Divisible Load Scheduling on Heterogeneous Distributed Systems

異種分散システムにおける可分タスクのスケジューリング

July 2008

Graduate School of Global Information and Telecommunication Studies
Waseda University

Audiovisual Information Creation Technology II

Abhay Ghatpande

In memory of my papa

PREFACE

With the proliferation of the Internet, *volunteer computing* is rapidly becoming feasible and gaining popularity. Volunteer computing is a form of distributed computing in which a large number of average users offer their computers to serve as processing and storage resources for scientific research projects or what are known as *grand challenge problems*. Similarly, *grid computing* and *cloud computing*, which provide mechanisms for users and applications to submit and execute computationally intensive workflows on remote resources, are being used for wide variety of applications today.

The jobs that can be submitted to these systems are limited because they use open and shared resources that introduce three important challenges: (a) heterogeneity, (b) uncertainty, and (c) network latency and delay. If an application has a complex structure, then a lot of time is spent waiting for data to be transferred between the participating nodes. This makes *divisible loads* ideally suited for execution on volunteer and grid computing systems. Divisible loads are perfectly parallelizable and can be arbitrarily partitioned into independently and identically processable load fractions.

Divisible loads usually follow the master-slave model of computation: a master node holds the entire (divisible) load and distributes it to the slave nodes; the slave nodes perform requisite processing on the allocated load fractions, and send the results back to the master. This research considers two of the above mentioned issues: (a) the slaves are heterogeneous, i.e., they differ in computation speed and the bandwidth of the network that links them with the master node, and (b) the networks have high latency, i.e., the bandwidths can be considerably lower than the computation speeds.

Under these conditions, the focus of this research is to minimize the *makespan*, i.e., the total time from the point that the master begins sending out load fractions to the point where result collection from all slaves is complete. This involves optimizing (a) the selection of slaves to allocate load, (b) the quantity of load to be allocated to each, (c) the order (sequence) in which the fractions are sent to the slaves, and (d) the order in which the slaves send the results back to the master. This optimization problem is referred to as the DLSRCHETS (Divisible Load Scheduling with Result Collection on HETerogeneous Systems) problem.

Divisible Load Theory (DLT), the mathematical framework for the optimization of Divisible Load Scheduling (DLS) has been studied for over ten years. Most of the work has concentrated on the case where no results are returned to the master. This simplifies the analysis to a great extent, and allows for derivation of optimal load fractions and sequence of distribution to the slaves. The addition of result collection and system heterogeneity breaks down this simplicity. The complexity of DLSRCHETS is an open problem and there is no known polynomial-time algorithm for an optimal solution to DLSRCHETS. Before this research, the only proposed solutions to DLSRCHETS were FIFO (First In, First Out) and LIFO (Last In, First Out) type of schedules, which are not always optimal.

This work considers the most general form of DLSRCHETS. No assumptions are made regarding the number of slaves that are allocated load, both the network and computation speeds of the slaves are considered to be heterogeneous, and idle time can be present in the schedule if it reduces the makespan.

The overall flow of this thesis is as follows. The theoretical basis of DLSRCHETS is first established, and it is defined in terms of a linear program for analysis. The optimal schedule for a system with two slaves is extensively explored because the proposed algorithms are built on it. Two new polynomial-time algorithms, namely ITERLP (ITERative Linear Programming) and SPORT (System Parameters based Optimized Result Transfer) are proposed as solutions to DLSRCHETS. The performance of traditional and new algorithms is compared using a large number of simulations and the proposed algorithms are shown to have superior performance. The thesis is organized into five main chapters, preceded by an introduction, and terminated by a conclusion as described below.

Chapter 1. Introduction establishes the research context that forms the basis for this thesis. It introduces the application areas of volunteer and grid computing, and the problems faced in scheduling applications on these platforms. Next, divisible loads and divisible load scheduling are introduced along with the important results to date. The shortcomings of traditional DLT and the research objectives are laid out. Traditional methods are compared with the new approaches in this thesis, and the contributions of this thesis are elaborated. The organization of the thesis is explained.

Chapter 2. System Model defines the system model upon which this thesis is built. There are several important constraints and assumptions that are used in the problem definition. The first one is that the communication and computation times are linearly increasing functions of the size of data. Similarly, the size of result data generated by a processor is directly proportional to the size of its allocated input load data. The constant of proportionality depends only on the application under consideration and is the same for all the processors. It is assumed that a processor can do only one thing at a time, i.e., communication and computation cannot overlap in a processor. Further, a processor can communicate with only one other processor at a time. This is known as the unidirectional one-port model. All operations including data transmission, reception, and computation follow the atomic or block-based model, i.e., they proceed uninterrupted in a single installment until the end. In this chapter, justification is given for all these assumptions considering the target applications and environment.

Chapter 3. Analysis of DLSRCHETS provides a detailed derivation of the DLSRCHETS problem definition. After first laying the theoretical basis, the DLSRCHETS problem is defined in terms of a linear program. The analysis of the optimal solution to DLSRCHETS is presented. Two important proofs are given — one for the *allocation precedence condition* and the other for the *idle time theorem*. The allocation precedence condition is necessary to limit the number of possible schedules of DLSRCHETS to a finite number. It argues that there always exists an optimal solution to DLSRCHETS in which the entire load is first distributed to the slaves before the master starts to receive results from the slaves. The proof uses rearrangement of the timing diagram to substantiate the claim.

The proof of the idle time theorem is more complicated as it uses the geometry of linear programming. A brief introduction to linear programming is included in the chapter for this reason. The idle time theorem states that not all slaves may be allocated load in the optimal solution, and irrespective of the number of slaves that are allocated load, at most one slave can have idle time in the optimal solution. In linear programming, some solutions can be degenerate. Analysis proves that the idle time theorem is true for both non-degenerate and degenerate cases.

Chapter 4. The ITERLP Algorithm proposes the new polynomial-time ITERLP algorithm. The complexity of DLSRCHETS is an open problem and finding the optimal solution is difficult. Thus, one has to resort to heuristic algorithms under the circumstances. The proposed ITERLP algorithm reduces the number of possible allocation and collection sequences to m each instead of the usual $m!$. The rationale behind the *pruning* of possible schedules in ITERLP is explained.

The computation cost of ITERLP is still quite high — in the worst case $O(m^3)$ linear programs have to be solved. The simulations show that ITERLP performance is much better than LIFO and FIFO over a wide range of parameter values. The performance of the algorithm is quite stable; schedules generated by ITERLP have execution time close to the optimal in most of the cases. In the extensive simulations performed, the maximum deviation of processing time with respect to the optimal is 0.8% for 5 processors, and it takes about 3 to 5 minutes to find the schedule. As the number of processors increase, the time required to compute the solution increases. For example, it takes around 80 minutes to compute the ITERLP schedule for 65 processors. Because the expected error is low, even though computation cost is high, ITERLP allows comparison of other heuristic algorithms when it is impractical to find the optimal solution.

Chapter 5. The Two-Slave System lays the foundation of the two-slave system that forms the basis for the SPORT algorithm. Several important concepts are introduced in this chapter. It begins with the three types of possible optimal schedules in a two-slave system and the related derivations. This is followed by the derivation of the optimal schedule for two processors using simple if-then-else clauses. This derivation includes two important results: (a) the condition for optimality of the LIFO and FIFO schedules, which shows that whether LIFO (resp. FIFO) is faster for a two-slave system depends only on the communication speeds of the links, and (b) the condition for the existence of idle time in a FIFO schedule that shows a relationship between the presence of idle time and the computation and communication speeds of the two processors, and the type of divisible load under consideration. Next, the equivalent processor for LIFO and FIFO schedules in a two-slave system is derived. The equivalent processor enables the combination of two processors into a single virtual processor. The equivalent processor concept is extended to an arbitrary number of processors, and its applications are explained. A method to determine the number of processors to allocate load is derived using the equivalent processor concept.

Chapter 6. The SPORT Algorithm introduces the SPORT algorithm as a solution to the DLSRCHETS problem. Along with the allocation and collection sequences, the SPORT algorithm finds: (a) the number of processors to use for computation, and (b) the load fractions to be allocated to the participants. The important point is that this is done without solving time-consuming linear programs. The number of possible allocation and collection sequences is limited to a few, potentially optimal permutations. Because of this, with a set of processors sorted by decreasing communication speed, the complexity of SPORT is $O(m)$, where m is the number of available processors. The algorithm is robust to system composition and it provides good schedules for both homogeneous and heterogeneous types of systems. In the large number of simulations performed, the maximum deviation of processing time with respect to optimal is 1.5% for 5 processors. SPORT is very fast — it takes less than a second to find the solution for 500 processors.

