thus, upon the existence of a feasible value of K, we will proceed to carry out the next installment as usual. Note that, even if condition (4.44) has been satisfied for the n-th installment, in the subsequent installments, a feasible value of K may not exists, as the number of processors participating will tend to increase. On the other hand, if condition (4.44) is violated at the n-th installment, then it guarantees that there will be no feasible value of K exists to finish processing the load L. For cases like this, we have no choice except to use heuristic algorithms.

Thus, starting from second installment onwards, (4.44) will be verified using the same number of processors used in the previous installment. Under this condition, if a K value is feasible, then we normally continue with the next installment with the proposed *Multi-installment strategy*. However, if K value is not feasible, then starting from the current installment, we follow any heuristic strategy, to be described later, to complete the processing of the entire load.

4.2.4 Special Cases

As a final step, we shall now present three special cases of interest that are to be taken care to complete our analysis.

Case 1: If $m = 2$. If all the analysis discussed above were to be used for this case, (4.11) (for identical release times) cannot be used. Note that one may end up with this case even for arbitrary release times when the number of processors that are participating is just two (P_1 and P_2) from any installment. However, in the latter case, if the two processors are different from P_1 and P_2, then (4.38) can be used as before, since (4.38) was also derived from (4.11). In any of these cases, the following equation will be used instead of (4.11).

$$t_n \geq t_{n-1} + L_{n-1}(1 - \alpha_{1,n-1})C_1 \tag{4.45}$$

Using (4.45), (4.38) can be simplified as,

$$L_n \alpha_{l,n} = \frac{L_{n-1}M_e - (\frac{\tau_l - \tau_2}{E_2})C_1}{\frac{E_l}{E_2}C_1} \tag{4.46}$$

Case 2: If $\tau_i = 0$, for some $i \in \{1, ..., m\}$. In this case, initially for the set of processors that have $\tau_i = 0$, say r (as a first installment), we need to solve a set of r equations involving r unknowns ($\alpha_{i,1}$) following the timing diagram. Complete derivation is presented in the Appendix.

At the completion of this first installment, all the above r processors will stop at τ_s, where τ_s is the earliest release time among the set of processors that have non-zero release times. Thus, from the second installment onwards, if there is any load left unprocessed, analysis presented for multi-installment strategy for arbitrary release times can be used directly, with $M_e = \alpha_{x,1}E_x$ for second installment and $\alpha_{x,1}$ is obtained from the procedure presented in appendix.

Case 3: If $\tau_i = 0, \forall i = 1, ..., m$. We refer to [39, 33] for this case. We omit details.

4.3 Heuristic strategies

In this section, we shall present two heuristic strategies that can be used to distribute the load when conditions for continuous processing of the load cannot be satisfied such as (4.23) and (4.44). We shall design a heuristic, referred to as heuristic A, when (4.23) is violated (for identical release times) and another heuristic, referred heuristic B, when (4.44) is violated for non-identical release times case.

Heuristic A: This case deals with identical release times. Thus, whenever (4.23) is violated, we shall employ this strategy. In this heuristic, the load portion will be transmitted to each processor in a single installment. We first partition the processors into two groups as those which receive their data before time τ and those receive their data after time τ. We shall call the former group as set S, and the processors in this set satisfy the following condition.

$$L\sum_{p=1}^{i-1}(1-\sum_{j=1}^{p}\alpha_{j,1})C_p \leq \tau,\ i=1,...,m \tag{4.47}$$

First, we assume that initially $S = \{P_1\}$. Hence, we will have the following relationship between $L\alpha_{i,1}$ and $L\alpha_{i+1,1}$.

$$L\alpha_{i,1}E_i = \sum_{j=i+1}^{m} L\alpha_{j,1}C_i + L\alpha_{i+1,1}E_{i+1},\ i=m-1,m-2,...,2 \tag{4.48}$$

Using (4.48), we can relate $L\alpha_{i,1}$, for $i = 2,...,m-1$ with respect to $L\alpha_{m,1}$ and using the fact that $\sum_{i=1}^{m}\alpha_{i,1} = 1$, we obtain,

$$L\alpha_{1,1} + L\sum_{p=2}^{m}\alpha_{p,1} = L \tag{4.49}$$

Expressing each of the $L\alpha_{i,1},\ i = 2,...,m$ in (4.49) with (4.48), we determine $L\alpha_{1,1}$ as a function of $L\alpha_{m,1}$. Hence, we obtained all the $L\alpha_{i,1}, i = 1,...,m$ with respect to $L\alpha_{m,1}$. Next, we define a condition wherein the processing time for P_1, starting from τ ($\tau + L\alpha_{1,1}E_1$), is the equal to the total communication time till P_m plus the processing time of P_m. That is,

$$\tau + L\alpha_{1,1}E_1 = L\sum_{p=1}^{m-1}(1-\sum_{j=1}^{p}\alpha_{p,1})C_p + L\alpha_{m,1}E_m \tag{4.50}$$

Solving (4.50) using all the $L\alpha_{i,1}, i = 1,...,m$ found previously, we can then calculate $L\alpha_{m,1}$. With $L\alpha_{m,1}$ known, all other $L\alpha_{i,1}, i = 1,...,m-1$ can be immediately calculated. Now, we use (4.47) to identify a set of processors that can be included in the set S together with P_1. After identifying the set S, we use the following set of recursive equations to determine the exact load portions to be assigned to the processors. Note that all the processors in the set S start computing at τ.

$$L\alpha_{i,1} = L\alpha_{1,1}\frac{E_1}{E_i} \tag{4.51}$$

Then, solving (4.51) for the processors in S, (4.48) for other processors and together with (4.49) and (4.50), we get another set of $L\alpha_{i,1},\ i = 1,...,m$. These are the load fractions that are to be assigned to the processors in the system. Note that although (4.51) assumes that the processors in S start at τ, the exact communication delays are not accounted. Therefore,

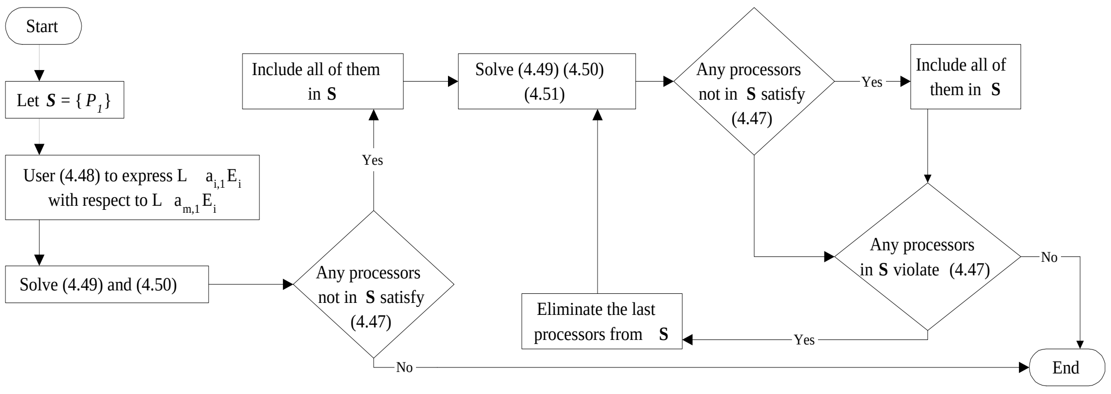

Start
Let S = { P1 }
User (4.48) to express L ai,1Ei with respect to L am,1Ei
Solve (4.49) and (4.50)
Any processors not in S satisfy (4.47)
Yes
No
Include all of them in S
Solve (4.49) (4.50) (4.51)
Any processors not in S satisfy (4.47)
Yes
Include all of them in S
Any processors in S violate (4.47)
No
Yes
Eliminate the last processors from S
End

Figure 4.4: Flow chart illustrating Heuristic A.

as a last step, we need to verify whether or not all the processors in S indeed start at τ using (4.47). Thus, in case, if some processors in S violate (4.47), we will then eliminate the processor with largest index and repeat the above procedure until all processors in S satisfy (4.47). On the other hand, if all the processors in S satisfy (4.47) it is guaranteed that all the processors in S can indeed start from τ. Further, at this stage, as an incentive, it may happen that few processors that do not belong to S may satisfy (4.47) and we include all these processors in the set S. The flow chart presented in Fig. 4.4 shows the complete description and the following example illustrates the workings of heuristic A.

Example 4.3 (Heuristic A): Consider a linear network system with the parameters C_1=1, C_2=1, C_3=4, C_4=2, C_5=5, E_1=5, E_2=5, E_3=15, E_4=15, E_5=10, E_6=5 with identical release time τ=10, and the system is given a load L=10. Initially we assume $S = \{P_1\}$. Therefore, for $P_i,\ i = 2, ..., 6$, we use (4.48) to relate $L\alpha_{i,1},\ i = 2, ..., 6$ with respect to $L\alpha_{6,1}$. Expressing each $L\alpha_{i,1},\ i = 2, ..., 6$ with respect to $L\alpha_{6,1}$ in (4.49), we can then relate $L\alpha_{1,1}$ as a function of $L\alpha_{6,1}$. Now with all $L\alpha_{i,1},\ i = 1, ..., 6$ in respect to $L\alpha_{6,1}$, we can immediately obtain the value of $L\alpha_{6,1}$ by solving (4.50). Other values, $L\alpha_{i,1},\ i = 1, ..., 5$ can then be calculated using $L\alpha_{6,1}$.

Now, using all the $L\alpha_{i,1},\ i = 1, ..., 6$, we observe that P_2 and P_3 satisfy (4.47). Hence, we include P_2 and P_3 in S and then use (4.51) to express $L\alpha_{2,1}$ and $L\alpha_{3,1}$ in terms of $L\alpha_{1,1}$. Then, using the expression obtained for $L\alpha_{2,1}$, $L\alpha_{3,1}$ and $L\alpha_{i,1}$, $i = 4, ..., 6$ into (4.49) to determine $L\alpha_{1,1}$ as a function of $L\alpha_{6,1}$. Then, we solve (4.50) to obtain $L\alpha_{6,1} = 0.639$. Next, we determine the remaining $L\alpha_{i,1}, i = 1, ..., 5$ values. With all these values obtained, we verify that both P_2 and P_3 satisfy (4.47) and no other processors except P_1, P_2 and P_3 satisfy (4.47), and hence the result. The distribution is given by $\alpha_1 = 0.3394,\ \alpha_2 = 0.3394,\ \alpha_3 = 0.1131,\ \alpha_4 = 0.0662,\ \alpha_5 = 0.0710,\ \alpha_6 = 0.0710$. The processors $P_2,\ P_3,\ P_4,\ P_5$, and P_6 will receive their respective loads at times 6.606, 9.818, 18.142, 20.98, and 24.525. Hence, $P_1,\ P_2$, and P_3 can start processing their load portions starting from their release times while others remains idle until their load portions arrive. The processing time is given by, $T_{process} = T(6,1) = 26.97$.

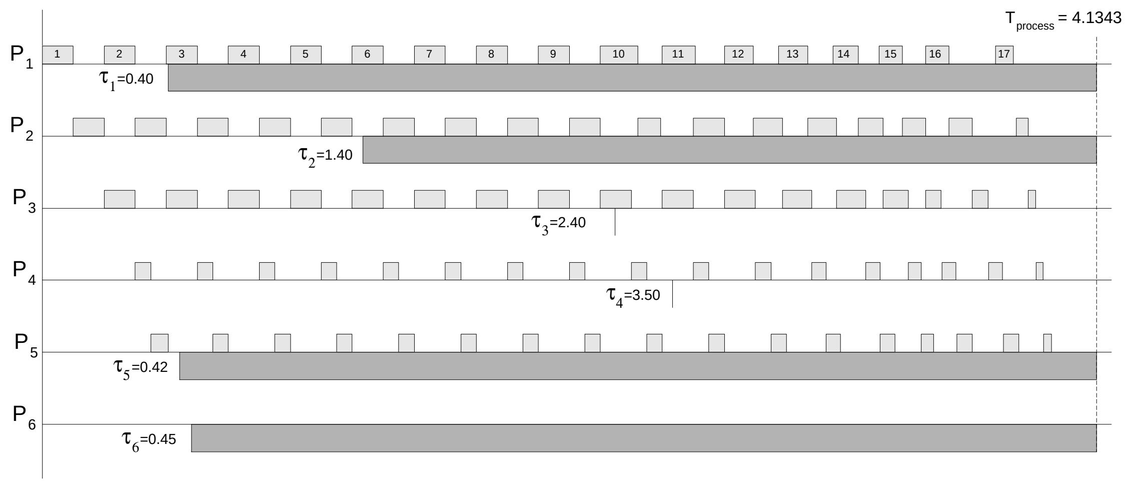
□

Heuristic B: This heuristic will be used when (4.44) is violated for the case of non-identical release times. Let (4.44) be satisfied in n-th installment. If, for example, (4.44) is violated at the $(n+1)$-th installment, then we consider n-th installment. Since condition (4.44) is satisfied at the n-th installment, this means that the remaining load $(L - \sum_{j=1}^{n} L_j)$ can be processed using a set of processors qualified in the n-th installment. This heuristic basically exploits this property to guarantee that the remaining load can be completely processed. If at the n-th installment, r processors were used, then we will continue using r processors for all the successive installments. We denote this r processors as set R. First, we will determine $L_j,\ j = n+1, n+2, ..., n+K$ (assuming that we need $(n+K)$ installments to finish the remaining load) with (4.43) and then determine $\alpha_{i,j},\ i \in R,\ j = n+1, n+2, ..., n+K$ with (4.40) for all the successive installments, respectively. Next, t_{n+1} the starting time for the $(n+1)$-th installment, is given as,

$$t_{n+1} = t_n + L_n H_{(n)} \tag{4.52}$$

where, $H_{(n)}$ is as defined in (4.42). Further, the following installments, the starting time t_j, is given by,

$$t_j = t_{j-1} + L_{j-1} H,\ j = n+2, ..., n+K \tag{4.53}$$



Iteration	1	2	3	4	5	6	7	8
r: Number of qualified processors	3	3	3	4	4	4	4	5
K: number of installments required to complete processing the load using r processors	12	11	10	13	12	11	10	$\emptyset$

System parameters: C_1=3, C_2=2, C_3=3, C_4=2, C_5=3, E_1=24, E_2=12, E_3=18, E_4=12, E_5=12, E_6=12, τ_1=0.40, τ_2=1.40, τ_3=2.40, τ_4=3.50, τ_5=0.42, τ_6=0.45, and L=1.

Figure 4.5: Example for Heuristic B: Arbitrary release times. The numbers appearing in communication blocks of P_1 denote the installment number.

where, H is as defined in (4.15). Note that, for this heuristic, the value $(n+K)$ (number of installment needed) can be determined with the following equation, which is similar to (4.22), however, now, it is derived from (4.43) instead of (4.19).

$$n + K = \left(\frac{ln(\frac{Q+(L-\sum_{p=1}^{n} L_p)(B-M_c)}{Q})}{ln(\frac{B}{M_c})} \right) \tag{4.54}$$

where, $Q = L_n(M_{c(n)} + M_{e(n)} - H_{(n)})$ and $B = M_c + M_e - H$.