The basic idea behind SPORT is very simple — to use two processors at a time and build a piecewise locally optimal schedule. However it is not very straightforward to be able to do this directly, and several necessary tools are designed. Detailed explanation regarding the working of the algorithm is given. The method of deriving load fractions using binary tree traversal is explained. Results of the comprehensive simulation testing of the algorithms are presented.

Chapter 7. Conclusion summarizes the various points covered in the thesis and presents several ideas for future work. It is proposed that future work can proceed in the following main directions: (a) Theoretical analysis of complexity and other optimality results, (b) Extensions to the current system model, (c) Modifying the nature of DLSRCHETS itself, and (d) Development of applications and physical testing.

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisor Prof. Hiroshi Watanabe, and co-advisors, Prof. Hidenori Nakazato and Prof. Olivier Beaumont. Without their guidance and encouragement, I would have made no progress. I thank Dean Yoshiyori Urano and Prof. Wataru Kameyama, for being on the jury, and for their questions and comments on the dissertation.

I am deeply indebted to several people who helped me over the years. Some of them are Prof. Thomas Robertazzi, Prof. Veeravalli Bharadwaj, Prof. Maciej Drozdowski, Prof. Gerassimos Barlas, Prof. Jeff Erickson, Prof. Akihisa Kodate, Dr. Henning Breuer, and Dr. Jim Dowling.

My thanks to all my friends and colleagues who have enriched my life in ways too numerous to mention.

And finally, a big thank you to my family — my greatest asset. I am fortunate to be surrounded by such wonderful, supportive people.

TABLE OF CONTENTS

Preface **i**

List of Figures **xii**

List of Tables **xiii**

1 Introduction **1**

1.1 Research Context 1

1.1.1 Volunteer and Grid Computing 1

1.1.2 Divisible Loads and Scheduling 3

1.1.3 Shortcomings of Traditional DLT 6

1.2 Research Objectives 9

1.3 Thesis Contributions 10

1.4 Thesis Organization 13

2 The System Model **17**

2.1 Introduction 17

2.2 Job Execution Model 17

2.3 Communication and Computation Model 18

2.4 Communication and Computation Parameters 20

2.5 Result Data Model 21

2.6 Communication and Computation Time 21

2.7 Summary 23

3 Analysis of DLSRCHETS **25**

3.1 Introduction 25

3.2 Problem Formulation 25

3.3 A Primer on Linear Programming 33

3.3.1 General Linear Programming Problem 33

3.3.2 Geometry of Linear Programming 34

3.3.3 Bases, Feasibility, and Local Optimality 36
3.4 Analysis of Optimal Solution 36
3.5 The Importance of Idle Time 43
3.6 Summary 45

4 The ITERLP Algorithm 47

4.1 Introduction 47
4.2 Brief Introduction to Permutations 47
4.3 Proposed Algorithm 50
4.4 Algorithm Explanation 51
4.5 Finding Candidate Sequences 52
4.6 Complexity and Discussion 53
4.7 Simulation Results and Analysis 54
4.7.1 Algorithm Variants for Comparison 55
4.7.2 Simulation Method 56
4.7.3 Result Plots and Analysis 57
4.8 Summary 60

5 Two-Slave System 63

5.1 Introduction 63
5.2 Two-Slave System Configuration 64
5.3 Schedule f 67
5.4 Schedule l 69
5.5 Schedule g 72
5.6 Optimal Schedule in Two-Slave System 74
5.7 The Concept of Equivalent Processor 76
5.8 Equivalent Processor for Schedule f 77
5.9 Equivalent Processor for Schedule l 78
5.10 Equivalent Processor for Schedule g 79
5.11 The Equivalent Processor Theorem 79
5.12 Extending the Equivalent Processor Concept 82
5.12.1 The General FIFO Schedule 83
5.12.2 General FIFO Equivalent Processor 84
5.12.3 The General LIFO Schedule 86
5.12.4 General LIFO Equivalent Processor 87
5.12.5 Results Using General Equivalent Processor 88
5.13 Summary 89

6 The SPORT Algorithm **91**
6.1 Introduction 91
6.2 Proposed Algorithm 92
6.3 Algorithm Explanation 95
6.4 Complexity and Discussion 99
6.5 Simulation Set A 100
6.5.1 Set A1 — $\mathcal{C}$ homogeneous, $\mathcal{E}$ homogeneous 101
6.5.2 Set A2 — $\mathcal{C}$ homogeneous, $\mathcal{E}$ heterogeneous 103
6.5.3 Set A3 — $\mathcal{C}$ heterogeneous, $\mathcal{E}$ homogeneous 105
6.5.4 Set A4 — $\mathcal{C}$ heterogeneous, $\mathcal{E}$ heterogeneous 106
6.5.5 Simulation Result Analysis 108
6.6 Simulation Set B 111
6.6.1 Simulation Method 111
6.6.2 Simulation Results and Analysis 111
6.6.3 Discussion on Performance of LIFOC and SPORT 113
6.7 Summary 115

7 Conclusion **117**
7.1 Summary of the Thesis 117
7.2 Ideas for Future Work 120
7.2.1 Theoretical Analysis 120
7.2.2 Extending the System Model 120
7.2.3 Modification of DLSRCHETS 120
7.2.4 Application Development 121

References **123**

List of Publications **131**

Acronyms and Abbreviations **133**

List of Symbols **135**

LIST OF FIGURES

1.1 A heterogeneous master-slave system 3
1.2 Divisible load partitioning 5
1.3 The timing diagram for AFS policy 6
1.4 Timing diagram for FIFO schedule 7
1.5 Timing diagram for LIFO schedule 8
1.6 A possible schedule in this research 10

2.1 A general schedule for DLSRCHETS 18
2.2 The heterogeneous master-slave system $\mathcal{H}$ 21

3.1 A possible schedule with $m = 3$ 27
3.2 Interleaved result collection 28
3.3 A schedule that satisfies the Feasible Schedule Theorem 31
3.4 Timing diagram for the optimal LIFO schedule 40
3.5 The rearranged optimal LIFO schedule 41
3.6 Timing diagram for an optimal FIFO schedule 42
3.7 An optimal FIFO schedule with idle time 42
3.8 Optimal LIFO schedule for Example 3.1 44
3.9 Optimal FIFO schedule with idle time in both processors 45
3.10 Optimal FIFO schedule with idle time in one processor 45

4.1 A feasible schedule for $m = 3$ 49
4.2 ITERLP progress illustrated for 4 processors 51
4.3 Normalized Execution Time for $m = 4$, $\delta = 0.2$, Case 9 57
4.4 $\langle \Delta T \rangle$ for $m = 4$, $\delta = 0.2$ 58
4.5 $\langle \Delta T \rangle$ for $m = 5$, $\delta = 0.8$ 59
4.6 $\langle \Delta T \rangle$ for $m = 5$, $\delta = 0.5$ 59
4.7 $\langle \Delta T \rangle$ w.r.t. ITERLP, $m = 10$, $\delta = 0.2$ 60
4.8 $\langle \Delta T \rangle$ w.r.t. ITERLP, $m = 20$, $\delta = 0.5$ 61
4.9 $\langle \Delta T \rangle$ w.r.t. ITERLP, $m = 30$, $\delta = 0.8$ 61

4.10 $\langle \Delta T \rangle$ w.r.t. ITERLP, $m = 30$, $\delta = 0.2$ 62

5.1 A feasible schedule for $m = 3$ 65
5.2 A heterogeneous two-slave system 66
5.3 Timing diagram for Schedule f 67
5.4 Timing diagram for Schedule l 69
5.5 Timing diagram for Schedule g 72
5.6 The concept of equivalent processor 76
5.7 Equivalent processor in Schedule f 78
5.8 Equivalent processor in Schedule l 79
5.9 Equivalent processor in Schedule g 80
5.10 General FIFO schedule and equivalent processor 83
5.11 General LIFO schedule and equivalent processor 86

6.1 The SPORT Algorithm 96
6.2 An example of SPORT for three processors 97
6.3 Calculating the load fractions in SPORT 98
6.4 The main and sub-intervals used for Simulation Set A 102
6.5 $\langle \overline{\Delta T} \rangle$ in Set A1 for $m = 4$, $\delta = 0.2$ 103
6.6 $\langle \overline{\Delta T} \rangle$ in Set A1 for $m = 5$, $\delta = 0.8$ 104
6.7 $\langle \overline{\Delta T} \rangle$ in Set A2 for $m = 4$, $\delta = 0.5$ 105
6.8 $\langle \overline{\Delta T} \rangle$ in Set A2 for $m = 5$, $\delta = 0.2$ 105
6.9 $\langle \overline{\Delta T} \rangle$ in Set A3 for $m = 4$, $\delta = 0.8$ 106
6.10 $\langle \overline{\Delta T} \rangle$ in Set A3 for $m = 5$, $\delta = 0.5$ 107
6.11 $\langle \overline{\Delta T} \rangle$ in Set A4 for $m = 4$ 108
6.12 $\langle \overline{\Delta T} \rangle$ in Set A4 for $m = 5$ 108
6.13 $\overline{\Delta T}$ in simulation set B, $m = 10$ 112
6.14 $\overline{\Delta T}$ in simulation set B, $m = 100$ 113
6.15 $\overline{\Delta T}$ in simulation set B, $m = 300$ 114
6.16 Comparison of wall-clock time for SPORT, LIFOC, and FIFOC 116

LIST OF TABLES

1.1 Comparison with traditional DLS without result collection 11
1.2 Comparison with traditional DLS with result collection 11

4.1 Results for $\mathcal{C} = \{10, 15\}, \mathcal{E} = \{10, 10\}, \delta = 0.5$ 53
4.2 Results for $\mathcal{C} = \{10, 15, 20\}, \mathcal{E} = \{10, 10, 1\}, \delta = 0.5$ 54
4.3 Parameters for ITERLP simulations 56
4.4 Maximum $\langle \Delta T \rangle$ of FIFOC 58
4.5 Maximum $\langle \Delta T \rangle$ of LIFOC 58
4.6 Maximum $\langle \Delta T \rangle$ of ITERLP 60

6.1 Parameters for SPORT simulation sets A and B 101
6.2 Minimum statistics for SPORT simulation set A 109
6.3 Maximum statistics for SPORT simulation set A 110
6.4 Statistics for LIFOC in simulation set B 115
6.5 Comparison of wall-clock time for SPORT, LIFOC, and FIFOC 115

7.1 Algorithm feature and performance comparison 118

CHAPTER 1

INTRODUCTION

1.1 RESEARCH CONTEXT

Parallel and distributed computing has been a topic of active research for over 50 years. The basic idea behind parallel computing is very simple — the simultaneous use of multiple computing elements to solve a computational problem. The primary reasons for using parallel computing are:

- To solve problems faster, i.e., save time
- To solve larger problems
- To solve several problems at the same time

There are other secondary reasons, such as economic benefits, technology limitations on serial computing, memory constraints of single computers, and availability of cheap, off-the-shelf (COTS) hardware.

The computing elements in parallel computing may be a single computer with multiple processors (these days *multiple cores* in a single CPU are also common), or several computers connected by a network, or usually a combination of both. The application under consideration decides the choice of architecture, i.e., the level at which parallelism can be most efficiently exploited as dictated by the algorithm. In this thesis, the focus is on *wide area distributed computing*, in which the computers are *loosely-coupled* and geographically dispersed. Parallelism follows the *data-parallel* model, and individual computers independently run entire programs on their share of the data.

1.1.1 Volunteer and Grid Computing

With the proliferation of the Internet, *volunteer computing* [3–5] is rapidly becoming feasible and gaining popularity. Volunteer computing is a form of distributed computing in which a large number of average users offer their computers to serve as processing and storage resources for scientific research projects or what are known as *Grand Challenge Problems* such as [93]:

- astronomy and astrophysics,
- biological, human genome,
- chemical and nuclear reactions,
- cryptography,
- geological, seismic activity,
- weather and climate study.

Similarly, *grid computing* [43, 44] and *cloud computing* [72, 96], which are new distributed computing paradigms, provide mechanisms for users and applications to submit and execute computationally intensive workflows on remote resources. Grids are being used for a large number of applications today such as:

- astronomy & space exploration [50, 90],
- distributed video capture, storage and retrieval [31],
- distributed image processing [80, 89],
- polygon rendering for simulation and visualization [58, 67],
- satellite data processing [91],
- computer vision [89],
- medical simulation [19], and many more.

All these systems essentially follow the *master-slave* approach shown in Fig. 1.1. It is a star connected (single-level tree) network where the center of the star (root of the tree) forms the master and the points of the star (leaf nodes of the tree) form the slaves. The master holds all the data associated with a job (problem) that has to be processed in some manner. The data is then divided into a number of parts, and distributed to the slaves. The slaves perform the requisite computation on their respective parts in parallel, and return the computed results back to the master.

Because of the use of open and shared resources, one of the biggest problems faced by these new forms of distributed computing is the network latency and delay. This places a

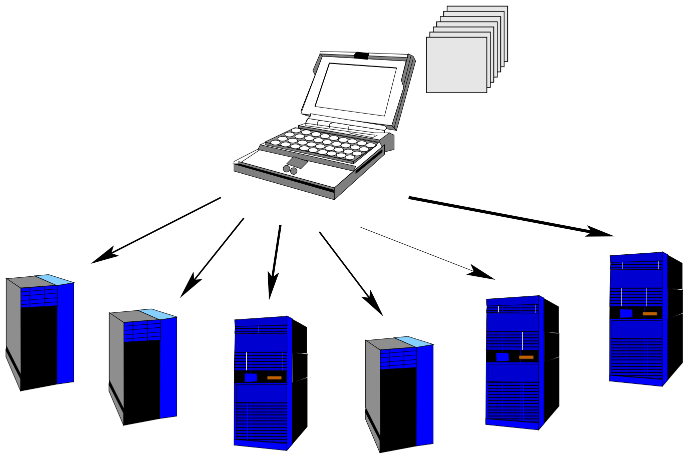


Figure 1.1 A heterogeneous master-slave system. Slaves can have different computing speeds and the network links connecting them to the master can have different bandwidths.

limitation on the types of jobs that can be submitted to these systems, if the systems are to be used efficiently. If an application has complex structure, then a lot of time is spent just waiting for data to be transferred between the participating nodes.

Volunteer computing uses anonymous contributed resources, and a large degree of heterogeneity exists in the network bandwidth and processing power of the participating nodes. In the case of grid or cloud computing, usually the resources are contributed by institutions, but here too, the resources are highly heterogenenous and shared among many users.

The geographical distribution of computing capabilities and the resource heterogeneity requires novel approaches to access, transfer, and process data. The basic technology for exploiting computational grids is grid middleware, which is provided by toolkits such as Globus [43], Legion [51], and UNICORE [76], and for volunteer computing by software such as BOINC [2]. These middleware toolkits offer information infrastructure services for security, communication, fault detection, resource and data management, and portability. The availability of these basic services permits the focus on algorithms for improving the utilization and efficiency of the system without worrying about *how* the implementation will be carried out.

1.1.2 Divisible Loads and Scheduling

Scheduling has been an important area of study for a long time. Scheduling is one of the main areas of contemporary mathematics as a branch of combinatorial optimization [87]. The origin of scheduling lies in the operations research mainly in production and project management [6, 30, 40]. As the complexity of computer systems grew, the results from these areas were applied to the management and control of computer systems. Scheduling is now an important part of the design of libraries and compilers [7, 71, 83], operating systems [41, 88], and real-time systems [53, 79, 81, 86, 102]. Scheduling theory and its range of theoretical and practical results is too vast to be presented here. There is however one branch of scheduling theory that has gained wide importance over the last few years and that has been studied extensively for its simplicity and tractability — *Divisible Load Theory* (DLT) and *Divisible Load Scheduling* (DLS).

Divisible loads are a special class of parallelizable applications that have very regular linear structure, and which if given a large enough volume, can be *arbitrarily* partitioned into any number of independently and identically processable *load fractions* (parts). In Fig. 1.2, each light gray square represents a unit task and the set of 100 such squares represents the divisible load. As seen in Fig. 1.2, the divisible load (job) can be partitioned in any number of ways. Each part undergoes the same processing as the others, and can be processed independently of the others. The shape of a part is not important, and the partitioning of a divisible load should not be confused with a 2-dimensional cutting problem [29]. Figure 1.2 just illustrates that a divisible load is composed of independent tasks that can be clustered together arbitrarily.

Examples of applications that satisfy this divisibility property include massive dataset processing, image processing and rendering, signal processing, computation of Hough transforms, tree and database search, Monte Carlo simulations, computational fluid dynamics, and matrix computations. Divisible loads are especially suited for execution on volunteer, grid and cloud computing systems because of the absence of interdependencies and precedence relations between the different parts into which the load is divided.