This heuristic is simple to implement however, since not all of the available processors are used, this heuristic will have a low system (processors) utilization, which is an obvious disadvantage. However, when the demands of the application requiring the processed load are not time critical, this strategy can be readily used. The timing diagram shown in Fig. 4.5 demonstrates the workings of this heuristic. The table in Fig. 4.5 shows the number of processors that are

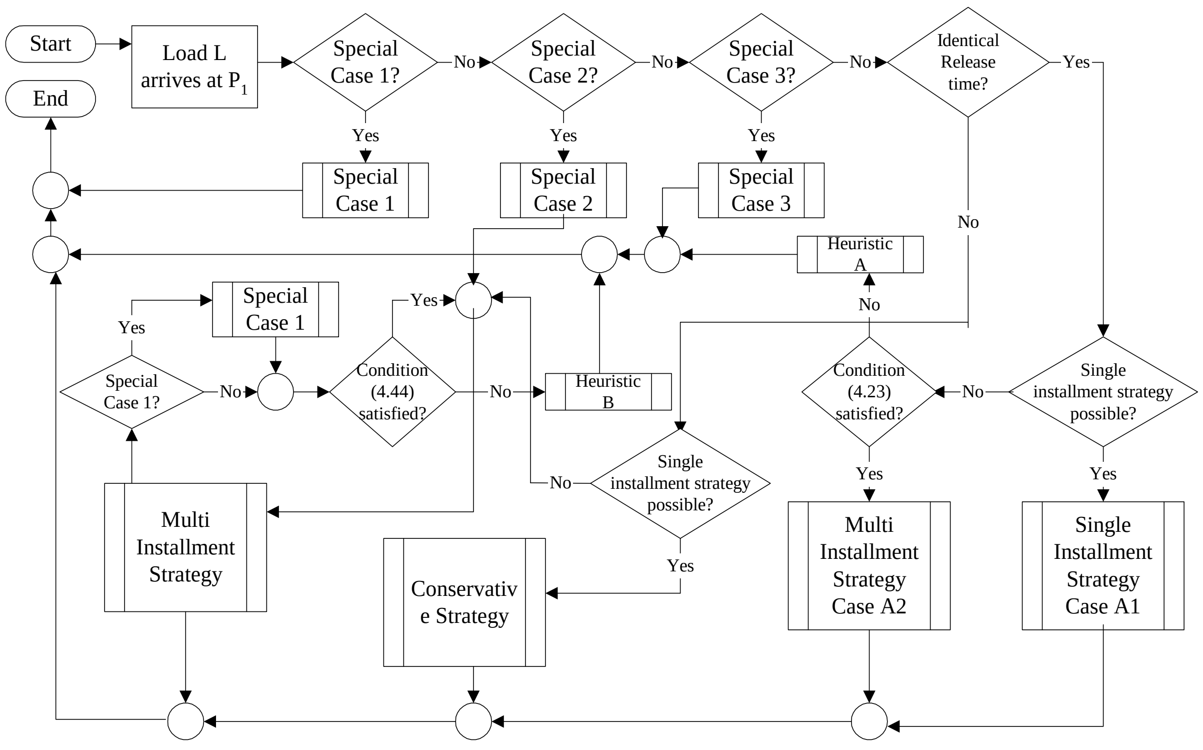

Start
End
Load L arrives at P1
Special Case 1?
Special Case 2?
Special Case 3?
Identical Release time?
No
Yes
Special Case 1
Special Case 2
Special Case 3
Heuristic A
Special Case 1
Special Case 1?
Condition (4.44) satisfied?
Heuristic B
Condition (4.23) satisfied?
Single installment strategy possible?
Single installment strategy possible?
Multi Installment Strategy
Conservative Strategy
Multi Installment Strategy Case A2
Single Installment Strategy Case A1

Figure 4.6: Flow chart showing the entire scheduling of a divisible load by a scheduler at P_1.

qualified in every iteration and the number of installments required to complete the processing of the entire load using these set of qualified processors. Note that so long as K value exists at each iteration, need for heuristic strategies does not arise. In this example, it may be noted that at the 8-th iteration, K value ceases to exist and hence, we attempt to utilize the 4 processors that participated in the 7-th iteration. Thus, in this case, the processing time is given by, 4.1343 units. We will discuss on other details in the next section.

4.4 Discussions of the Results

In this section, we shall discuss our contributions in this chapter. As mentioned, designing divisible load distribution strategies for linear networks when processors having non-zero release times is never attempted in the literature so far. Although the problem of release times were addressed for bus networks [19], designing load distribution strategies for linear networks is a challenging problem as the data distribution has a pipelined communication pattern involving $(m-1)$ links, as opposed to a bus network which has a single communication link. The load is assumed to arrive at P_1 (boundary processor) where our scheduler resides. The function of the scheduler in generating a load distribution is completely described in Fig. 4.6. Further, in the case of linear networks, adopting multi-installment strategy for load distribution is very complex as there is a scope for "collision" among the adjacent front-end operations, if communication phase is not scheduled carefully. Clearly, a load distribution strategy must be such that it must recommend a load distribution that is free of any possible collisions among the front-end operations. Thus, it becomes imperative to check this collision-free requirement in the design of any load distribution strategy for linear networks. Note that this collision-free requirement arises only in the case of multi-installment strategy.

In this chapter, we systematically consider different possible cases that can arise in the design of a load distribution strategy. If the processors are idle at the time of arrival of the load $(\tau_i = 0, \forall i = 1, ..., m)$, the idle case algorithm presented in the literature [39, 33] can be immediately used. On the other hand, if the processors are engaged in some computational work when the load arrives, we follow the load distribution strategies described in this chapter. We consider the boundary case for analysis and hence, our scheduler which carries out the load distribution process as per the strategies designed in this chapter, resides on P_1. As mentioned in Section 4.1, the scheduler on P_1 will first determine the release times, either through explicit communication with the processors in the system or it can estimate the release times. Of course, each processor in the system can also estimate their respective release times and convey the estimated information to P_1. We assume that P_1 knows the respective release times.

As done in the literature, we consider two possible cases of interest, namely identical release times and non-identical release times. When the release times are identical and satisfy the condition for *Case A1*, then an optimum load distribution is guaranteed by using (4.4), i.e., we simply use single-installment strategy. However, if condition for *Case A1* is violated, then (4.23) will be used to check for an optimum solution. (4.23) opens up the possibility for using multi-installment strategy to achieve an optimum solution. Thus, when (4.23) is satisfied, multi-installment strategy (discussed for *Case A2*) will give an optimum solution. On the other hand, when (4.23) fails to hold, we attempt to use heuristic strategies, presented in Section 4.3. The choice of heuristic strategies depends on several issues. As mentioned in Section 4.3, Heuristic A is customized to handle identical release times case and the load will be distributed in a single installment. The strategy guarantees (by solving the equations in Section 4.3) that a maximal set of processors will start computing at time τ while the rest will start after τ. This ensures that only a minimum amount of processor time is wasted during this processor idling phase for those processors that start after τ, thus yielding a good solution. Thus, if the number of such processors is small compared to the total number of processors that start at τ, the solution proposed by this heuristic can very well be accepted. In Example 4.3, we observe that processors $1, 2$ and 3 start at $\tau = 10$ whereas the rest of the processors start after the release time. It may be observed that for larger τ values for which (4.23) continues to violate, the number of processors that will start at τ will be increasing and hence, the processing time decreases. Finally, it may be noted that one can attempt to use multi-installment to further reduce the processing time, however, it may be a very complex procedure.

When the processor release times are arbitrary, (4.26) can be used to determine whether or not all the m processors can be used to participate in computing the entire load as described in the *conservative strategy*, i.e., the strategy in which we first attempt to distribute the entire load in a single installment using m processors. In case (4.26) fails with m processors, we recursively use (4.26) to determine the maximal set of qualified processors that are able to participate in processing the load. Thereafter, if all the qualified processors can satisfy (4.27), then an optimum distribution is given by (4.24). At this stage, an important observation to make is the result of Lemma 4.1. Lemma 4.1 clearly testifies the fact that it is sufficient to consider a maximal set of processors governed by (4.26) for processing the entire load instead of waiting for processors who release times are farthest. In fact, Lemma 4.1 shows that the entire processing time will be completed on or before the first earliest release time of a processor that was not qualified to participate in the computation. Hence, with the choice of using single installment policy, the proposed solution is indeed optimal. The solution approach becomes very complicated when (4.27) is violated, and we are forced to use multi-installment strategy to improve the performance. Similar to the case of identical release times, (4.44) is used to check if multi-installment strategy can yield an optimal solution. If (4.44) is satisfied for all the K installments, then the multi-installment strategy presented in this chapter will give an optimum distribution. However, if (4.44) is violated, we attempt to use heuristic strategies as before.

The workings of our heuristic strategy B is shown via an example in Fig. 4.5, wherein it may be noted that before the commencement of 8-th iteration, we check for a feasible K value. We observe that K value ceases to exist and hence, we attempt to utilize only the 4 processors that participated in the 7-th iteration. It may be noted that for the first few iterations, until iteration 3, the number of processors that qualify remains as 3 and the number of installments required to complete the processing the load with 3 processors decreases steadily. However, during 4-th iteration, we observe that P_2 participates in the computation and recalculating K, we observe that the number of installments required to complete processing now increases. Thus, whenever the same set of processors participate K decreases steadily. However, it may be noted that increase in K value does not imply that the processing time increases.

4.5 Concluding Remarks

The problem of scheduling a divisible load on a linear network of processors, by taking into account their release times, is addressed in this chapter. This chapter presents a number of useful results on the problem attacked. Firstly, conditions for a collision-free front-end operation for the case of multi-installment strategy are explicitly derived. Use of single, as well as multi-installment policies are adopted in the design of load distribution strategies. A systematic approach is followed in the design of strategies. Firstly, the case when all the processors have identical release times is considered. We have attempted to use single installment strategy and if it fails to hold, we attempt to obtain an optimal solution using multi-installment. Here, when condition (4.23) is satisfied, we derived the total number of installments to be used to distribute the total load. In case the above condition fails, we attempt to use heuristic strategy A. Secondly, we considered non-identical release times case and derived certain conditions to obtain an optimal solution. When conditions derived in Section 4.2 for single and multi-installment strategies cannot be satisfied, possible, we attempt to use heuristic strategy B. It may be noted that there may be a number of heuristic strategies possible, however, the choice of our strategies are based on certain features such as, attempting to employ all the processors in the system before the release time τ, in the case of heuristic strategy A and attempting to use a maximal set of processors in computation, in the case of heuristic strategy B. In multi-installment strategy for the case of non-identical release times, we attempt to use as many processors as possible to minimize the processing time in every installment, however, when continuing the computation of the load without any processor idling is no longer possible in the "near future" (with the estimated value of K), we simply utilize the number of processors used in the previous installment to complete processing the load (heuristic strategy B). While this strategy may underutilize the processors available, coupled with the result of Lemma 4.1, it guarantees an acceptable solution.

Chapter 5

Aligning Biological Sequences: A Divisible Load Scheduling Approach

Comparative analysis is often used in biological researches. For example, determining the similarity between a newly discovered gene sequence and a known gene (from a database), may gives significant understanding on the function, structure, as well as the origin of the new gene. Biological sequences are made up of residues. In DNA (DeoxyriboNucleic Acid) sequences, these residues are nucleic acids, while in protein sequences, these residues are amino acid. In comparing two sequences, commonly known as aligning two sequences, residues from one sequence are compared with the residues of the other while taking into account of the position of the residues. Residues can be inserted, deleted or substituted from either two sequences to achieve maximum similarity, or optimum alignment, between the two sequences [41]. There are as much as $(1 + \sqrt{2}\ )^{2x+1}\sqrt{x}$ combinations [42] for these insertion, deletion, and substitution operations, where x is the length of the sequences. Hence, aligning sequences is a time consuming procedure especially in the case of aligning multiple sequences.

In 1970, Needleman and Wunsch [43] introduced an algorithm for comparing two biological sequences for similarities without the need of going through all the combinations of insertion, deletion or substitution operations. This algorithm is then improved by Sellers [44] and later generalized by Smith and Waterman [45, 46]. These algorithms are still popular today in aligning DNA (Needleman-Wunsch) and protein (Smith-Waterman) sequences.

Although the Needleman-Wunsch algorithm does not go through all the possible combinations, it still has a complexity of $O(x^2)$. The Smith-Waterman algorithm, on the other hand, has a complexity of $O(x^3)$ but was later improved by Gotoh [47] to just $O(x^2)$. Nevertheless, as longer sequences are generated with ever advancing sequencing technology, a complexity of $O(x^2)$ may still be unacceptable in many cases. This is especially true if considering the vast amount of sequences available today available from databases such as [48, 49, 50]. Further, these gigantic databases are growing rapidly, i.e., the GenBank is growing at an exponential rate, with rate as much as of 1.2 million new sequences a year [51]. As a result, a wide variety of heuristic methods have been proposed for aligning sequences, such as FASTP [52], FASTA [53, 54], BLAST [55], and FLASH [56]. These heuristics obtain computation speed-up at the cost of less sensitivity. Other methods, such as [57], are able achieve speed-ups without losing sensitivity. Nevertheless, the speed-ups achievable are heavily dependant on the similarity of the sequences. Further this method does not generate the complete Smith-Waterman matrix that may be useful to detect multiple subsequence similarities.

Meanwhile, other researchers on the other hand, attempt to gain speed-up by exploiting the advantages of parallel processing systems. The native way of parallelling the Needleman-Wunsch (or Smith-Waterman) algorithm is to compute the matrix elements in a diagonal fashion [58]. The level of parallelism that can be achieved in this manner is limited by the heavy communication cost due to the data dependency. As a result, this method is only practical when implemented in expensive, customized MIMD (Multiple-Instruction Multiple-Data) systems. Other more cost effective approaches that utilized general-purpose processors are presented in [59, 60, 61]. In [59], Yap *et al.,* presented a speculative computation approach where he exploit the independency characteristic of the Beger-Munson [62] algorithm for multi-sequence alignment. As illustrated in his paper, the speculative computation approach can reduce a 28 computation steps process to just 13 steps, hence achieving speed-up. Nevertheless, speed-up is dependant to the similarities of the group of sequences being compared and homogenous processors are required in order achieve high level of parallelism. Further, due to the working of this speculative approach and only small amount of processors can be utilized as using more processors may not be efficient.

In [60], Trelles *et al.,* presented a new clustering strategy for multi-sequence alignment that is able to achieve speed-up by significantly reduce the number computational steps (pairwise alignments). Further, this clustering strategy also able to incur independent processes that are able to be processed in parallel, hence achieving more speed-up. The disadvantage of this parallel processing strategy is that the number of idle processors are considerably more in the early stage of the alignment process and utilizing non-homogenous processors may heavily cripple the overall speedup.

In a recent paper [61], Torbjorn *et al.,* presented a method that is able to utilize a generic Intel processor with MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) technology to achieve speed-up by parallelism. The major draw back of this method is that the amount of speed-up that can be achieve is restricted by the processor's technology, i.e. a Pentium III processor can only allow a 8-way parallelism.

In this chapter, we present a parallel implementation of Smith-Waterman algorithm that utilized Divisible Load Theory (DLT) to achieve high level of parallelism by pipelining the computational process. Our strategy can be easily modified to be used with the Needleman-Wunsch algorithm or other similar algorithm. One of the advantages of our approach is that we are able to utilize non-homogenous (heterogenous) processors as we will determine the amount of load to be assigned to each processors such that the level of parallelism will not be deteriorated, i.e., slower processors will receive less amount of load. Further, since our strategy achieve speed-up at the Smith-Waterman (or Needleman-Wunsch) matrix generation level, it can be easily implemented/integrated in either 3 of the strategies mentioned above [59, 60, 61] and more, i.e., the Divide and Conquer method [63] that is able to reduce the amount of memory required to align long sequences.