The partitioning of a divisible load, the allocation (mapping) of the parts to appropriate processors for execution, and the sequencing (ordering) of the transfer of the parts to and from the processors, is together known as Divisible Load Scheduling. Divisible Load Theory is the mathematical framework that has been established to study the optimization of DLS [9, 11–13, 22–27, 34, 36, 48, 49, 56, 62, 63, 65, 73–75, 84, 85].

The hallmark of DLT has been its relative simplicity and deterministic nature. The basic principle of DLT to determine an optimal schedule for the master-slave system in Fig: 1.1 is the AFS (All slaves Finish Simultaneously) policy [9]. This states that the optimal schedule is one in which the load is distributed such that all the nodes involved in the computation finish processing their individual load fractions at the same time [52]. It is illustrated in Fig. 1.3 that shows the timing diagram of the slaves' communication and computation. The time spent in communication with the master is shown above the horizontal axes, and the time spent in computation by the individual slaves is shown below

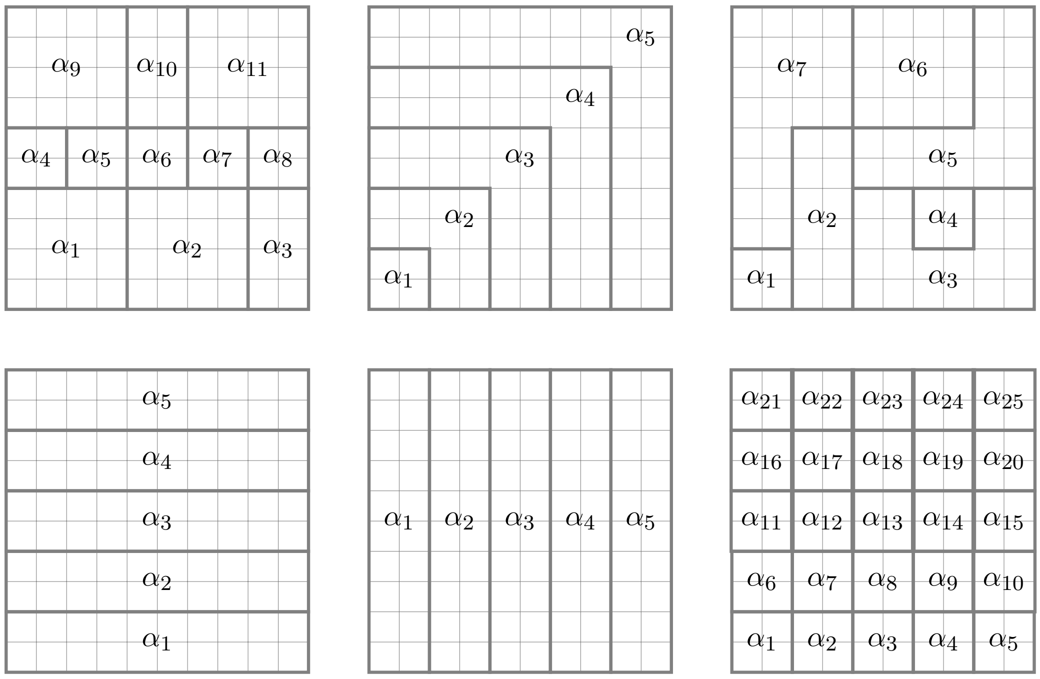


Figure 1.2 The data set associated with a divisible load (job) can be arbitrarily partitioned in any number of ways. Each part undergoes the same processing as the others, and can be processed independently of the others. The shape of the part is not important. The figure just illustrates that a divisible load is composed of independent tasks that can be clustered together arbitrarily.

the horizontal axes. This convention is followed throughout this thesis. As can be seen, all the slaves finish their computation at time T.

Blazewicz and Drozdowski [28] revised the traditional system model with overhead factors, while Bharadwaj et al. [26] extended the study with closed-form solutions for optimal processing time. Sohn et al. [85] used the DLT to minimize total monetary cost when utilizing system resources. There are several papers that have tackled other implementation related issues. For example, Li et al. [63] dealt with finite size buffer constraints which are very important when considering shared resources. Another issue is the granularity of the input load, i.e., the fact that practical loads are not infinitely divisible but have a certain minimum size. Bharadwaj and Viswanadham [21] analyzed bus networks by introducing a *divisibility factor* and dividing the load in integral multiples of the factor. Several other issues such as processor release times, multi-installment load distribution, fault-tolerance, and start-up costs have been addressed in [23, 101, 104]. Some other important papers in the DLT area are [9, 11–13, 22, 23, 25, 26, 34, 36, 48, 49, 56, 62, 65, 74, 84, 85]. The definitive reference for this field is the book by Bharadwaj et al. [24], while Drozdowski [39] offers some excellent insights into the practical implementation of DLT on various

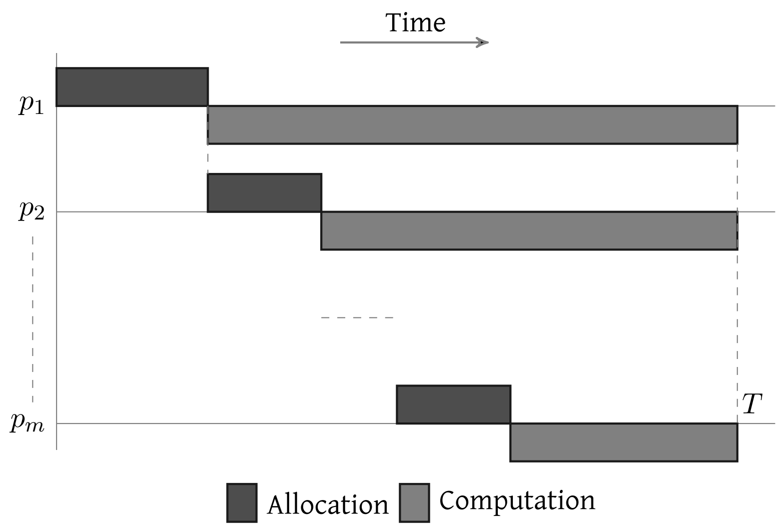


Figure 1.3 The timing diagram for AFS policy. All processors finish their computation at the same time T. This is the optimal schedule when there is no result collection phase.

architectures. Bharadwaj et al. [27] and Beaumont et al. [15] recently published reviews of the work done to date in DLT. An exhaustive listing of papers regarding DLT and DLS is available on Thomas Robertazzi's homepage [73].

1.1.3 Shortcomings of Traditional DLT

The AFS policy yields closed-form equations to determine the optimal load fractions to be allocated to the processors, and allows easy theoretical analysis of the system performance. The AFS policy implies that after the nodes finish computing their individual load fractions, no results are returned to the source. This is an unrealistic assumption for most applications on volunteer and grid computing, as the result collection phase contributes significantly to the total execution time. This is a serious shortcoming of traditional DLT.

Along with the AFS policy, there are two assumptions that have implicitly pervaded DLT literature to date:

(a) Load is allocated to *all* available processors (slaves), and

(b) A processor is never idle except for the time when it is waiting for the reception of its allocated load fraction from the master.

The presence of idle time in the optimal schedule has been overlooked in DLT work on result collection and heterogeneity. It is a very important issue because it may sometimes be possible to improve a schedule by inserting idle time.

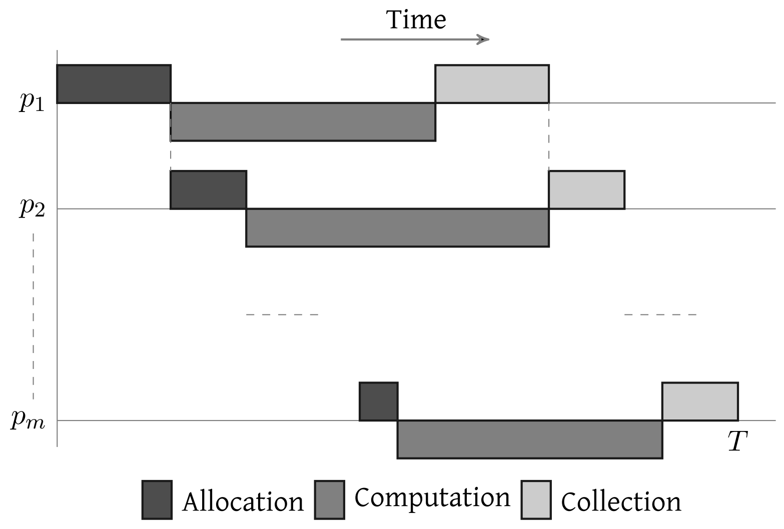


Figure 1.4 Timing diagram for FIFO schedule. The results are collected in the same sequence as the load fractions are allocated. Though not shown in this figure, optimal FIFO schedules may have idle time.

A few papers have dealt with DLS on heterogeneous systems to date [16–18, 24, 36, 77]. Bharadwaj et al. [24], Chap. 5 proved that the sequence of allocation of data to the processors is important in heterogeneous networks. Without considering result collection, they proved that for optimum performance,

(a) when processors have equal computation capacity, the optimal schedule results when the fractions are allocated in the order of decreasing communication link capacity, and

(b) when communication capacity is equal, the data should be allocated in the order of decreasing computation capacity.

As far as can be judged, no paper has given a satisfactory solution to the scheduling problem where both the network bandwidth and computation capacities of the slaves are different, and the result transfer to the master is explicitly considered.

Cheng and Robertazzi [34] and Bharadwaj et al. [24], Chap. 3 addressed the issue of result collection with a simplistic constant result collection time, which is possible only for a limited number of applications on homogeneous networks. All other papers that have addressed result collection to date, advocated FIFO (First In, First Out) and LIFO (Last In, First Out) type of schedules. As shown in Fig. 1.4, in FIFO, results are collected in the same order as that of load allocation, while in LIFO, the order of result collection is reversed as shown inFig. 1.5. Barlas [9] addressed the result collection phase for single-level and

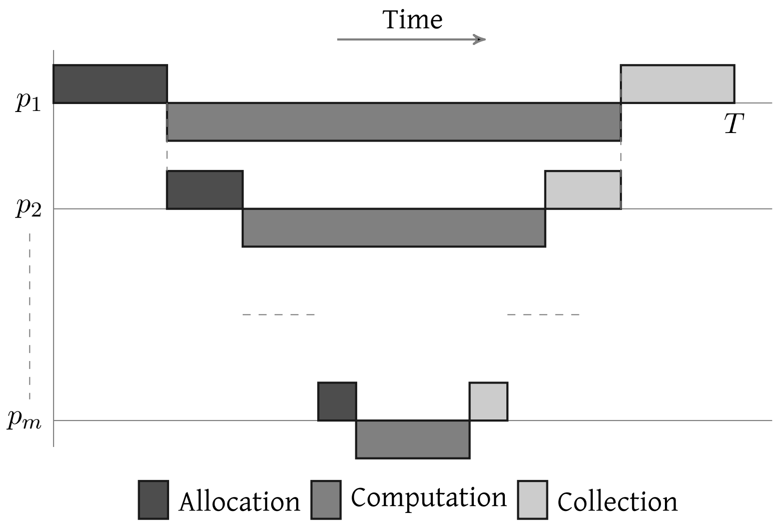


Figure 1.5 Timing diagram for LIFO schedule. The results are collected in the reverse sequence as allocation of load fractions. In the optimal LIFO schedule, no processor has idle time.

arbitrary tree networks, but an assumption regarding the absence of idle time was made without justification. Essentially he analyzed only two cases:

(a) when communication time is zero, and

(b) when communication networks are homogeneous.

The optimal sequences derived were essentially LIFO or FIFO. Rosenberg [77] too proposed the LIFO and FIFO sequences for result collection. He concluded through simulations that FIFO is better when the communication network is homogeneous with a large number of processors, while LIFO is advantageous when the network is heterogeneous with a small number of processors.

For the first time, it was shown in [17] that the LIFO and FIFO orderings are not always optimal for a given set of processors. In [16, 18], it was proved that all processors from a given set of processors may not be used in the optimal solution. For the unidirectional single-port communication model (see Chapter 2), [16–18] proved the following features in optimal schedules:

- In optimal LIFO and FIFO schedules, load is allocated in the order of decreasing communication link bandwidth.
- In the optimal LIFO schedule, no processor has idle time.
- There exists an optimal FIFO schedule in which at most one processor may have idle time.

- If there exists a processor with idle time in an optimal FIFO schedule, then it can always be chosen to be the last processor in the allocation sequence (i.e. the processor with the slowest communication link).

The above optimality results have been derived strictly for LIFO and FIFO type of schedules, and are not applicable for the general case considered in this thesis.

1.2 RESEARCH OBJECTIVES

The focus of this research is to study the scheduling of divisible loads on heterogeneous master-slave systems when the slaves return result data to the master. Specifically, it seeks to minimize the *makespan* of such a schedule, i.e., to minimize the total time from the point that the master begins sending out load fractions to the point where result collection from all slaves is complete. This involves optimizing

(a) the selection of slaves to allocate load,

(b) the quantity of load to be allocated to each,

(c) the order (sequence) in which the fractions are sent to the slaves, and

(d) the order in which the slaves send the results back to the master.

This optimization problem is called the Divisible Load Scheduling with Result Collection on HETerogeneous Systems (DLSRCHETS) problem, and it is formally defined and analyzed in detail in this thesis. The most general form of DLSRCHETS is considered. No assumptions are made regarding the number of slaves that are allocated load, both the network and computation speeds of the slaves are assumed to be heterogeneous, and idle time can be present in the schedule if it reduces the makespan.

Finding an optimal solution to DLSRCHETS is surprisingly difficult. In fact, the complexity of DLSRCHETS is an open problem and there is no known polynomial-time algorithm for an optimal solution to DLSRCHETS. Thus it is important to find some characteristics of DLSRCHETS and to gain insight into the optimal solution. Some of the questions that are addressed in this thesis are:

- How can the general DLSRCHETS problem be defined mathematically? What are the necessary conditions to be able to do that?
- Should all slaves be allocated load first before they start sending results back to the source in the optimal solution?
- Will all available slaves be allocated load in the optimal solution?

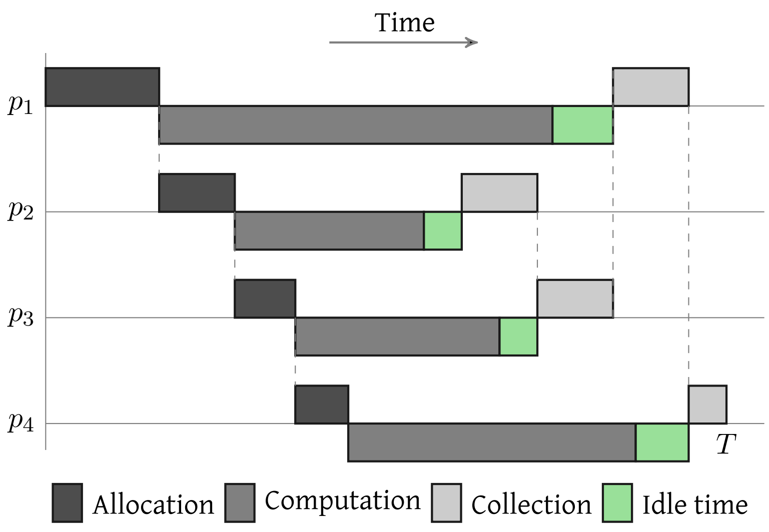


Figure 1.6 A possible schedule in this research. Allocation and collection sequences can be arbitrary. Idle time may be present in all participating processors if it helps reduce the total processing time T.

- How many slaves that are allocated load will have idle time in the optimal solution?
- Is it possible to identify some relationship between system computation and communication speeds and the optimal solution? Under what circumstances is a particular schedule optimal? What causes it to be optimal?
- If finding an optimal solution is difficult, is it possible to find some near-optimal heuristic algorithms to solve DLSRCHETS?

Tables 1.1 and 1.2 summarize the differences between traditional divisible load scheduling and this research.

1.3 THESIS CONTRIBUTIONS

Several original and unique contributions resulted from the work on this thesis:

The *Allocation Precedence Lemma* (Chapter 3) The allocation precedence condition states that, *the master distributes load to all participating slaves first, before receiving any results*. The allocation precedence lemma proves that in the general case considered in this thesis, there always exists an optimal schedule that satisfies the allocation precedence condition. This is necessary to limit the range of optimal solutions to DLSRCHETS to a finite number.

The *Idle Time Theorem* (Chapter 3) A proof is given for the Idle Time Theorem, which states that, *there exists an optimal solution for DLSRCHETS in which, irrespective of whether load*

Table 1.1 Salient features of this research as compared to traditional DLS without result collection. The theorems in traditional DLS are not applicable to this research.

DLS without result collection	This Research
Ignores result collection.	Explicitly schedules result collection phase.
Theorem Optimal schedule uses all available processors.	Selects only necessary number of processors.
Theorem Optimal allocation sequence is in order of decreasing communication bandwidth irrespective of computation speed.	Optimal sequence depends on both communication and computation speeds; it cannot be predefined.
Theorem No processor has idle time in the optimal schedule.	**Theorem** There is an optimal solution where only one processor may have idle time.