The organization of the chapter is as follow, in Section 5.1, we will briefly introduce some preliminaries knowledge and formally define the problem we address here. We will then present our strategy for the problem concerned in Section 5.2. In this section, we also be deriving certain conditions that needs to be satisfied in order to guarantee an optimal solution. In cases where these conditions cannot be satisfied, we will then resolve to heuristic strategies. We will present three heuristic strategies in Section 5.3 for such cases. In Section 5.4, we will discuss the performance of our strategy using rigorous simulation study. Finally, we will conclude this chapter in Section 5.5.

5.1 Preliminaries and Problem Formulation

5.1.1 Smith-Waterman Algorithm

We first briefly introduce an improved version of Smith-Waterman algorithm by Gotoh [47] as well as some characteristics of the matrix generated by the algorithm. In aligning two sequences, denoted as SqA and SqB, of length α and β respectively, the algorithm basically generates 3 separate matrices, denoted as matrix $\mathbf{S}$, $\mathbf{h}$ and $\mathbf{f}$. Each row and column of these matrices represent a residue of SqA and SqB, respectively. Given $s(a_x, b_y)$ as the substitution score[1] by replacing the x-th residue from SqA with the y-th residue from SqB, z as the penalty for introducing a gap, and v as the penalty for extending a gap. The **S**, **h** and **f** matrices are related by the following recursive equations.

$$S_{0,y} = S_{x,0} = h_{0,y} = f_{x,0} = 0 \tag{5.1}$$

$$S_{x,y} = \max \left\{ \begin{array}{l} h_{x,y} \\ S_{x-1,y-1} + s(a_x, b_y) \\ f_{x,y} \end{array} \right\} \tag{5.2}$$

$$h_{x,y} = \max \left\{ \begin{array}{l} S_{x-1,y} + z \\ h_{x-1,y} + v \end{array} \right\} \tag{5.3}$$

$$f_{x,y} = \max \left\{ \begin{array}{l} S_{x,y-1} + z \\ f_{x,y-1} + v \end{array} \right\} \tag{5.4}$$

for the range $1 \leq x \leq \alpha,\ 1 \leq y \leq \beta$ where $S_{x,y}, h_{x,y}$, and $f_{x,y}$ represent the matrix elements in the x-th row, y-th column of the matrices **S**, **h** and **f** respectively. In this computation process, residues in SqA and SqB are tested for a best possible alignment in a recursive fashion. The computation of the above mentioned matrices leads to possible alignment of the respective sequences. The score of the matrix element, $S_{x,y}$, quantifies on the quality of alignment (from the 1-st residue of SqA and SqB respectively) until x-th residue of SqA and y-th residue of SqB. Thus, higher the score at $S_{x,y}$, better the alignment between the sequences until those residues. The details on the workings of this algorithm can be found in [58].

The **S** matrix contains all these alignment score of SqA and SqB and it is used to determine the optimal alignment. As we can see from the equations above, the matrix element $S_{x,y}$ is dependant to the $(x-1, y-1)$, $(x-1, y)$, and $(x, y-1)$ elements from the **S**, **h** and **f** matrices, respectively. This is illustrated in Fig. 5.1. Due to this dependency, the **S** matrix

[1]Score defines the similarities between two residues. The scores can be found in substitution score matrices such as the identity score matrix for DNA or the PAM250 matrix for protein [64]. These substitution scores matrices are predefined based on biological factors.

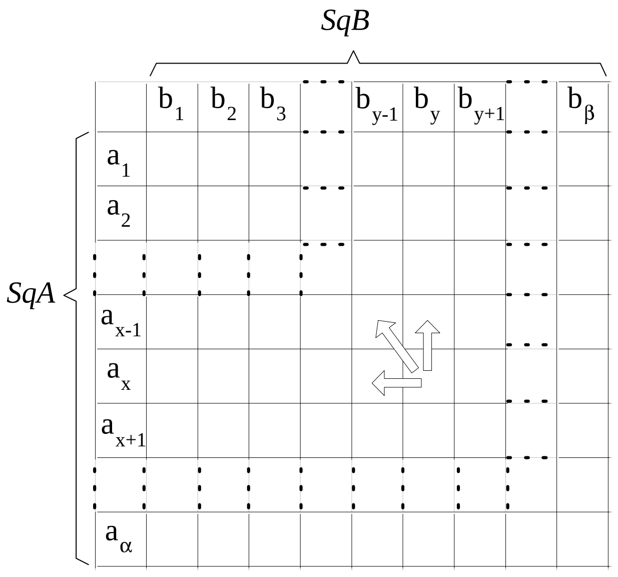

SqB
b_1 b_2 b_3 b_{y-1} b_y b_{y+1} b_β
SqA
a_1 a_2 a_{x-1} a_x a_{x+1} a_α

Figure 5.1: Illustration of the computational dependency of the element (x,y) in the **S** matrix

elements cannot be computed independently, either column-wise or rows-wise. Nevertheless, the elements along the diagonal line given by, $S_{x+1,y-1}$, $S_{x,y}$, and $S_{x-1,y+1}$, are independent to each other and hence, they can be calculated independently. In our strategy we exploit this property and attempt to distribute the computations of the matrix elements across several processors in our cluster system.

5.1.2 Trace-back process

The trace-back process is used to determine the optimal alignment between two sequences. The process utilize the characteristic of the **S** matrix where every matrix elements represents the maximum alignment score of the sequences until the respective residues. For example, when aligning SqA and SqB, the $S_{x,y}$ in the **S** matrix represent the maximum score of aligning $a_1, ..., a_x$ and $b_1, ..., b_y$ where a_x and b_y are the x-th and y-th are the residues of SqA and SqB respectively. Hence, to obtain the optimum alignment, we start from the bottom right of the **S** matrix and move towards the upper left of the matrix. At each matrix element, we are only allow to move to the adjacent element in three different directions, they are, up, left

	T	G	C	G	G	A	A	T
T	**10**	10	10	10	10	10	10	15
G	10	**20**	20	25	30	30	30	30
C	10	20	**30**	30	30	35	35	35
A	10	20	**30**	35	35	40	45	45
A	10	20	**30**	35	40	45	50	50
C	10	20	**35**	35	40	45	50	55
G	10	25	35	**45**	**50**	50	50	55
A	10	25	35	45	50	**60**	**65**	**65**

Table 5.1: Example 5.1: Trace-back process

or diagonally (upper-left). The choices are determined by the largest values among them. Moving up and left represent the introduction of a 'gap' in *SqA* and *SqB* respectively while moving diagonally represent a 'substitution' where no gap is added. Hence, moving diagonally is preferred if there is a tie in between score for the direction up/left and diagonal.

Example 5.1 : Consider the arbitrary **S** matrix given in Table 5.1 where *SqA* and *SqB* have the residues TGCAACGA and TGCGGAAT respectively.

According to the result from the trace-back process the optimal alignment is

```
T G C A A C G _ A _ _
T G C _ _ _ G G A A T
```

■

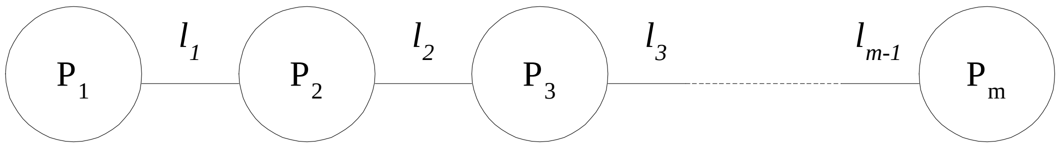


Figure 5.2: Linear network with m processors interconnected by $(m-1)$ communication links

5.1.3 Problem Formulation

In this section, we will formally defined the linear network architecture as well as the problem we address. We consider a loosely coupled multiprocessor system, interconnected in a linear daisy chain fashion, as shown in Fig. 5.2, with m processors denoted as $P_1, P_2, P_3, ..., P_m$, and $(m-1)$ links, denote as $l_1, l_2, l_3, ..., l_{m-1}$. All the processors in the system are assumed to have front-end. A front-end is a coprocessor that off loads the communication task from the processor such that the processor can compute and communicate at the same time instant. Nevertheless, the front-end cannot send and receive data simultaneously.

As stated, we consider the problem of aligning two sequences. Our objective is to design a strategy such that the *processing time*, defined as the time when the computation starts until it ends, is minimum. The computation process involves generating the **S**, **h**, and **f** matrices, as presented in Section 5.1.1. In this work, we do not consider the computation time of post-processes, i.e., the trace-back process that is required to determine the optimal alignment between the two sequences.

We denote the two sequences being aligned as *SqA* and *SqB*. We assume that all processors, $P_1, ..., P_m$, already have *SqA* and *SqB* in their local memory. This is a practical assumption as sequences can be broadcasted to all processors. Further, in the case of multi-sequence alignment, sequences are often compared with each other more than once where only slight differences are made each time the sequences are being compared, i.e., the Berger-Munson algorithm [62]. Hence, it would be feasible for all the processors to keep a copy of all sequences in their local memory and only modifications on the sequences are broadcasted. It should be clear at this stage that we are concerned with the design of the strategy after the processors already had the sequences to be aligned and we do not address the problem how these sequences are broadcasted to the processors.

We shall now introduce an index of definitions and notations that are used throughout this chapter.

m	The total number of processors in the system
P_i	The i-th processor, where $i = 1, .., m$
l_i	The communication link between P_i and P_{i+1}
E_i	The time taken for P_i to compute one matrix element in the **S** matrix, including the necessary two matrix elements in **h** and **f** matrices
C_i	Time taken for l_i to communicate one matrix element
α	Length of *SqA* or the number of residues in *SqA*
β	Length of *SqB* or the number of residues in *SqB*
Q	Number of iteration used to compute the **S**, **h**, and **f** matrices
α_j	Number of residues of *SqA* assigned to P_j, where $$\sum_{j=1}^{m} \alpha_j = \alpha$$
$\beta^{(k)}$	Number of residues of *SqB* assigned in the k-th iteration, where $$\sum_{k=1}^{Q} \beta^{(k)} = \beta$$
$L_{i,k}$	Sub-matrix of **S** that is assigned to P_i in the k-th iteration
$T(m)$	The *processing time*, defined as the time period when the process of generating the **S** matrix starts until the end, with m processors.

5.2 Design and Analysis of Parallel Processing Strategy

In this section, we will present our multiprocessor strategy. Firstly, let us consider the distribution of the task of generating the **S** matrix. It may be noted that when we mention generating the **S** matrix, we also take into the generation of other two required matrices **h**, and **f**, as generating $S_{x,y}$ demands computing the entries of $h_{x,y}$, and $f_{x,y}$ elements as well. The **S** matrix is partitioned into sub-matrices $L_{i,k}, i = 1, ..., m,\ k = 1, ..., Q$, where each sub-matrix consist of a portion of SqA and SqB. We assign the computation of the sub-matrices $L_{i,k},\ i = 1, ..., m,\ k = 1, ..., Q$ respectively, in Q iterations, to processor P_i. The distribution pattern is as illustrated in Fig. 5.3. Due to the characteristic of the **S** matrix (as discussed in Section 5.1.1), sub-matrices $L_{i,k}$ with the same value of $(i+k)$ can be calculated concurrently. Thus, for instance, sub-matrices $L_{1,3}$, $L_{2,2}$, and $L_{3,1}$ can be calculated in parallel as all three $L_{i,k}$ have the same value $(i + k) = 4$.

Note that computing the values of a sub-matrix $L_{i,k}$ means we compute the values of the sub-matrices of the matrices **S**, **h**, and **f**, respectively. At this stage, it may be noted that, due to data dependency, P_i needs the values of sub-matrix $L_{i-1,k}$ from P_{i-1} in order to start computing $L_{i,k}$. To be more precise, P_i requires only the data from the last row of $L_{i-1,k}$ (of the **S** and **h** matrices) to start computing $L_{i,k}$. Note that the size of the last row (number of columns) of each sub-matrices of **S** and **h**, with processor P_{i-1}, is $\beta^{(k)}$, respectively. Hence, on the whole we need to transmit $2\beta^{(k)}$ values (last rows of **S** and **h** matrices from $L_{i-1,k}$) to P_i, before initiating the computation of $L_{i,k}$. Lastly, it may be noted that P_i does not require the values of **f** from P_{i-1} as the computation $L_{i,k}$ in every iteration uses values of **f** generated within the same processor.

Now, the question that remains unanswered is the amount of residues from SqA, i.e., the value $\alpha_i, i = 1, ..., m$, that should be assigned for each processor P_i such that a high degree of parallelism can be achieved. Further, we need to determine the amount of residues of SqB,

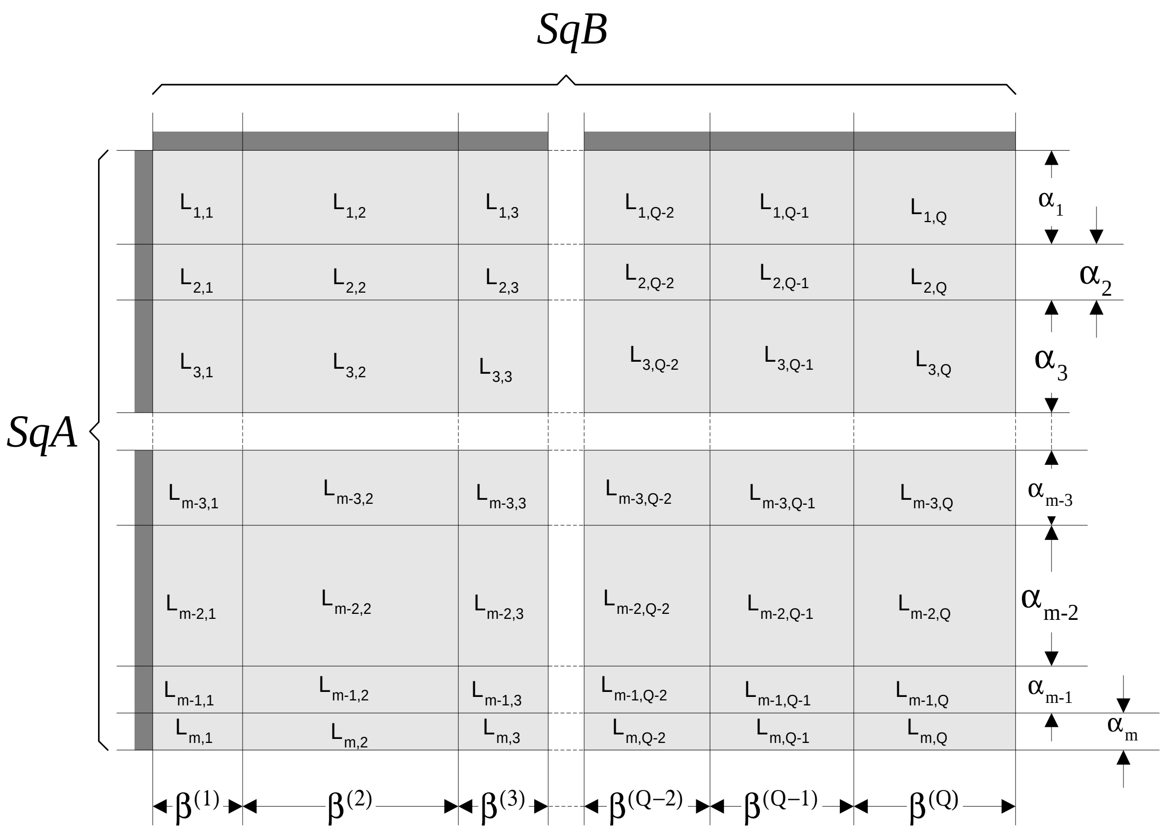

SqB
SqA
$L_{1,1}$ $L_{1,2}$ $L_{1,3}$ $L_{1,Q-2}$ $L_{1,Q-1}$ $L_{1,Q}$
$L_{2,1}$ $L_{2,2}$ $L_{2,3}$ $L_{2,Q-2}$ $L_{2,Q-1}$ $L_{2,Q}$
$L_{3,1}$ $L_{3,2}$ $L_{3,3}$ $L_{3,Q-2}$ $L_{3,Q-1}$ $L_{3,Q}$
$L_{m-3,1}$ $L_{m-3,2}$ $L_{m-3,3}$ $L_{m-3,Q-2}$ $L_{m-3,Q-1}$ $L_{m-3,Q}$
$L_{m-2,1}$ $L_{m-2,2}$ $L_{m-2,3}$ $L_{m-2,Q-2}$ $L_{m-2,Q-1}$ $L_{m-2,Q}$
$L_{m-1,1}$ $L_{m-1,2}$ $L_{m-1,3}$ $L_{m-1,Q-2}$ $L_{m-1,Q-1}$ $L_{m-1,Q}$
$L_{m,1}$ $L_{m,2}$ $L_{m,3}$ $L_{m,Q-2}$ $L_{m,Q-1}$ $L_{m,Q}$
α_1 α_2 α_3 α_{m-3} α_{m-2} α_{m-1} α_m
$\beta^{(1)}$ $\beta^{(2)}$ $\beta^{(3)}$ $\beta^{(Q-2)}$ $\beta^{(Q-1)}$ $\beta^{(Q)}$

Figure 5.3: Distribution pattern for matrices **S**, **h**, and **f**

the value $\beta^{(k)},\ k = 1, ..., Q$, that should be considered in each iteration for matching with each of the assigned residues of SqA. We shall discuss these in the next section.

5.2.1 Load distribution strategy

In the design of our strategy, we utilize DLT to determine the amount of residues of SqA that should be given to each processor, as well as the amount of residues that should be considered from SqB in each iteration. The distribution strategy is as shown the timing diagram in Fig. 5.4. In the timing diagram, the x-axis represents the time and the y-axis represent the processors. We represent the communication by a block above each of the x-axis and computation by a block below the x-axis of a processor. The block $L_{i,k}$ represent the computation of $L_{i,k}$, that is the sub-matrix to be computed by P_i at the k-th iteration. The distribution strategy is as follows. First, P_1 will compute $L_{1,1}$. After it has finished computing $L_{1,1}$, it will continue processing $L_{1,2}$, since it only requires the results from $L_{1,1}$. At the same time, it will send the last row of $L_{1,1}$ to P_2 so that P_2 may start processing $L_{2,1}$ from time $\alpha_1\beta^{(1)}E_1 + 2\beta^{(1)}C$. Note that by the time P_2 finishes computing $L_{2,1}$, it would have received the last row of $L_{1,2}$ (from P_1), hence it can start computing $L_{2,2}$ continuously right after completing the computation of $L_{2,1}$. At the same time instant, it will send the last row of $L_{2,1}$ to P_3. This process continues until P_m.

From the timing diagram, we may observe that there are two phases of communication processes involved for $P_i,\ i = 1, ..., m-1$: **(a)** sending the last row of $L_{i,k}$, ($\mathbf{S}$ and $\mathbf{h}$ matrices), of total size $2\beta^{(k)}$, to P_{i+1} and **(b)** sending $L_{i,k}$ ($\mathbf{S}$ matrix only), of size $\alpha_i\beta^{(k)}$, via $P_{i+1}, P_{i+2}..., P_{m-1}$ to P_m. As stated in the previous section, phase **(a)** is required for $P_{i+1}, i = 1, ..., m-1$ to start its computation. On the other hand, phase **(b)** is required such that P_m can have the complete $\mathbf{S}$ matrix to perform any necessary post-processing such as the trace-back process to determine an optimal alignment. In our strategy, each of the

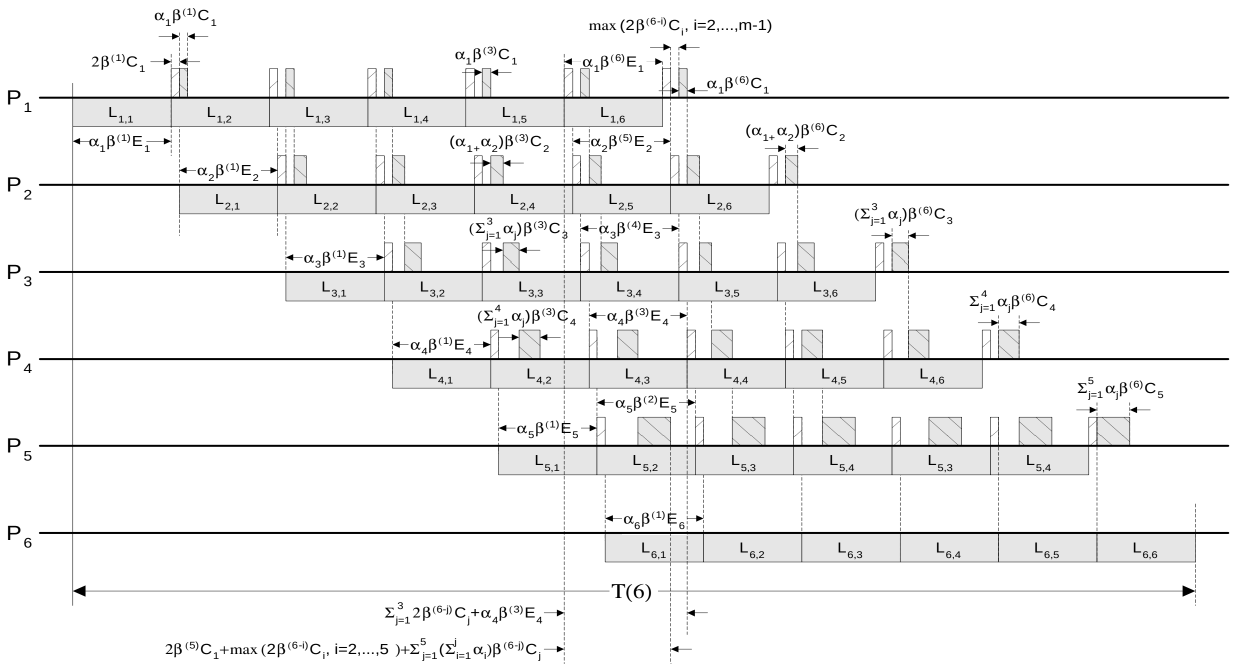


Figure 5.4: Timing diagram when $m = 6$

sub-matrices that was computed by the respective processors will be transmitted to P_m right after the phase **(a)** mentioned above. Thus, whenever the communication link is available and after phase **(a)**, the results of a processor are communicated to P_m. This may be observed from the timing diagram. It may also be noted that **(b)** may not be required in some cases as it is possible to perform post-processing (i.e. trace-back process) at individual processors. We shall discuss this later in this chapter.

Now we shall derive the amount of residues that should be given to $P_i, i = 1, ..., m$ according to the distribution strategy described above. From the timing diagram, we see that,

$$\alpha_{i-1}\beta^{(k)}E_{i-1} + 2C_{i-1}\beta^{(k)} = 2C_{i-1}\beta^{(k-1)} + \alpha_i\beta^{(k-1)}E_i \quad , \quad i = 2, ..., m \ , \ k = 2, ..., Q$$

Alternatively,

$$\alpha_i = \alpha_{i-1}\frac{\beta^{(k)}}{\beta^{(k-1)}}\frac{E_{i-1}}{E_i} + 2\frac{\beta^{(k)}}{\beta^{(k-1)}}\frac{C_{i-1}}{E_i} - 2\frac{C_{i-1}}{E_i} \quad , \quad i = 2, ..., m \ , \ k = 2, ..., Q \tag{5.5}$$

As stated in Section 5.1.3, the front-ends cannot send and communicate at the same time instant. We can observe from Fig. 5.4, in order to avoid front-end collisions, the following condition needs to be satisfied.

$$\sum_{j=1}^{m-3} 2\beta^{(k-j)}C_j + \alpha_{m-2}\beta^{(k-m+2)}E_{m-2} \geq$$

$$2\beta_{k-1}C_1 + \max(2\beta^{(k-i)}C_i, i = 2, ..., m-1) + \sum_{j=1}^{m-1}\left(\sum_{i=1}^{j}\alpha_i\beta^{(k-j)}C_j\right) \quad , \quad k = 1, ..., Q \qquad (5.6)$$

The above inequality is captured using the fact that during k-th iteration $k = 1, ..., Q$, the communication phases **(a)** and **(b)** described above by P_{m-1} must complete on or before the computation process of P_{m-2}, as P_{m-2} needs to send the last row of $L_{m-2,k}$ immediately after processing $L_{m-2,k}$ to P_{m-1}.

The set of equations (5.5) satisfying (5.6), are difficult to solve to yield an optimal solution, as the equations may generate inconsistent values for the unknowns α_i and $\beta^{(k)}$. However, this does not inhibit from deriving a practically realizable solution in which one may attempt to fix the number of residues to be considered in each iteration in order to deliver an acceptable quality solution. Thus, if we set $\beta^{(k)} = \dfrac{\beta}{Q}, k = 1, ..., Q$, then this means that we consume identical number of residues in each iteration. With this modification, we will be able to solve (5.5) together with the fact that $\sum_{i=1}^{m}\alpha_i = \alpha$, to determine the value of $\alpha_i, i = 1, ..., m$ as,

$$\alpha_i = \frac{1}{E_i}\frac{\alpha}{\sum_{j=1}^{m}\frac{1}{E_j}} \quad , \quad i = 1, ..., m \qquad (5.7)$$

However to solve (5.5) we assumes that Q is a fixed or a known parameter. As far as the choice on the value of Q is concerned, we set it to largest possible value, i.e., $Q = \beta$ (implying that $\beta^{(k)} = 1$), to maximize the degree of parallelism that can be achieved. Note that one may have $Q < \beta$, which will only degrade the quality of the solution (speed-up) as shown in Theorem 5.1 below.

Theorem 5.1: The processing time for a m-processor system consuming q iterations is strictly greater than the processing time for the system consuming $q + 1$ iterations.

Proof: Let us redenote the processing time as $T(m, q)$ when q iterations are consumed in aligning two sequences using our strategy. We need to show that $T(m,q) > T(m,q+1)$. From the timing diagram, we see that the processing time using m processors to align the sequences in q number of iterations, includes the computation time of $L_{1,k},\ k = 1,...,q$, communication time for $2\beta^{(q)}C$ (of $(m-1)$ processors) and computation time for $L_{i,q},\ i = 2,...,m$. Hence, we have

$$T(m,q) = \sum_{k=1}^{q} \alpha_1 \beta^{(k)} E_1 + \sum_{j=1}^{m-1} 2\beta^{(q)} C_j + \sum_{i=2}^{m} \alpha_i \beta^{(q)} E_i \tag{5.8}$$

where $\beta^{(k)} = \dfrac{\beta}{q},\ k = 1,...,q$. Substituting these in the above equation, we have,

$$T(m,q) = \alpha_1 \beta E_1 + H\frac{\beta}{q} \tag{5.9}$$

where $H = \sum_{j=1}^{m-1} 2C_j + \sum_{i=2}^{m} \alpha_i E_i$. Using (5.9), we have,

$$T(m,q) - T(m,q+1) = \frac{H\beta}{q(q+1)} > 0 \tag{5.10}$$

Hence the proof. ■

As for the condition (5.6) mentioned above, since we consider $\beta^{(k)} = 1, k = 1,...,Q$ (as we have set $Q = \beta$), we have,

$$\sum_{j=2}^{m-3} 2C_j + \alpha_{m-2}E_{m-2} \geq \max(2C_i, i = 2,...,m-1) + \sum_{j=1}^{m-1}\left(\sum_{i=1}^{j} \alpha_i C_j\right) \tag{5.11}$$

Hence, before we begin aligning *SqA* and *SqB*, we first check if (5.11) can be satisfied. If (5.11) holds, then we will distribute and compute the **S** matrix as proposed above. The processing time is,

$$T(m) = \alpha_1 \beta E_1 + \sum_{j=1}^{m-1} 2C_j + \sum_{i=2}^{m} \alpha_i E_i \tag{5.12}$$

with α_i derived from (5.7). However, it may happen that (5.11) may not be satisfied, i.e., typically when communication links are too slow. This is also equivalent to a situation wherein processor speeds are extremely faster than the link speeds. In such cases, we will need to resolve using alternate (heuristic) strategies. We shall propose two heuristic strategies in the later section of this chapter and an illustrative example to demonstrate their workings.

As mentioned earlier in this section, in some cases, it is possible to execute post-processing, i.e. trace-back process, at individual processors without the need of having entire **S** matrix. Hence the **S** matrix is not required to be sent to P_m for post-processing. Performing post-processes in individual processors, or *distributed post-processing* will significantly relax the condition (5.11) as transmissions of **S** matrix is the prime influential factor in (5.11). In the next section, we shall demonstrate how the trace-back process can be done at individual processors without the entire matrix **S** as well as the improvement that can be achieved.

5.2.2 Distributed post-processing : Trace-back process

In this section we shall demonstrate how the trace-back process can be done at individual processors, eliminating the need to send the **S** matrix to P_m. This may act as a template for any other post-processes that maybe done at individual processors with only the partial **S** matrix.

As mentioned in the Section 5.1.2, the trace-back process starts at the bottom-right of **S** and move towards the top-left by taking the maximum score among the adjacent matrix element on the left, top and top-left. This essentially generate a 'path', in **S**, starting from the bottom-right to top-left. If the trace-back process is to be done at individual processors, each processors has only a portion of the trace-back path, with a starting point (bottom) and an ending point (top) as shown in Figure 5.5. We denote the starting point (bottom) and an ending point (top) of the trace-back path portion in $L_{i,k}, k = 1, ..., Q$ as $S_i^{(b)}$ and $S_i^{(t)}$ respectively.