Table 1.2 Salient features of this research as compared to traditional DLS with result collection. This research is distinctly more general in nature. The theorems for LIFO and FIFO are not applicable here.

DLS with result collection	This Research
Considers only LIFO or FIFO	Considers completely general schedule
Theorem Optimal schedule uses all available processors.	Selects only necessary number of processors.
Theorem Optimal allocation sequence is in order of decreasing communication bandwidth irrespective of computation speed.	Optimal order depends on both communication & computation speeds; it cannot be predefined.
Usually ignores idle time. Recently considered idle time in FIFO.	Considers idle time to reduce makespan in some cases.
Theorem There is an optimal FIFO schedule where only one processor may have idle time.	**Theorem** There is an optimal schedule where only one processor may have idle time.

is allocated to all available slaves, at most one of the slaves allocated load has idle time, and that the idle time exists only when the result collection begins immediately after the completion of load distribution. This is one of the principal contributions of this thesis. First, because it shows that in some cases insertion of idle time can be beneficial, and second, because it enables the definition of a constraint on the number of processors to be used in the SPORT algorithm.

The ITERLP Algorithm (Chapter 4) The new ITERLP (ITERative Linear Programming) algorithm is proposed and found to be near-optimal after rigorous testing. The ITERLP algorithm does not necessarily use all processors (slaves) and determines the number of processors to be used by repeatedly solving a number of linear programs. The complexity of ITERLP is polynomial in the number of slaves (m) and requires solving $O(m^3)$ linear programs in the worst case. Though the algorithm is computationally too expensive to be used for a large number of slaves, nevertheless it can be used as a benchmark to compare other heuristic algorithms when obtaining the optimal solution is impractical.

Condition for Idle Time (Chapter 5) The idle time theorem proves that under some conditions, idle time may be present in a single processor, but does not specify when the idle time will be present, i.e., under what conditions of the processor communication and computation speeds does it occur. For the first time in DLT, the condition to identify the presence of idle time in a FIFO schedule for two slaves is derived. It has already been proved that there can never be idle time in a LIFO schedule. What is surprising is the simplicity of the condition, and how it is related not only to the communication and computation speeds, but also to the particular divisible load under consideration, specifically to the ratio of size of the result data to the size of input data.

Condition for Optimality (Chapter 5) The identification of the limiting condition for the optimality of the FIFO and LIFO schedules for two processors is a significant addition to DLT. This condition shows that even though the presence of idle time depends on the divisible load under consideration, whether LIFO or FIFO is optimal in a two-slave system depends *only on* the communication speeds of the two processors, and the computation speeds do not matter. This condition supports the conclusions drawn by Rosenberg [77] regarding the performance of LIFO and FIFO.

The concept of *equivalent processor* (Chapter 5) The equivalent processor concept was used by Bharadwaj et al. [24] to prove a number of results in traditional DLT. It is introduced here for the first time in divisible load scheduling for heterogeneous systems with result collection. The equivalent processor is used to summarize the total processing capacity of a pair of slaves. It enables derivation of a piecewise locally optimal solution to DLSRCHETS by combining two processors into one (virtual) processor at a time.

The SPORT Algorithm (Chapter 6) The polynomial-time heuristic algorithm SPORT (System Parameters based Optimized Result Transfer) is another principal contribution of this thesis. The algorithm gives near-optimal solutions to DLSRCHETS and is robust to system heterogeneity. The SPORT algorithm does not necessarily use all processors and determines the number of processors to be used based on the system parameters (computation and communication capacities). SPORT simultaneously finds the sequence of load allocation and result collection, and the load fractions to be allocated to the processors. Given m processors sorted in the order of decreasing network link bandwidth, the complexity of SPORT is of the order $O(m)$, which is a huge improvement over ITERLP. It is rigorously tested using simulations and its performance is found to be only slightly worse than that of ITERLP.

1.4 THESIS ORGANIZATION

This thesis is organized into five main chapters, preceded by an introduction, and terminated by a conclusion as described below.

Chapter 1. Introduction establishes the research context that forms the basis for this thesis. It introduces the application areas of volunteer computing and grid computing, and the problems faced in scheduling applications on these platforms. Next, divisible loads and divisible load scheduling are introduced along with some important results to date. The shortcomings of traditional DLT are enumerated and the research objectives are laid out. The traditional methods are compared with the new approaches in this thesis, and the contributions of this thesis are elaborated. The organization of the thesis is explained.

Chapter 2. System Model defines the system model upon which this thesis is built. It explains the various choices made to represent the communication and computation speeds, the model used for size of result data, the assumptions and reasons regarding continuous delivery of data, the unidirectional one-port communication model, and the decision to use linear models of computation and communication time. Models are ultimately approximations of the real systems and are necessary to be able to analyze the system performance without getting caught up in the finer details. It is important to strike the right balance between abstraction and accuracy.

The model proposed is appealing in its simplicity and at the same time, complete in its coverage. The assumptions are well justified for the applications and the environment targeted in the thesis — divisible loads on heterogeneous master-slave platforms. The most important constraints are:

- Communication and computation time is a linearly increasing function of the size of data.
- Size of result data is proportional to the size of allocated input load data.

- Communication and computation cannot overlap — a processor can do only one thing at a time.
- A processor can communicate with only one other processor at a time — the unidirectional one-port model.
- Data transmission and reception occurs in a single installment — non-preemptive or atomic or block-based model.

Chapter 3. Analysis of DLSRCHETS provides a detailed derivation of the DLSRCHETS problem definition. After first laying the theoretical basis, the DLSRCHETS problem is defined in terms of a linear program.

This chapter primarily presents the analysis of the optimal solution to DLSRCHETS. Two important proofs are given — one for the allocation precedence condition and the other for the idle time theorem. The allocation precedence condition is necessary to limit the number of possible schedules of DLSRCHETS to a finite number. It argues that there always exists an optimal solution to DLSRCHETS in which the entire load is first distributed to the slaves before the master starts to receive results from the slaves. The proof uses rearrangement of the timing diagram to prove the claim.

The proof of the idle time theorem is a bit more complicated. It uses the geometry of linear programming. A brief introduction to linear programming is also included in the chapter for this reason. The idle time theorem makes a very interesting claim — that not all slaves may be allocated load in the optimal solution, and irrespective of the number of slaves that are allocated load, at most one slave can have idle time in the optimal solution.

The assumption that all processors are allocated load can greatly simplify analysis, but it is not realistic. Instead of making this assumption without justification, the case when all processors are not assumed to be allocated load in the optimal solution is considered. The analysis is not so simple in this case. In linear programming, there is a possibility of some solutions being degenerate. Hence the analysis is carried out for both non-degenerate and degenerate cases. It is proved that the idle time theorem is true for both cases.

Chapter 4. The ITERLP Algorithm proposes the new polynomial-time ITERLP algorithm. The complexity of DLSRCHETS is an open problem and finding the optimal solution is difficult. Thus, one has to resort to heuristic algorithms under the circumstances. The logical approach to solving a combinatorial optimization problem by approximation is *pruning*. That is, to find some criterion that can be used to reduce the number of possible output combinations. The proposed ITERLP algorithm reduces the number of possible allocation and collection sequences to m each instead of the usual $m!$. The rationale behind the pruning of possible schedules in ITERLP is explained.

The computation cost of ITERLP is still quite high — in the worst case $O(m^3)$ linear programs have to be solved. The simulations show that ITERLP performance is much better than LIFO and FIFO over a wide range of parameter values. The performance of the algorithm is quite stable; schedules generated by ITERLP have execution time close to the optimal in most of the cases. In the extensive simulations performed, the maximum deviation of processing time with respect to the optimal is 0.8% for 5 processors, and it takes about 3 to 5 minutes to find the schedule. As the number of processors increase, the time required to compute the solution increases. For example, it takes around 80 minutes to compute the ITERLP schedule for 65 processors. Because the expected error is low, even though computation cost is high, ITERLP allows comparison of other heuristic algorithms when it is impractical to find the optimal solution. A possible hypothesis is offered for the near-optimality of the algorithm.

Chapter 5. The Two-Slave System lays the foundation of the two-slave system that forms the basis for the SPORT algorithm. Several important concepts are introduced in this chapter as below.

- The three types of possible optimal schedules in a two-slave network and the related derivations.
- Derivation of optimal schedule for two processors using simple if-then clauses and closed-form equations.
- The condition for optimality of the LIFO and FIFO schedules. The result shows that whether LIFO (resp. FIFO) is faster for a two-slave system depends only on the communication speeds of the processor links.
- The condition for the existence of idle time in a FIFO schedule. It shows a relationship between the computation and communication speeds of the two processors and the type of divisible load under consideration.
- Equivalent processor for LIFO and FIFO schedules and related derivations. The equivalent processor enables the combination of two processors into a single virtual processor.
- The extension of the equivalent processor concept to an arbitrary number of processors and its applications. A method to determine the number of processors to allocate load is derived using the equivalent processor concept.

Chapter 6. The SPORT Algorithm introduces the SPORT algorithm as a solution to the DLSRCHETS problem. Along with the allocation and collection sequences, the SPORT algorithm finds: (a) the number of processors to use for computation, and (b) the load fractions to be allocated to the participants. The important point is that this is done without solving time-consuming linear programs. The number of possible allocation and collection sequences is limited to a few, potentially optimal permutations. Because of this, given a set of processors sorted in the order of decreasing communication speed, the complexity of SPORT is $O(m)$, where m is the number of available processors. The algorithm is robust to system composition and it provides good schedules for both homogeneous and heterogeneous types of systems. In the large number of simulations performed, the maximum deviation of processing time with respect to optimal is 1.5% for 5 processors. SPORT is very fast — it takes less than a second to find the solution for 500 processors.

The basic idea behind SPORT is very simple — to use two processors at a time and build a piecewise locally optimal schedule. However it is not very straightforward to be able to do this, and several necessary tools are designed. Detailed explanation regarding the working of the algorithm is given. The method of deriving load fractions using binary tree traversal is explained.

The comprehensive simulation testing of the performance of the algorithms is undoubtedly the highlight of this chapter. SPORT performance is proved to be robust to heterogeneity, number of participants, and value of δ. Moreover, this superior performance is obtained at a fraction of the computation time of other algorithms.

Chapter 7. Conclusion summarizes the various points covered in the thesis and presents several ideas for future work. It is proposed that future work can proceed in the following main directions:

1. Theoretical analysis of complexity and other optimality results,
2. Extensions to the current system model,
3. Modifying the nature of DLSRCHETS itself, and
4. Development of applications and physical testing.

CHAPTER 2

THE SYSTEM MODEL

2.1 INTRODUCTION

To study and analyze a physical problem, it is necessary to reduce it to its equivalent mathematical form. This is called creating a *model* of the problem, or *modeling* the problem. It is difficult and sometimes unnecessary to capture every aspect of a physical system. The correct level of abstraction to be used, and the parameters to be modeled, depend on the problem under consideration and the purpose of the analysis. Once a correct model is identified, it enables fairly accurate results to be obtained mathematically without having to construct the actual system.

The model used in this thesis to approximate the master-slave system is fairly standard in DLT literature. This chapter explains the various parameters that are used throughout the remainder of this thesis and the reasons behind some of the choices that were made.

2.2 JOB EXECUTION MODEL

This thesis targets divisible loads (also interchangeably called jobs or applications) to be executed on heterogeneous master-slave platforms. The master-slave type of job execution is a popular choice for developing parallel processing applications. For example, the master-slave execution model has been used for image processing and computer vision [25], matrix multiplication problems [48], large genetic database searches [82], image rendering algorithms [95], computational fluid dynamics (CFD) codes [32], Monte Carlo simulations [10], and tree search algorithms [60].

The execution of a divisible job on each slave comprises of three distinct phases in the following order — the allocation phase, where data is sent to the slave from the master, the computation phase, where the data is processed, and the result collection phase, where the slave sends the result data back to the source. The computation phase begins only after

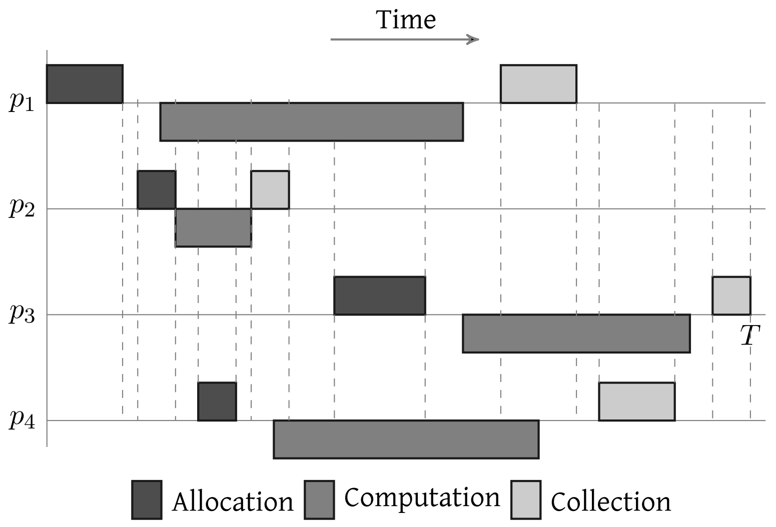


Figure 2.1 A general schedule for DLSRCHETS. Processors can do only one thing at a time — either compute or communicate. There are three phases for each processor — allocation, computation, and result collection, in that order. However, phases of different processors may be interleaved. Each phase is *atomic*, i.e., continues to its end without interruption. Communication phases (either allocation or collection) cannot overlap as shown by the dashed lines. Computation phases are independent of each other.

the entire load fraction allocated to that slave is received from the source. Similarly, the result collection phase begins only after the entire load fraction has been processed, and is ready for transmission back to the master. This is known as the *non-preemptive*, *atomic*, or *block based* model, and each phase forms a block on the time line as shown in Fig. 2.1.

2.3 COMMUNICATION AND COMPUTATION MODEL

The non-preemptive communication and computation phases necessitate that the slaves are continuously and exclusively available during the course of execution of the divisible load, and have sufficient buffer capacity to receive the entire load fraction in a single installment from the master. Traditionally, DLT has used the single installment delivery of data. Banino et al. [8] and Beaumont et al. [14] considered a multi-installment strategy. The data is considered to be split into equal sized tasks, and the maximum number of tasks that can be delivered to the processors in a given time interval is found. They argue that in the *steady state*, separate modeling of result collection is unnecessary. They concluded that allocation should proceed in the order of decreasing communication bandwidth for optimal performance in the steady-state.

In this thesis, the focus is on the more traditional form of single-installment DLS on account of the following reasons:

- To get a better understanding of the underlying problem structure when result collection and node heterogeneity are considered together.

- The scheduling of tasks is essentially left to chance in the multi-installment strategy. Collisions during communication are likely to cause delays.

- For certain applications, multi-installment distribution of data is difficult or not desirable.

All processors (master as well as slaves) follow a *unidirectional single-port* communication model, and a *no-overlap* computation model. That is, master and slaves can do only any one thing at a time — either communicate or compute (the no-overlap model), and if communicating, then either send data or receive data (the unidirectional one port model). During communication between the master and slave, both are kept busy for the entire duration of the transmission. This model is well accepted and has been used in most of the papers in this area [9, 11–13, 22–26, 34, 36, 57, 74, 84, 85], just to cite a few.

A few papers have considered DLS with a *multi-port* model for the master [54, 70, 103, 104]. If it is possible to have as many ports as there are slaves, and also be able to program the source to communicate simultaneously with all the slaves, then the problem of sequencing allocation and result collection becomes irrelevant.

The unidirectional single-port model is used in this thesis for the following reasons:

- Traditionally, DLT has used the unidirectional single-port (sequential) communication model, as evidenced by the large body of literature using this model mentioned above and in Chapter 1 versus the few papers [54, 70, 103, 104] cited above for the multi-port model.

- As mentioned in Chapter 1, this thesis addresses DLS on generic heterogeneous systems such as volunteer and grid computing platforms. The master-slave topology is an application-level *logical construct* on these systems. The source is not a special machine as used in the papers [54, 70, 103, 104] cited above, but can be any machine that wants to participate in the computation.

- An experimental setup such as the one described in [16, 18] using MPI (Message-Passing Interface) to implement the master-slave processing follows the unidirectional single-port model as it is found to be more realistic in practice. As noted in [92], *scatter-gather* operations in MPI need to be improved before it can be reliably used for simultaneous data transfer to and from several slaves.

- If each slave was to be connected to the master by a dedicated link (port), then the number of slave processors that could be used would be seriously limited as it is not practical to have a large number of physical ports on a computer.