We shall now describe the distributed trace-back process between two adjacent processors, P_i and P_{i-1}, for the ease of understanding. Let us assume that optimum alignment from $S_m^{(b)}$

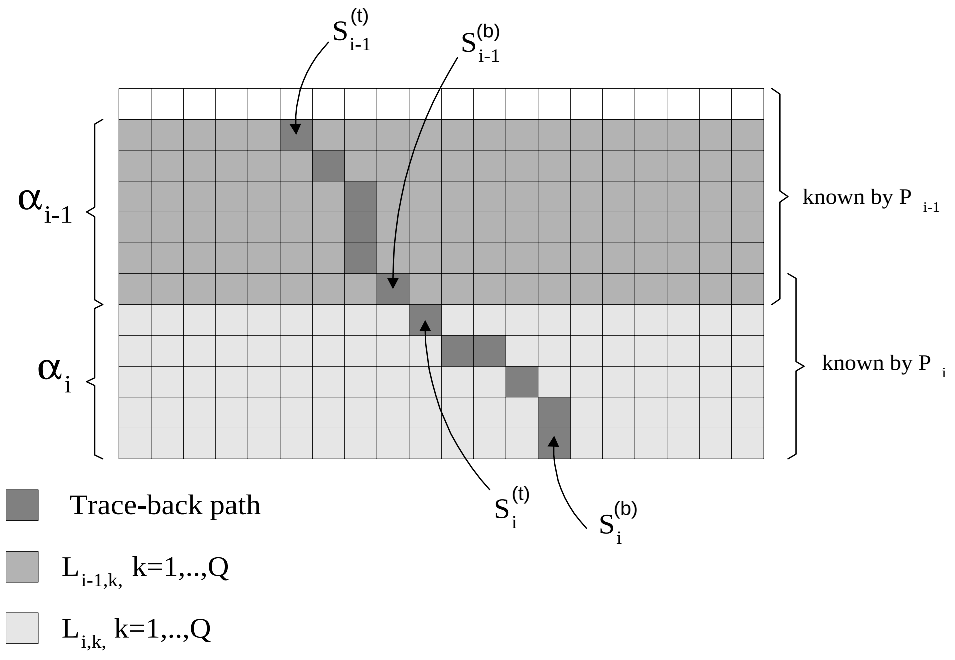


Figure 5.5: Distributed trace-back process between P_i and P_{i-1}

until $S_i^{(b)}$ as well as the point $S_i^{(b)}$ is known by P_i. The trace-back process can be carried out in P_i until the path reaches the top row of $L_{i,k}, k = 1, ..., Q$. The point $S_i^{(t)}$ can be determined by P_i with the previous information given by P_{i-1} when calculating $\mathbf{S}$, that is the last row of $\mathbf{S}$ in $L_{i-1,k}, k = 1, ..., Q$. After $S_i^{(t)}$ has been identified, P_i can then determine $S_{i-1}^{(b)}$ and then transmit this information to P_{i-1} together with the optimum alignment from $S_m^{(b)}$ until $S_{i-1}^{(b)}$.

With distributed trace-back process, all the processors no longer required to send the their respective $\mathbf{S}$ matrix to P_m, reducing the overall communication overhead. The timing diagram is as shown in Fig. 5.6. From the timing diagram, we can see that, in order to avoid front-end collision, at the k-th installment, the total communication time for P_i to transmit data to P_{i+1} (via l_i) and P_{i+1} to transmit data to P_{i+2} (via l_{i+1}), cannot exceed the total processing time of P_i at k-th installment, i.e. $\alpha_i E_i$. This is due to the fact that, at k-th installment, when P_i finish processing $L_{i,k}$, it will need to transmit the respective data to P_{i+1}. At the same time instant, if P_{i+1} is still busy transmitting data to P_{i+2} (via l_{i+1}), front-end collision will occur. Hence, in order to avoid front-end collision the following conditions needs to be

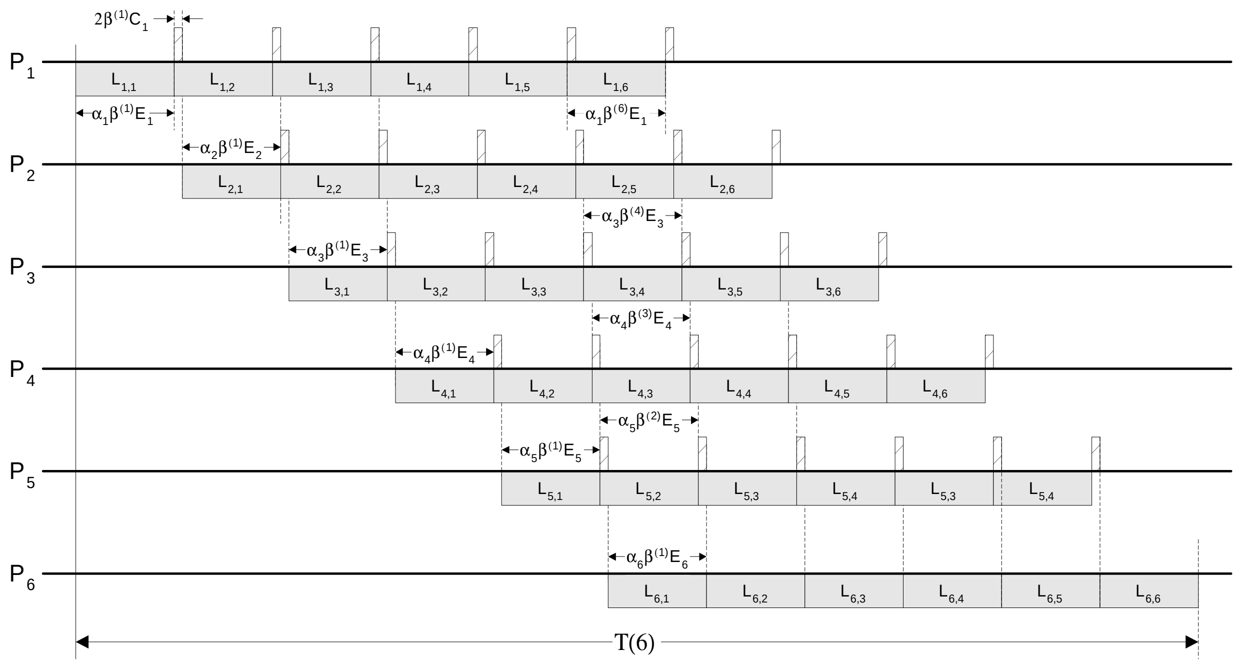

2β(1)C1
P1
P2
P3
P4
P5
P6
L1,1 L1,2 L1,3 L1,4 L1,5 L1,6
α1β(1)E1
α1β(6)E1
α2β(1)E2
L2,1 L2,2 L2,3 L2,4 L2,5 L2,6
α3β(4)E3
α3β(1)E3
L3,1 L3,2 L3,3 L3,4 L3,5 L3,6
α4β(3)E4
α4β(1)E4
L4,1 L4,2 L4,3 L4,4 L4,5 L4,6
α5β(2)E5
α5β(1)E5
L5,1 L5,2 L5,3 L5,4 L5,3 L5,4
α6β(1)E6
L6,1 L6,2 L6,3 L6,4 L6,5 L6,6
T(6)

Figure 5.6: Timing diagram when **S** is not required to be sent to P_m

satisfied,

$$\alpha_i E_i \geq 2(C_i + 2C_{i+1}) \quad , \ \forall i = 1, ..., m-2 \tag{5.13}$$

If the above condition is violated, we have to resolve to heuristic strategies. Nevertheless, as we can see, the above condition is much easier to satisfy compared to (5.11). Hence, performing distributed post-processes can significantly improve the overall performance as the probability of using heuristic strategies decreases. We shall further elaborate this point later in Section 5.4.

5.3 Heuristic Strategy

In this section, we shall propose three heuristic strategies that shall be used to aligned the two sequences in the case when (5.11) and (5.13) cannot be satisfied.

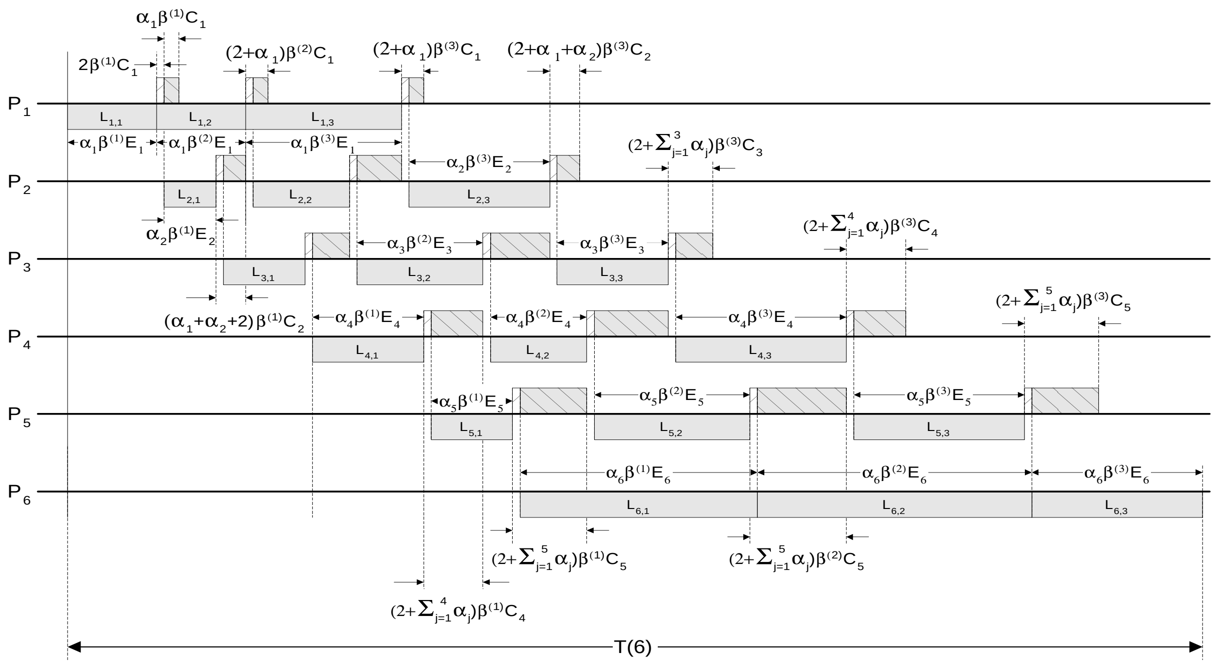


Figure 5.7: Timing diagram for the *idle time insertion* heuristic when $m = 6$

5.3.1 Idle time insertion

This heuristic strategy can be used in the case where distributed post-processing is not possible and (5.11) cannot be satisfied. In this heuristic strategy, we attempt to insert redundant idle time into the computation process to compensate the slow communication links. The timing diagram of this strategy is as shown in Fig. 5.7. From the timing diagram, we can see that,

$$2\beta^{(k-1)}C_1 + \alpha_2\beta^{(k-1)}E_2 + (2 + \textstyle\sum_{j=1}^{2}\alpha_j)\beta^{(k-1)}C_2 = \alpha_1\beta^{(k)}E_1 \quad , \; k = 2, ..., Q$$

Alternatively, taking into account that $\beta_k = 1$ for $k = 1, ..., Q$, we have

$$\alpha_2 \;\; = \frac{\alpha_1}{E_2 + C_2}(E_1 - C_2) - \frac{2C_1 + 2C_2}{E_2 + C_2} \quad , \; k = 2, ..., Q \tag{5.14}$$

Similarly, from timing diagram, we have the relationship of P_i and P_{i-1} where $i = 3, ..., m-1$, we have

$$\alpha_i\beta^{(k-1)}E_i + (2 + \textstyle\sum_{j=1}^{i}\alpha_j)\beta^{(k-1)}C_i \;\; = \alpha_{i-1}\beta^{(k)}E_{i-1} + \displaystyle\sum_{j=1}^{i-1}\alpha_j\beta^{(k-1)}C_{i-1} + 2C_{i-2}\beta^{(k)}$$

Alternatively

$$\alpha_i = \alpha_{i-1}\frac{E_{i-1}}{E_i + C_i} + \sum_{j=1}^{i-1}\alpha_j\left(\frac{C_{i-1} - C_i}{E_i + C_i}\right) + \frac{2C_{i-2} - 2C_i}{(E_i + C_i)} \quad , \ i = 3, ..., m-1 \tag{5.15}$$

Finally, from timing diagram, we have the relationship of P_{m-1} and P_m

$$\alpha_m\beta^{(k-1)}E_m \quad = \alpha_{m-1}\beta^{(k)}E_{m-1} + (2 + \sum_{j=1}^{m-1}\alpha_j)\beta^{(k-1)}C_{m-1} + 2\beta^{(k)}C_{m-2}$$

Alternatively

$$\alpha_m = \frac{(\alpha_{m-1}E_{m-1} + 2(C_{m-1} + C_{m-2}))}{E_m} + \sum_{j=1}^{m-1}\alpha_j\frac{C_{m-1}}{E_m} \tag{5.16}$$

Solving (5.14), (5.15) and (5.16) together with the fact that $\sum_{j=1}^{m}\alpha_j = \alpha$, we will able to obtain the value α_i , $i = 1, ..., m$.

For this heuristic, the conditions that needs to be satisfied are,

$$\alpha_i E_i \geq \textstyle\sum_{j=1}^{i-1}\alpha_j C_{i-1} + 2C_{i-2} \quad , \quad i = 2, ..., m \tag{5.17}$$

where $C_0 = 0$. As we can see, the condition (5.17) is easier to satisfy than (5.11), hence this heuristic strategy may be able to implement even if (5.11) is not satisfied. When (5.17) is satisfied, the processing time will be

$$T(m) = \alpha_1(\beta - 1)E_1 + \sum_{j=1}^{m}(\alpha_j E_j + 2C_j) \tag{5.18}$$

If (5.17) cannot be satisfied, we will then attempt to use the next heuristic strategy.

5.3.2 Reduced set processing

This heuristic strategy can be used in both the cases where (5.11) and (5.13) cannot be satisfied. For ease of understanding, we only discuss the case where distributed post-processing is not possible and (5.11) cannot be satisfy. In the case where distributed post-processing is possible and (5.13) cannot be satisfy, the heuristic strategy is identical.

In this heuristic strategy, we attempt to use less processors to process the load such that (5.11) can be satisfied. From (5.11), we can see that the major cause for the violation of (5.11) is due to the large amount of communication time involved as compare with the computation time. Using less than available processors is able to solve the problem as it will increase the computation time of each processor. The procedure is as follow. First, when (5.11) is violated, we will exclude the last processor from computing the load, i.e., $m = m - 1$. We will then recalculate $\alpha_i, i = 1, ..., m$ of the reduced processors set and check (5.11) again. If (5.11) is still violated, we will exclude another processor and repeat the above procedure until (5.11) can be satisfied. When (5.11) is satisfied with the reduced processors set, we will compute the load as proposed.

5.3.3 Hybrid strategy

In this heuristic strategy, we attempt to utilize both heuristic strategies presented above, for the case when (5.11) cannot be satisfied, in an alternating fashion. Initially, when (5.11) is violated, we will use the *Idle time insertion strategy.* If (5.17) is satisfied, we will align the sequences as proposed. On the other hand, if (5.17) is violated, we will use the *Reduced set processing strategy.* In *Reduced set processing strategy,* we eliminate the last processor from participating in the computation process and check if (5.11) can be satisfied. If (5.11) is satisfied, we will align the sequences as proposed by the distribution. However, when (5.11) is violated, we will attempt to use the *Idle time insertion strategy* on this reduced processors set to check if (5.17) can be satisfied. If (5.17) is violated, we will repeat the *Reduced set processing strategy* and eliminate the last processor (from the reduced processors set) from participating the computation process. The procedure is repeated until either (5.11) or (5.17) is satisfied.

In this heuristic strategy, we attempt to utilize the maximum number of processors possible by continuously checking both the conditions (5.11) and (5.17) before eliminating the last processors from the set. Thus this methodology avoids any inadvertent use of all the processors (due to first strategy) and at the same time attempts to use a maximal (optimal) number of processors (as proposed by the second strategy), thus exploiting the advantages of the two strategies. This method is particulary recommended when the system can affords to spend additional computational time, especially while handling non-time-critical jobs. The following example illustrates the workings of these heuristics strategies.

Example 5.2: We consider aligning two sequences, *SqA* and *SqB*, with length of (number of residues) $150,000$ and $100,000$ respectively. To demonstrate the workings of the heuristics, we consider a homogenous system with parameters $m = 7$, $C_j = C_i = 10$ns/element, $E_j = E_i = 50$ns/element $\forall j \neq i$. First, we calculate the values of $\alpha_i, i = 1, ..., 7$ as described in Section 5.2. The values found are $\alpha_1 = 21429$, $\alpha_2 = 21429$, $\alpha_3 = 21428$, $\alpha_4 = 21429$, $\alpha_5 = 21428$, $\alpha_6 = 21429$, and $\alpha_7 = 21428$. With these values, we observe that (5.11) is violated. We now attempt to use the *Idle time insertion strategy.* The values of $\alpha_i,\ i = 1, ..., 7$ are, 34150, 22766, 18971, 15810, 13175, 10978, and 34150 respectively. With these values, it may be noted that (5.17) does not hold. As a result, we use *Reduced set processing strategy* and eliminate P_7 from participating in the computation. Now, with $m = 6$, we repeat the procedure and obtain $\alpha_1 = 25000$, $\alpha_2 = 25000$, $\alpha_3 = 25000$, $\alpha_4 = 25000$, $\alpha_5 = 25000$, and $\alpha_6 = 25000$, which still violates (5.11). We repeat the procedure until $m = 3$, with values $\alpha_1 = 50,000$, $\alpha_2 = 50,000$, and $\alpha_3 = 50,000$, respectively, satisfying (5.11). Hence, we will distribute the number of residues of *SqA* according to these $\alpha_i,\ i = 1, 2, 3$ and the processing time is given by, $T(3) = 250$ seconds. Finally, to see the speed-up delivered using the multiprocessor solution, the time taken on a single processor is $T(1) = 750$ seconds, which amounts to a speed-up factor of 3.

Suppose we wish to use the hybrid strategy as described above, we proceed as follows. After the first step of the above reduced set strategy, we observe that (5.11) continues to violate. Now, we use the *Idle time insertion strategy* and found that (5.17) is also violated. Then we invoke reduced set strategy and proceed. When $m = 5$, (5.11) is again violated hence we continue using the *Idle time insertion strategy.* Using the heuristic, we found values $\alpha_i, i = 1, ..., 5$ are 40704, 27135, 22613, 18844, and 40704 respectively which satisfies (5.17). Hence we will align the sequences as proposed and the processing time of the heuristic strategy is $T(5) = 203.53$ seconds, which amounts to a speed-up factor of 3.68. As can be seen from this example, although the *Idle time insertion strategy* heuristic is not able to efficiently utilized the processors, it is able to achieve better processing time as it is able to utilize more processors. ■

5.4 Performance Evaluation and Discussions

To quantify and understand the performance of our strategy, we perform rigorous simulation experiments to compare the processing time of our strategy with the direct implementation of Smith-Waterman algorithm using a single machine (non-parallel version). We define speed-up as,

$$\text{Speed-up} = \frac{T(1)}{T(m)} \tag{5.19}$$

where $T(m)$ is the processing time of our strategy on a system using m-processors. $T(1)$ is the processing time using a single processor and is given by,

$$T(1) = \alpha\beta E_1 \tag{5.20}$$

In our experiments, we consider the influencing parameters of communication link speeds and number of available processors. Further, in order to show the effectiveness of our strategy in exploiting the advantages of linear networks, we perform comparative experiments against similar strategy in bus network. We categorize the experiments in a systematic fashion and describe them as follows.

5.4.1 Effects of communication links speeds and number of processors

As stated in Section 5.1.3, in this paper, we consider a loosely coupled multiprocessors system where communication delays are taken into account. From (5.11) and (5.13), we can sense large delays, owing to the presence of slow communication links, is one of the major factors that limits the use of processors, i.e., when the 'Reduced set processing' heuristic strategy is used. We perform rigorous experiments to observe the effects of communication links speeds on the performance of our strategy with, in systems with different number of processors.

In this experiments, we consider case when distributed post-processing is not possible, i.e. the result, $\mathbf{S}$, is required to be sent to P_m. We consider a homogenous system with number of processors varied between $m = 3$ to 25 with $E_x = E_y = E$ chosen in the range $[15.0, 45.0]$ time units/element, $\forall x \neq y$ following a uniform probability distribution. We vary the link speed parameter in the range $[0.5, 10.0]$ time units/element. We consider two real-life DNA samples in our experiments. First sample is the DNA of house mouse mitochondrion (Mus Musculus Mitochondrion, NC_001569, denoted as SqA) consisting of 16,295 residues and the DNA of human mitochondrion (Homo sapiens mitochondrion, NC_001807, denoted as SqB) consisting of 16,571 residues, obtainable from the GenBank [48], in our experiments. The choice of these DNA samples are very typical in such sequence alignment studies reported elsewhere as drug manufacturers often use mouse model to test their products and understanding the differences and similarities between the two species is utmost important. The results are as shown in Fig. 5.8.

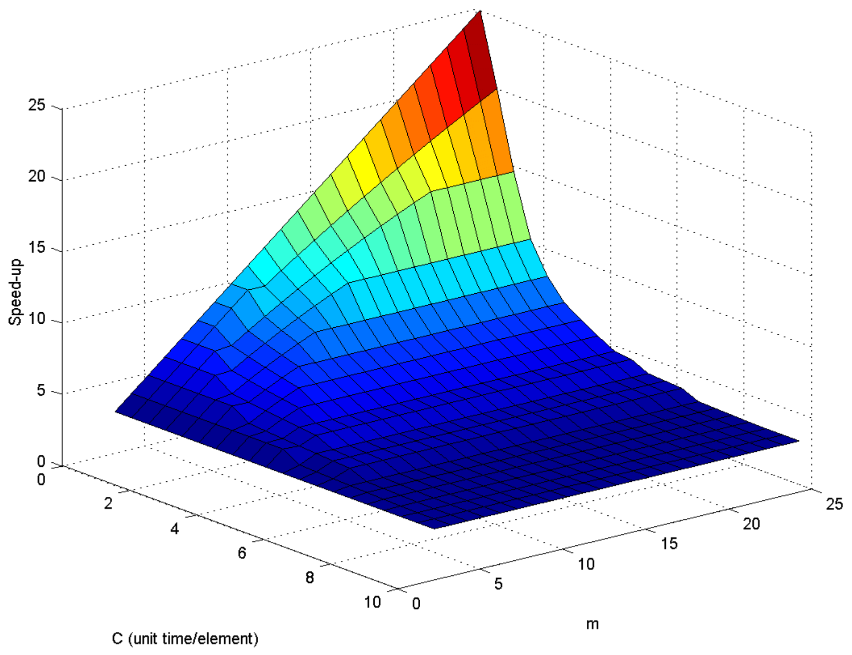


Figure 5.8: Effect of communication link speed and number of processors on the speed-up when **S** is required to be sent to P_m

It may be noted that when the communication delays are dominate (owing to a slow links) heuristic strategies may be in place for processing, as (5.11) may not hold. This fact can be captured from our experiments when $C > 7$ time units/element, where the speed-up saturated at 4 due to the fact when the communication links are slow, both (5.11) and (5.17) cannot be easily satisfied and hence the number of processors that can be utilized is limited. On the other hand, when the links are relatively fast, the strategy is able to give a linear speed up with respect to the number of processors available in the system. This can be see from the figure when $C \leq 2$. It may be noted that, at $C \approx 2$ and $m \approx 7$, we observe that there are changes in the slope of the speed up. This changes are due to the use of *Idle time insertion* strategy (when (5.11) cannot be satisfied) which is less efficient compared to the optimal distribution strategy proposed in Section 5.2.

We perform the identical study in the case where distributed post-processing is possible, i.e. the resulting matrix **S** is not required to be sent to P_m (distributed trace-back process) as

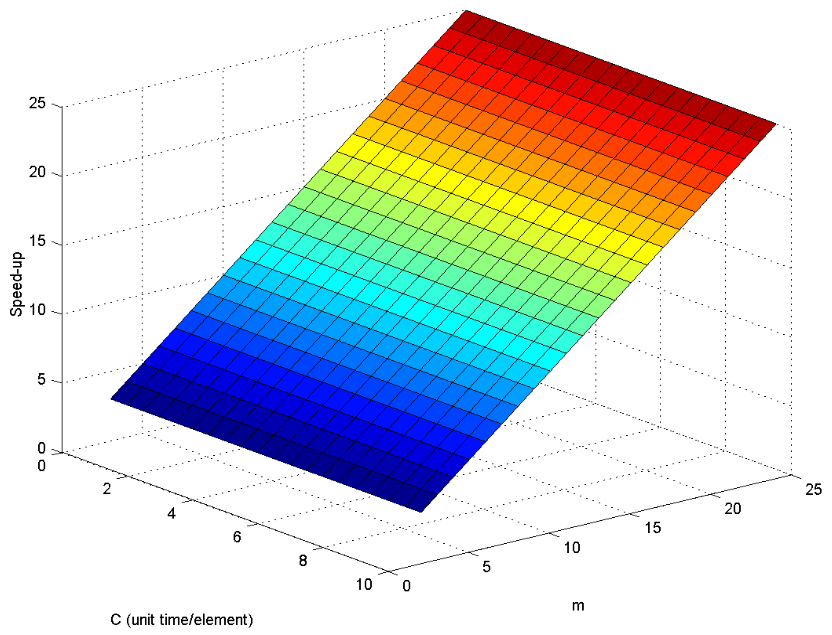


Figure 5.9: Effect of communication link speed and number of processors on the speed-up when **S** is not required to be sent to P_m

discussed in Section 5.2.2. The results are as shown in Fig. 5.9. We observe that the speed-up generated in this case is linear with respect to the number of processors in the system. This is due to the fact that slow communication links has insignificant effect on the overall performance as the data required to be transmitted are small in size. As a result, the condition (5.13) can always be satisfy in the experiment and hence none of the heuristic strategies have been executed.

From the experiments above, we observe that the transmitting large amount of data, i.e. the **S** matrix, is the major factor in determining the performance of the strategy. This is particularly obvious in the case of linear networks as the data being transfer (i.e. from P_1 to P_m) are required to percolate through the system and increasing in size as it passes from processor to processor. Nevertheless, in the case where data are only required to be sent in between adjacent processors, i.e. in cases where distributed post-processing is possible, the independent communication links in linear networks are able to provide significant advantages.

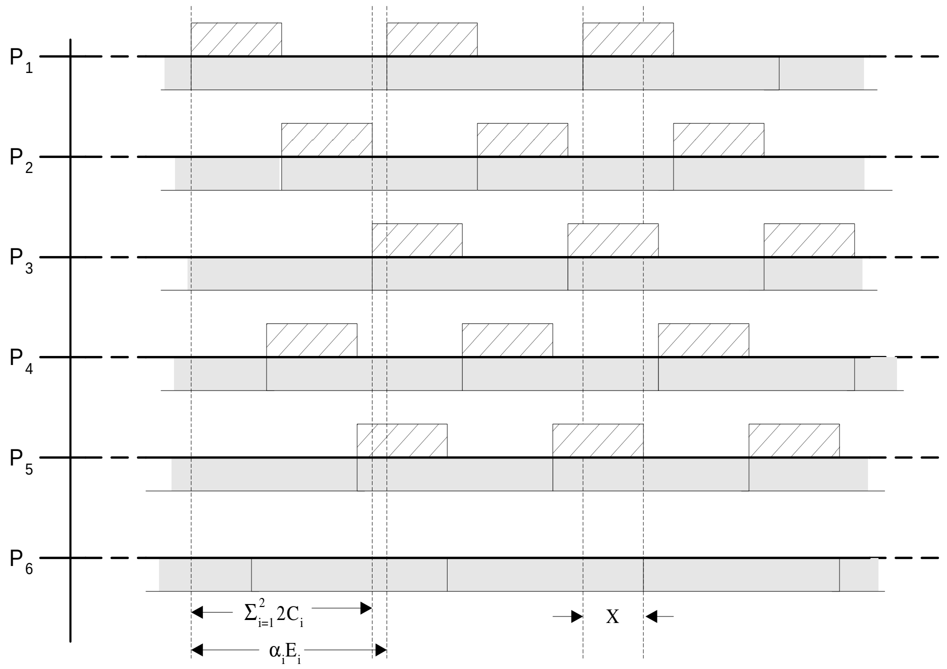


Figure 5.10: Extreme case when the condition (5.13) is at the verge of being satisfied

This can be clearly seen in extreme cases where conditions such as (5.13) is at the verge of being satisfied, i.e. when

$$\alpha_i E_i \approx 2(C_i + C_{i+1}) \ \ , \ \forall i = 1, ..., m-2 \tag{5.21}$$

In such cases, the independent links in linear network are able to transmit data concurrently, as shown in Fig. 5.10. From the figure, we observe that at time interval **X**, maximal number of links ($\frac{m}{2}$) are being utilized at the same time instant, i.e. links l_1, l_3, and l_5 are transmitting concurrently.

5.4.2 Performance evaluation against the bus network architecture

As stated in the Chapter 1, although the linear networks have a complex pipelined communication pattern that may incurs large communication delay, the independent communication links between processors in a linear network may offers significant advantages, depending on the underlying applications. In order to evaluate these advantages, we design similar strategy

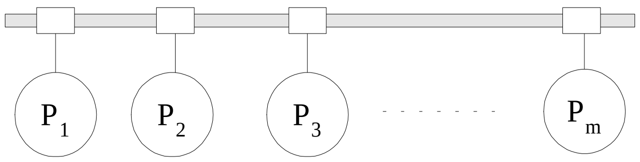


Figure 5.11: Bus network architecture with m processors

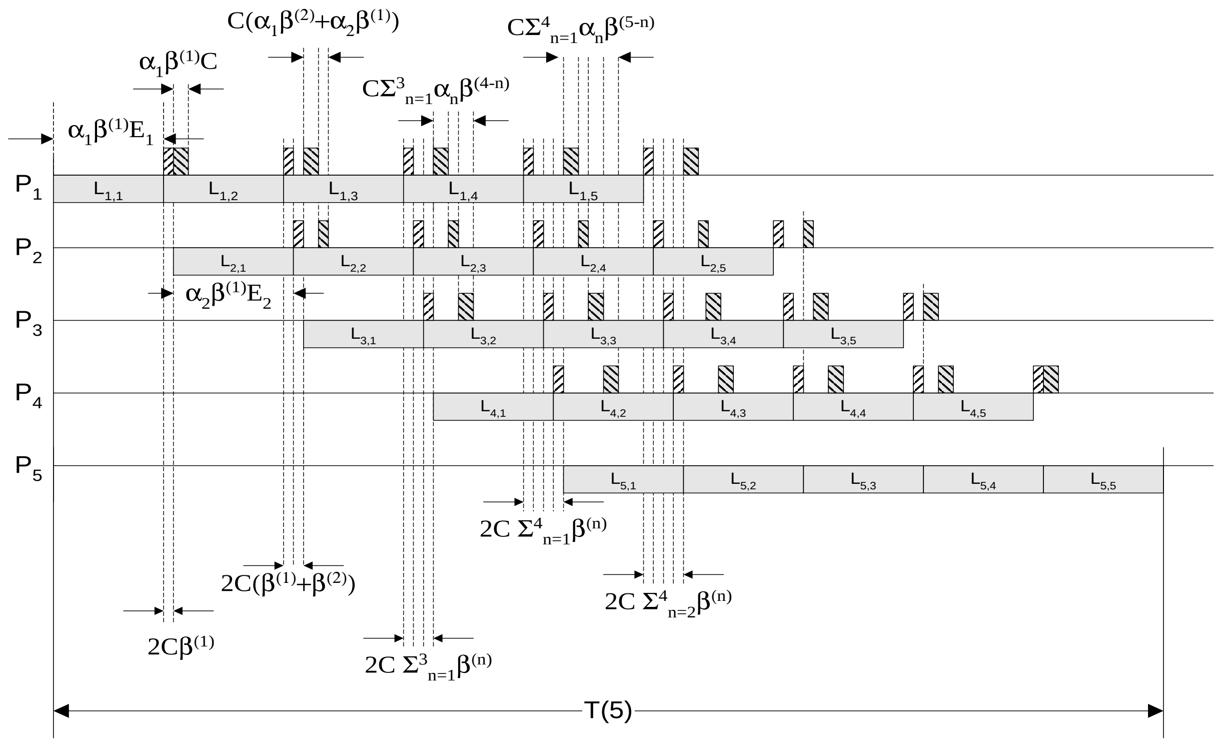


Figure 5.12: Timing diagram of the distribution strategy when $m = 5$, and $Q = 5$

for the Bus network (single level tree) topology, which has a simple communication pattern but only allow a pair processors are to communicate at any time instant.

5.4.2.1 Load distribution strategy in bus networks

In this section, we consider design of load distribution strategy for a multiprocessor system, interconnected by a bus communication link, as shown in Fig. 5.11, with m processors denoted as $P_1, P_2, P_3, ..., P_m$. The distribution strategy for the bus network is as shown in the timing diagram in Fig. 5.12. Similar to Section 5.2.1, we derive following equation from the timing diagram.

$$\alpha_i = \frac{1}{E_i} \frac{\alpha}{\sum_{j=1}^{m} \frac{1}{E_j}} \quad , \quad i = 1, ..., m \tag{5.22}$$

As stated above, in the bus network architecture, only one pair of processors can communicate at any time instant. Hence, we observe that, from Fig. 5.12, in order to avoid communication overlap/contention, the following condition needs to be satisfied.

$$C \leq \frac{\alpha}{\sum_{i=1}^{m} \frac{1}{E_i} (\sum_{j=1}^{m-1} \alpha_j + 2(m-1))} \tag{5.23}$$

In the case where distributed post processing is possible, the above equation can be simplified as

$$C \leq \frac{\alpha_1 E_1}{2(m-1)} \tag{5.24}$$

The conditions (5.23) and (5.24) will be used to verify if the optimal solution is feasible. If these conditions are violated, we will then resolve to heuristic strategy. The heuristic strategy **Reduced set processing** described in Section 5.3, will be use for the case of bus networks when the conditions (5.23) and (5.24) are violated.

5.4.2.2 Performance evaluation

Experiments were performed to determine the performance of the strategy in bus networks with identical system parameters in the experiment described in Section 5.4.1. The experiments include the case where (a) distributed post processing is not possible (b) distributed post processing is possible, respectively. The results are as shown in Fig. 5.13 and Fig. 5.14, respectively. As we can observe from both Fig. 5.9 (linear networks) and Fig. 5.14 (bus networks), for the case when distributed post processing is possible, both architecture perform equally well. To differentiate the performance, we increase the number of processors to the range of [150 : 350] and repeat the experiments. The results is as shown in Fig. 5.15 and Fig. 5.16.

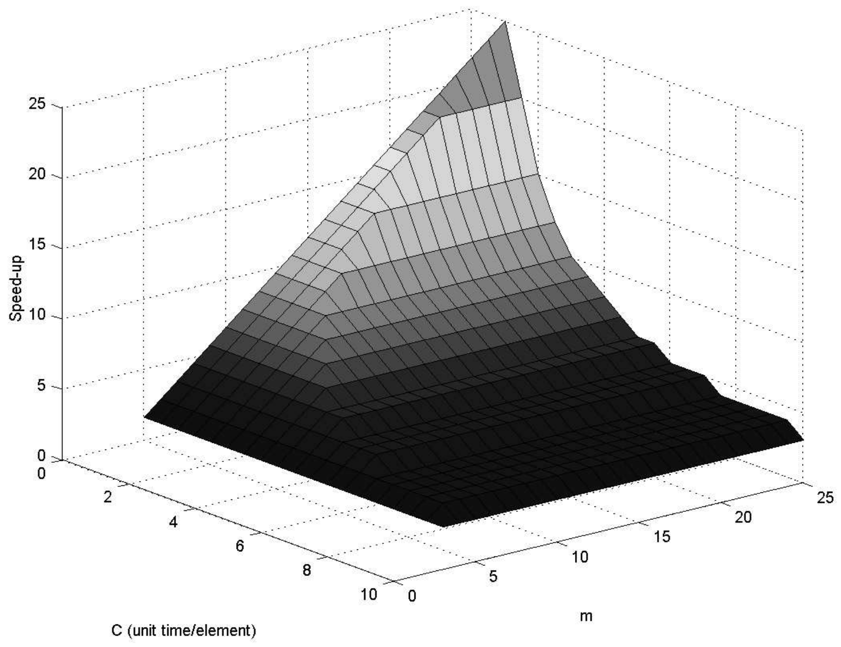


Figure 5.13: Effect of communication link speed and number of processors on the speed-up when **S** is required to be sent to P_m, in bus network

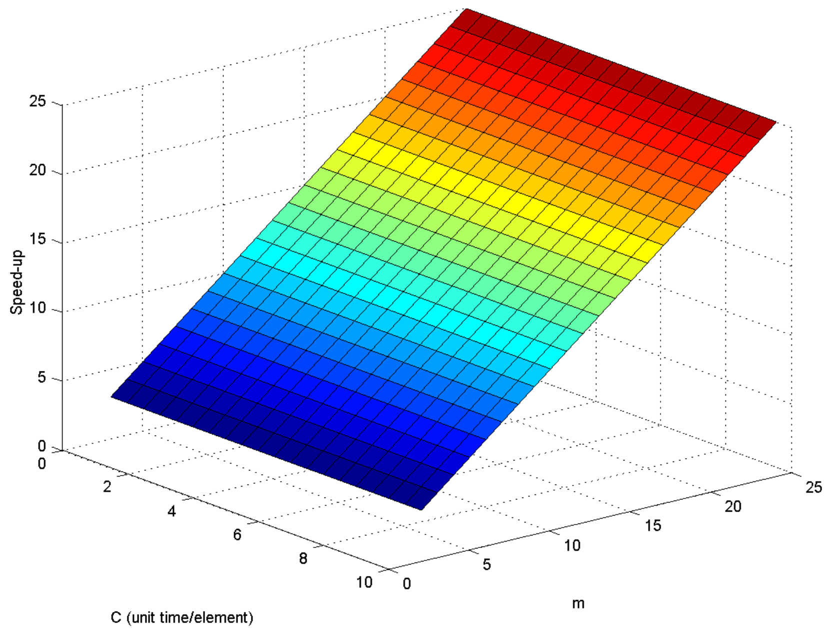


Figure 5.14: Effect of communication link speed and number of processors on the speed-up when **S** is not required to be sent to P_m, in bus network

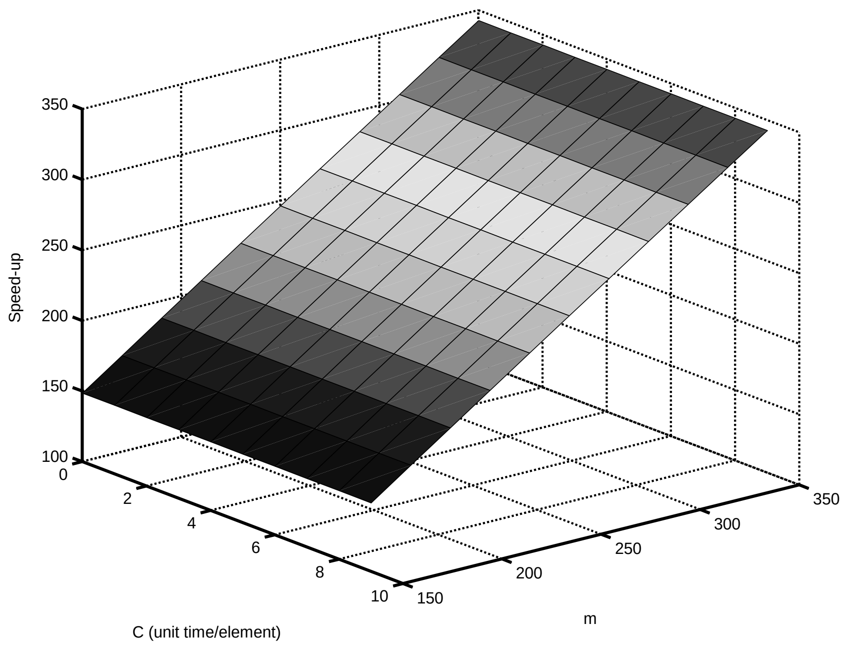


Figure 5.15: Effect of communication link speed and large number of processors on the speed-up when **S** is not required to be sent to P_m

From the results of these experiments, we observe that in the case where distributed post processing is not possible, Fig. 5.8 and Fig. 5.13, the strategy implemented in bus network is able to perform better than linear network. This is due to the pipeline communication pattern in linear networks where data are percolated down the system inducing long communication delays. This restrict the number of processors that can be used to compute the load. On the other hand, in the case where distribute post processing is possible, linear networks out perform bus networks. This is due to the fact that the mentioned disadvantage of linear networks do not have any effect in this case as the data are not required to be percolated to P_m. Further, the independent links in linear networks enable more processors to participate in processing the load. This can be observed from the conditions (5.13) and (5.24) for the linear and bus networks respectively. From these equations, we can see that condition (5.13) is much easier to be satisfied than (5.24), as a result more processors can be used to process the load.

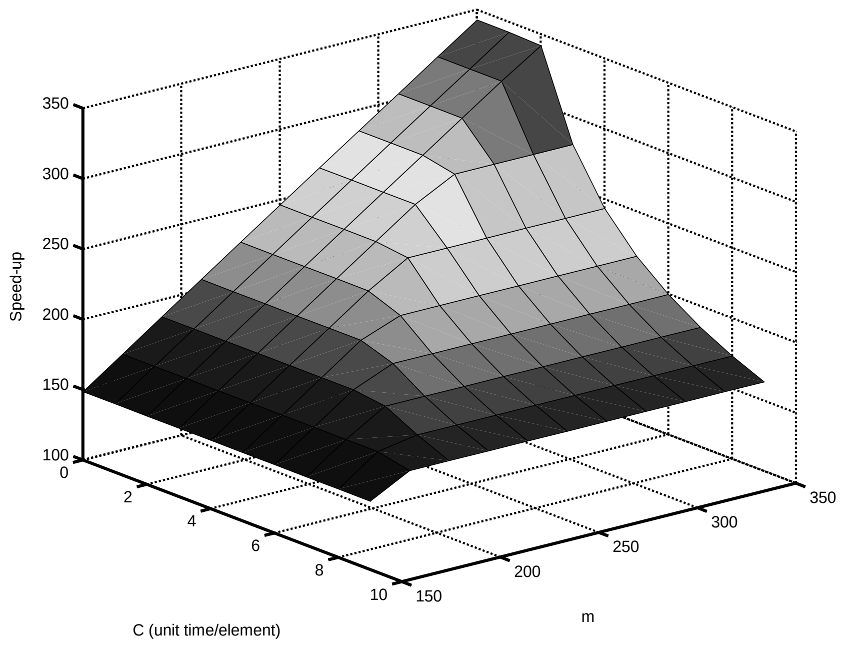


Figure 5.16: Effect of communication link speed and large number of processors on the speed-up when **S** is not required to be sent to P_m, in bus network

5.5 Concluding Remarks

The problem of aligning two biological sequences is addressed in this chapter. We proposed an efficient multiprocessor solution that uses a loosely coupled linear daisy chain networks, where communication delays are non-zero and are taken into consideration. In the design of our strategy, we utilized DLT paradigm to determine the exact amount of residues to be assigned to each of the processors such that the processing time is minimal. In our strategy, we employ the Smith-Waterman algorithm, a widely used algorithm, to achieve a high degree of parallelism. Our strategy can be easily implemented with other algorithms which uses Smith-Waterman algorithm as well as with other similar procedures.

A systematic approach is presented in deriving our strategy. Firstly, we divide the **S** matrix into sub-matrices, where the processors in the system will compute the respective sub-matrices in more than one iteration. We derived equations that will determine the size of these sub-matrices according to the processors speeds and communication links speeds. In designing our strategy, we exploit the advantage of the linear networks' independent communication links by enabling concurrent data transmission. Finally, we derived a condition to check if an optimal solution can be achieved. We then considered the cases of distributed post-processing where post-processing can be done at individual processors. Distributed post-processing offer significant advantages as the communication overhead is substantially reduced. Similarly, a condition is also derived to determine if optimal solution can be guaranteed.

In cases where optimal solution cannot be achieved, we will then resolve to heuristic strategies. In designing heuristic strategies, there are a lot of factors that can be taken into consideration. In the design of our first heuristic strategy, we introduce redundant idle time in the computation process to compensate the slow communication links. Whereas, in our second heuristic strategy, we use only a subset of processors in the computation process. This will provide more communication time at the cost of longer processing time. The key difference between the strategies is that the first one attempts to retain all the available processors in the hope of achieving maximum possible speed-up whereas, the second strategy attempts to carefully considers the number of processors that can be used at every step whenever necessary. In the first strategy, we derive a condition similar to (5.11) which in this case, by and large, is simple to satisfy. However, it may be possible that sometime even this simple condition may not hold. In this case, we choose the second strategy for processing which guarantees that processing will be completed, as in this second strategy there will be at least one processor to complete the processing in the worst case scenario.

In Section 5.4, we studied the performance of our strategy in the effect of slow communication links. From (5.11) and (5.13), we can see that slow communication links is one of the major factors that limits the performance of the strategies, i.e., the number of processors used is restricted when 'Reduced set processing' heuristic strategy is used. We performed simulated experiments to evaluate the performance of our strategies (both with and without distributed post-processing) in various scenarios with different number of available processors and communications link speeds. The experiments results showed that, for fast communication links, our strategies are able to achieve linear speed-up. Further, it also shows the significant improvements in the case when distributed post-processing is possible.

We also performed experiments to evaluate the performance of our strategy in exploiting the advantages of linear networks. Experiments are performed to compare the performance of the strategy when implemented in linear and bus networks respectively. Results show that when the strategy is executed in linear networks, more processors are able to participate in computing the load. This is due to the fact that our strategy is able to exploit the advantages of linear networks where processors are able to communicate concurrently.

Chapter 6

Conclusions and Future Work

The design and analysis of load distribution strategies for linear networks with various real-life constraints are considered in this thesis. Designing load distribution strategies for linear networks is a challenging task as the communication of linear networks involved a complex pipelined communication pattern through the $m-1$ independent links. In designing these strategies, the communication pattern has to be taken into consideration to avoid any collision among the communication resources, i.e. the front-ends. Although linear networks may incur unnecessary complexity into the design of load distribution strategies, the independent links in linear networks may be able to provide some significant advantages as concurrent communications are possible.

In the DLT literature, extensive studies and experiments [5, 11, 33] have been carried out for load distribution strategies in linear networks for a single divisible load. However, in reality, the dedicated network based parallel processing system is most likely to be given more than one load to be processed. Further, in real-life scenario, it may also occur that the processors in the system are busy with other form of computational tasks such that it will not able to process the arriving loads when the loads arrive.

In Chapter 3, we designed a load distribution strategy for handling multiple divisible load in linear networks. In designing the strategy, the conditions of the previous load are taken into consideration when scheduling the current load in order to minimize the unutilized computational time. Further, we take into consideration of the availability of the front-ends and derived a set of conditions that will guarantee a collision-free front-end operation among the adjacent loads for both single and multi-installment strategies. In the case where multi-installment strategy is used, it may happen that a feasible number of installments does not exist and we resolve this situation by using heuristic strategies. Two heuristic strategies, referred to as A and B, are proposed. The choice of heuristic strategies depends on several issues. Heuristic A, which utilizes single-installment distribution strategy, may offer computation simplicity; while for Heuristic B, which utilized multi-installment distribution strategy, may offer better performance. Rigorous simulated experiments have been performed to evaluate the performance of these heuristic strategies under various conditions. In our experiments, we designed a number of strategies that utilized the combinations of Heuristic A, Heuristic B, and the optimal distribution strategy. We also considered the cases where the set of loads may be sorted with the largest load first or last. Finally, simulations are performed to show the significant improvements that can be achieved using our proposed strategy as compared to using the strategy for single load, when processing multiple loads.

As an important extension of the work in Chapter 3, we considered the problem of designing load distribution strategy for linear network with arbitrary processor release times, in Chapter 4. In this work, we considered the practical situation where the processors in the system are occupied with other computational tasks such that the processors are not able to process any in coming load instantly when the load arrive. In this work, we systematically considered all possible cases that can arise. If the processors are idle at the time of arrival of the load, the idle case algorithm presented in the literature [5, 33] can be immediately used. As done in the literature, we considered two possible cases of interest, namely identical release times and non-identical release times. In the case of identical release times, we derived a condition to determine if the load can be fully distributed to all processors within a single-installment and will resolve to multi-installment strategy when this condition is violated. For the case of non-identical release times, the problem becomes much more challenging as utilizing all the available processors may not be beneficial, i.e. using processors with significantly large release time will increase the overall processing time. As such, we designed a recursive algorithm that is able to determine the optimal (qualified) set of processors that should participate in processing the load. We have also proved that the load can be fully processed by the set of processors before the release times of any of the un-qualified processors. Similar to the work in Chapter 3, conditions have been derived to determine if these strategies can be used and we will then resolve to heuristic strategies if these conditions are violated.

Finally, as to complete our analysis on distribution strategies in liner networks, we designed a strategy that is able to fully harness the advantages of the independent links in linear networks. We studied various possible applications of DLT and considered the bioinformatics problem of aligning biological sequences. For the first time in the domain of DLT, bioinformatics problems were attempted. Our objective is to design a load distribution strategy such that the overall processing time is a minimal. In designing our strategy, we utilized the popular Smith-Waterman algorithm which is able to determine the optimal alignment of two biological sequences. We exploited the characteristic of the algorithm and designed a parallel implementation of the Smith-Waterman algorithm with high degree of parallelism. We consider two possible cases of aligning sequences where the results are required and not required, to be collected at a processor. We also proposed a method that enables the trace-back process to be performed at individual processor such that the results need not be collected at any processor. Finally, as to highlight the advantages of the independent links in linear networks, we implemented the similar strategy in bus networks and compare the performance between these two network topologies.

There are several possible future extensions for the works in this thesis. Firstly, we can consider designing load distribution strategies under a combined influence of both the processor release times and communication link release times constraints. Clearly, in this case, designing load distribution strategies will be more challenging. Another extension that was not yet attempted in the literature is to attack the similar problem with the load originates at an interior processor. Since there exists two possible sequences of load distribution in this case [6], attempting to derive optimal load distribution will be an interesting problem and finally, it may be noted that the solution and the strategies for the interior case may also be applicable to boundary case situation addressed in this thesis. Alternatively, one may consider extending the bioinformatics works presented in this thesis by performance real-life experiments with large scale biological sequence database. Finally, performance evaluations can also be done between the presented strategies and existing strategies in the bioinformatics literature.

Bibliography

[1] Yu, D. and T.G. Robertazzi, "Divisible Load Scheduling for Grid Computing", Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003), November 2003.

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

[3] Gerogiannis D. and S. C. Orphanoudakis, "Load balancing requirements in parallel implementations of image feature extraction tasks", IEEE Transactions on Parallel Distributed Systems, 4, pp. 994-1013, 1993.

[4] Choudhary A. N. and R. Ponnusamy, "Implementation and Evaluation of Hough Transform Algorithms on a Shared-memory multiprocessor", Journal of Parallel Distributed Computing, 12, pp. 178-188, 1991.

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

[6] 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.

[7] Bharadwaj, V., D. Ghose, and T. G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems", *Special Issue on Divisible Load Scheduling in Cluster Computing*, Kluwer Academic Publishers, January 2003.

[8] Bharadwaj, V., D. Ghose, and T. G. Robertazzi, " Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems", http://opensource.nus.edu.sg/∼elebv/DLT.htm

[9] Robertazzi, T. G., "Scheduling in parallel and distributed systems", http://www.ee.sunysb.edu/∼tom/dlt.html

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

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

[12] Drozdowski M. and W. Glazek, "Scheduling Divisible Loads in a Three Dimensional Mesh of Processors", *Parallel Computing*, 25, pp. 381-404, 1999.

[13] Glazek, W., "A Multisate Load Distribution Strategy for Three-Dimensional Meshes", *Special Issue on Divisible Load Scheduling in Cluster Computing*, Kluwer Academic Publishers, January 2003.

[14] 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.

[15] Ghose, D., and H. J. Kim, "Load Partitioning and Trade-Off Study for Large Matrix-Vector Computations in Multicast Bus Networks with Communication Delays", *Journal of Parallel and Distributed Computing*, Vol. 55, No. 1, pp. 32-59, November 1998.

[16] Li, K., “Managing Divisible Loads in Partitionable Networks”, in High Performance Computing Systems and Applications, J. Schaeffer and R. Unrau (Ed.), Kluwer Academic Publishers, pp. 217-228, 1998.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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, January 2003.

[24] Kim, H.-J., “A Novel Optimal Load Distribution Algorithm for Divisible Loads”, *Special Issue on Divisible Load Scheduling in Cluster Computing*, Kluwer Academic Publishers, January 2003.

[25] Ghose, D., “A Feedback Strategy for Load Allocation in Workstation Clusters with Unknown Network Resource Capabilities using the DLT Paradigm,” *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’02)*, Las Vegas, Nevada, USA June 2002, Vol. 1, pp. 425-428.

[26] Drozdowski Maciej, and P. Wolniewicz, “Out-of-Core Divisible Load Processing”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 10, pp. 1048-1057, October 2000.

[27] 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.

[28] 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*, 58, pp. 71-79, 2001.

[29] Ghose, D. and H.J. Kim, “Matrix-vector Product Computations on Multicast Bus-Oriented Workstation Clusters”, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’02)*, Las Vegas, Nevada, USA, Vol. 1, pp. 436-441, June 2002.

[30] Bharadwaj, V., and G. Barlas, "Access time minimization for distributed multimedia applications", *Special Issue on Multimedia Authoring and Presentation in Multimedia Tools and Applications*, Kluwer Academic Publishers, Issue 2/3, pp.235-256, November 2000.

[31] Balafoutis E., Paterakis M., P. Triantafillou, G. Nerjes, P. Muth, and G. Weikum, "Clustered Scheduling Algorithms for Mixed-Media Disk Workloads in a Multimedia Server", *Special Issue on Divisible Load Scheduling in Cluster Computing*, Kluwer Academic Publishers, January 2003.

[32] Yongwha C., and V. K. Prasanna, "Parallelizing Image Feature Extraction on Coarse-Grain Machines", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 20, no 12, 1389-1394, December 1998.

[33] 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.

[34] Bharadwaj, V., X. Li, and C. C. Ko, "Efficient Partitioning and Scheduling of Computer Vision and Image Processing Data on Bus Networks using Divisible Load Analysis", *Image and Vision Computing,* Elsevier Science, Vol. 18, No. 11, pp. 919-938, August 2000.

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

[36] Goswami, K. K., M. Devarakonda, and R.K. Iyer, "Prediction-Based Dynamic Load-Sharing Heuristics", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 638-648, June 1993.

[37] Ahmad, I., A. Ghafoor, and G.C. Fox, “Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers”, *Journal of Parallel and Distributed Computing*, vol. 20, pp. 317-329, 1994.

[38] 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.

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

[40] 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.

[41] Gribskov, M., and D. John, “Sequence Analysis Primer”, *University of Wisconsin Biotechnology Center(UWBC) Biotech Resource Series*, 1991.

[42] Waterman, M. S., “Mathematical Methods for DNA Sequences”, *Boca Raton, Florida, CRC Press Inc.*, 1986.

[43] Needleman, S. B., and C.D. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Sequences” *Journal of Molecular Biology*, Vol. 48, pp. 443-453, 1970.

[44] Sellers, P. H., “On the Theory and Computation of Evolutionary Distances”, *SIAM Journal of Applied Mathematics*, 26, pp.787-793, 1974.

[45] Waterman, M. S., T. F. Smith, and W. A. Beyer, “Some Biological Sequence Metrics”, *Advances in Mathematics*, 20, 367-387, 1976.

[46] Smith, T. F., and M. S. Waterman, “Identification of Common Molecular Subsequence,” *Journal of Molecular Biology*, 147, pp. 195-197, 1981.

[47] Gotoh, O., “An Improved Algorithm for Matching Biological Sequences”, *Journal of Molecular Biology*, 162, pp. 705-708, 1982.

[48] GenBank - http://www.ncbi.nlm.nih.gov

[49] The EMBL (European Molecular Biology Laboratory) Nucleotide Sequence Database - http://www.ebi.ac.uk/embl

[50] DNA Data Bank of Japan - http://www.ddbj.nig.ac.jp

[51] Benson, Dennis A., K. Ilene, J. L. David , O. James, A. R. Barbara , and L. W. David, “GenBank”, *Nucleic Acids Research*, Vol. 28, No. 1, pp. 15-18, 2000.

[52] Lipman, D. J., and W. R. Pearson, “Rapid and Sensitive Protein Similarity Searches”, *Science*, 227, pp. 1435-1441, 1985.

[53] Pearson, W. R., and D. J. Lipman, “Improved Tools for Biological Sequence Comparison”, *Proceedings of the National Academy of Sciences USA*, 85, pp.2444-2448, 1988.

[54] Pearson, W. R., “Rapid and Sensitive Sequence Comparison with FASTA and FASTP”, *Methods in Enzymology*, 183, pp. 63-98, 1990.

[55] Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. Lipman, “A Basic Local Aligment Search Tool”, *Journal of Molecular Biology*, 215, pp. 403-410, 1990.

[56] Califano, A., and I. Rigoutsos, “FLASH: A Fast Look-Up Algorithm for String Homology”, *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, pp. 56-64, 1993.

[57] Myers, W. Eugene, “An O(ND) Difference Algorithm and Its Variations”, *Algorithmica*, vol 1, 2, pp. 251-266, 1986.

[58] Yap, T. K., O. Frieder, and R. L. Martino, "High Performance Computational Methods for Biological Sequence Analysis" *Kluwer Academic Publishers*, 1996.

[59] Yap, T.K., F. Ophir, L. Robert, "Parallel Computation in Biological Sequence Analysis", *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, 3, March 1998.

[60] Trelles, O., M.A. Andrade , A. Valencia, E.L. Zapata, and J.M. Carazo, "Computational Space Reduction and Parallelization of a New Clustering Approach for Large Groups of Sequences", *Bioinformatics*, Vol 14, 5, pp. 439-451, June 1998.

[61] Rognes, Torbjorn, and S. Erling, "Six-fold Speed-up of Smith-Waterman Sequence Database Seaches Using Parallel Processing on Common Microprocessors", *Bioinformatics*, 16(8), pp. 699-706, 2000.

[62] Berger, M.P., and P.J. Munson, "A Novel Randomized Iteration Strategy for Aligning Multiple Protein Sequences", *Computer Applications in The Bioscience*, 7, pp. 479-484, 1991.

[63] Myers, E.W., and W. Miller, "Optimal Alignments in Linear Space", *Computer Applications in The Biosciences*, 4, pp. 11-17, 1988.

[64] Dayhoff, M., R.M. Schwartz, and B.C. Orcutt, "A Model of Evolutionary Change in Protiens", *Atlas of Protien Sequences and Structure*, 5, pp. 345-352, 1978.

Author's Publications

[1] Bharadwaj, Veeravalli, and Wong Han Min, "Scheduling Divisible Loads on Heterogeneous Linear Daisy Chain Networks with Arbitrary Processor Release Times", To appear in *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, February 2004

[2] Wong Han Min, Bharadwaj Veeravalli, and Gerassimos Barlas,"Scheduling Multiple Divisible Loads on Heterogeneous Linear Daisy Chain Networks", In the Proceedings of the *International Conference on Parallel and Distributed Computing Systems (PDCS) 2002*, Cambridge, USA, 2002.

[3] Wong Han Min, Bharadwaj Veeravalli, and Gerassimos Barlas, "Design and Performance Evaluation of Load Distribution Strategies for Multiple Divisible Loads on Heterogenous Linear Daisy Chain Networks", (submitted to *Journal of Parallel and Distributed Computing*), 2003

[4] Wong, Han Min, and Bharadwaj Veeravalli, "Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach", (submitted to *IEEE Transactions on Information Technology in Biomedicine*), 2003.

Appendix

Derivation for Case 2 in Chapter 4, Section 4.2.4

Consider a set of processors $S_0 = \{P_g, P_{g+1}, ..., P_{x-1}, P_x\}$ such that the release time of $P_i \in S_0$ is given by $\tau_i = 0$. Note that the actual indices of the processors may be actually different. That is, $P_g = P_3, P_{g+1} = P_{11}$, and so on. We generate a set of equations to determine the load fractions to be assigned to these processors as follows.

$$L_1\alpha_{x-1,1}E_{x-1} = \sum_{p=x-1}^{x} (L_1\alpha_{x,1})C_p + L_1\alpha_{x,1}E_x \tag{A.1}$$

Rearranging the above equation, we obtain,

$$L_1\alpha_{x,1} = L_1\alpha_{x-1,1}\left(\frac{E_{x-1}}{\sum_{p=x-1}^{x} C_p + E_x}\right) \tag{A.2}$$

Note that in the above equation, the $\sum_{p=x-1}^{x} C_p$ accounts for all the communication delays incurred between the processors P_x and P_{x-1}. Similarly,

$$L_1\alpha_{x-2,1}E_{x-2} = \sum_{p=x-2}^{x-1} (L_1(\alpha_{x,1} + \alpha_{x-1,1}))C_p + L_1\alpha_{x-1,1}E_{x-1} \tag{A.3}$$

$$L_1\alpha_{x-1,1} = L_1\alpha_{x-2,1}\left(\frac{E_{x-2}}{\sum_{p=x-1}^{x}(\frac{E_{x-1}}{\sum_{p=x-1}^{x} C_p + E_x} + 1)C_p + E_x - 1}\right) \tag{A.4}$$

Repeating the procedure above, we obtain $L_1\alpha_{i,1}, i = (g+1), ..., (x-1), x$ with respect to $L_1\alpha_{g,1}$. To determine $L_1\alpha_{g,1}$, we have a condition wherein the total communication and processing time is equated to τ_s, where τ_s is the smallest release time among the processors that have $\tau_i \neq 0$.

$$\sum_{p=1}^{g-1}(\sum_{i=g}^{x} L_1\alpha_{i,1})C_p + L_1\alpha_{g,1}E_g = \tau_s \tag{A.5}$$

If there are r processors with release times equal to 0, then with the above approach, we can have r equations with r unknowns. Hence, we will be able to solve for all $L_1\alpha_{i,1}, i = g, (g+1), ..., (x-1), x$.