2.4 COMMUNICATION AND COMPUTATION PARAMETERS

These days there is an almost endless variety of computers available. Computers can differ in almost every aspect – right from the low motherboard interconnection bus level to the high OS (operating system) level, and everything else in between including the CPU type, CPU speed, CPU quantity, CPU on board cache memory, size of RAM, hard disk space, auxiliary processors, to name just a few features. Thus, if one were to truly model heterogeneity, the feature vector of differentiation would be large.

However, in a master-slave system, from the master's point of view, observing the performance of an application on a slave, it is not necessary to model most of the features mentioned above. This is because, even though each feature affects the performance of the application to a certain extent, it does not help the master to know how that happens. The master has to finish a certain task in the minimum amount of time, so all it really needs to know is how much time will it take for a slave to process the task, so that the master can decide whether or not to allocate that task to that particular slave. The processing time of a task on a slave includes the time to send the relevant data to the slave, the time for the slave to carry out the requisite computation on the data, and the time required to receive the result data from the slave once the computation is complete.

Thus, even though the slaves may actually differ in a number of ways, as far as the master is concerned, the heterogeneity manifests itself only in the different time it takes to communicate and receive data from the slaves, and for the slaves to compute the results. Hence a master-slave network can be characterized in terms of the communication and computation speeds (times) of its components.

A heterogeneous master-slave (sometimes called as *star* or *single-level tree*) system $\mathcal{H} = (\mathcal{P}, \mathcal{L})$ is as shown in Fig. 2.2, where $\mathcal{P} = \{p_0, \ldots, p_m\}$ is the set of $m+1$ processors, and $\mathcal{L} = \{l_1, \ldots, l_m\}$ is the set of m network links that connect the master scheduler (source) p_0 at the center of the star (root of the tree), to the slave processors $p_1, \ldots, p_m$ at the points of the star (leaves of the tree). $\mathcal{E} = \{E_1, \ldots, E_m\}$ is the set of unit computation times of the slave processors, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is the set of unit communication times of the network links, i.e., p_k takes E_k time units to process a unit load transmitted to it from p_0 in C_k time units over the link l_k. It follows that $E_k, C_k > 0, k \in \{1, \ldots, m\}$. The values in $\mathcal{E}$ and $\mathcal{C}$ are assumed to be deterministic and available at the master.

The master holds a divisible load (job) $\mathcal{J}$ that is to be distributed and processed on $\mathcal{H}$. Based on the unit communication and computation time values of the slaves, the master p_0 splits $\mathcal{J}$ into parts (fractions) $\alpha_1, \ldots, \alpha_m$ and sends them to the respective slave pro-

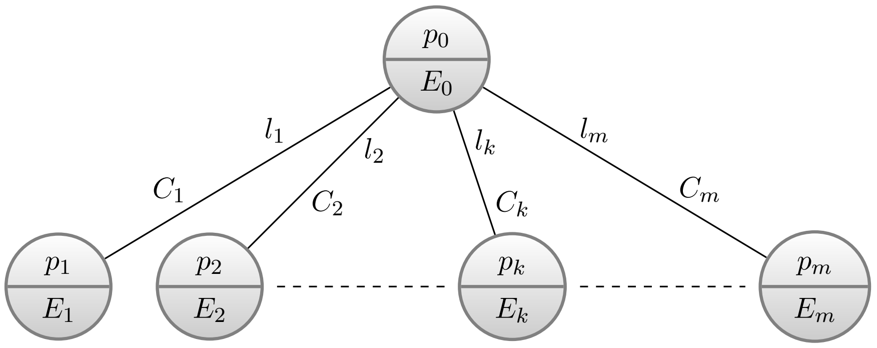


Figure 2.2 The heterogeneous master-slave system $\mathcal{H}$. The processors have different computation speeds and network bandwidths.

cessors $p_1, \ldots, p_m$ for computation. Each such set of m fractions is known as a *load distribution* $\alpha = \{\alpha_1, \ldots, \alpha_m\}$. The source does not retain any part of the load for computation. Since the job $\mathcal{J}$ is assumed to be arbitrarily divisible, $\alpha_k \in \mathbb{R}_0^+$, $\alpha_k \geq 0, k \in \{1, \ldots, m\}$.

The unit communication and computation times are conditional upon the job $\mathcal{J}$ under consideration. So ideally, the values should be indexed as $C_k^{\mathcal{J}}$ and $E_k^{\mathcal{J}}$, to indicate that the values are valid only for the job $\mathcal{J}$. In this thesis, this index is omitted as the context of analysis and discussion is always clear to be the job $\mathcal{J}$.

2.5 RESULT DATA MODEL

For the divisible loads under consideration, such as image and video processing, Kalman filtering, matrix conversions, etc., the computation phase usually involves simple linear transformations on the input data, and the volume of returned results can be considered to be proportional to the amount of load received in the allocation phase. If the allocated load fraction is α_k, then the returned result is equal to $\delta\alpha_k$. The constant δ is application specific, and is the same for all processors for a particular load $\mathcal{J}$. This is the accepted model for returned results in literature to date [1, 9, 16–18, 24, 36, 77, 104]. In this thesis, it is assumed that $0 \leq \delta \leq 1$.

2.6 COMMUNICATION AND COMPUTATION TIME

The time taken for communication and computation is assumed to be a linearly increasing function of the size of load fraction. For a load fraction α_k, $\alpha_k C_k$ is the transmission time from p_0 to p_k, $\alpha_k E_k$ is the time it takes p_k to perform the requisite processing on α_k, and $\delta\alpha_k C_k$ is the time it takes p_k to finally transmit the results back to p_0.

Though a linear model is considered for computation and communication times for the sake of simplicity, all results can be easily extended to other (e.g. affine) cost models. For example, the computation time and the load allocation time of a processor p_k can be defined as $e_k(\alpha_k)$ and $c_k(\alpha_k)$, where $e_k(\cdot)$ and $c_k(\cdot)$ are functions of the allocated load fraction α_k. Similarly, the size of result data can be an application-dependent function $d_{\mathcal{J}}(\cdot)$ of α_k, giving a result collection time $c_k(d_{\mathcal{J}}(\alpha_k))$ for a processor p_k.

The functions $c_k(\cdot)$, $e_k(\cdot)$, and $d_{\mathcal{J}}(\cdot)$ are concave, monotonically non-decreasing functions of α_k. This category of functions includes all non-decreasing linear, affine, and constant functions. The concave non-decreasing nature ensures that the communication or computation time of a slave for a larger load fraction is always greater than the time for a smaller load fraction. In this thesis, $c_k = \alpha_k C_k$ and $e_k = \alpha_k E_k$ are linear functions, while $d_{\mathcal{J}} = \delta$ is a constant function.

In *packet-switched* networks, (most distributed and grid computing platforms will use packet-switched public networks such as the Internet) the communication time of a data message between two nodes can be approximated as

$$t_{\text{comm}} = S + DU + LC,$$

where S is the message *start-up time* (required for message packing, routing decision, etc.), C represents the transmission rate (time units per data unit), L is the message length, D is the distance between two nodes (number of hops), and U is the commutation time per switch [39, 68]. It has been confirmed by Ni and McKinley [68] that $DU \ll S$ and can be neglected. In case of grid computing, the term LC is expected to dominate since the amount of data to be transferred is considerable. Thus, it is reasonable to omit S and approximate the communication delay (time) to be directly proportional to the amount of data being transferred.

Casanova and Marchal [33] introduced an interesting difference between the two types of links in grid computing — the local-area intra-cluster links, and the Internet backbone inter-cluster links. On the intra-site links, bandwidth is shared approximately in the ratio of the number of processors sharing the link. However, on the Internet backbone links, the sharing of bandwidth is not predictable. By assuming point-to-point links between each pair of processors, the details of bandwidth sharing are abstracted. This is feasible, since bandwidth between two processors will be measured using services such as the NWS (Network Weather Service) [98, 99], ReMoS (Resource Monitoring System) [38, 64], or GIS (Grid Information Service) [42, 43] that take into consideration the underlying network topology while computing (predicting) the bandwidth.

In general, the processors in a network are classified as *identical*, *uniform*, or *unrelated* [24]. Processors are *identical* if they take the same amount of time to compute a given load. Processors are said to be *uniform*, if they have different speeds, but the speeds are independent of the type of load. For example, if a processor p_x takes time t_x to process a given load, and processor p_y takes time t_y to process the same load, then the ratio t_x/t_y is a constant for all loads. The processors are *unrelated* when the speed of processors is dependent on the type of load, i.e., the ratio t_x/t_y defined above, changes with the type of load. In this thesis, all processors are of either uniform or unrelated type.

The computational capacity of a processor is *not* a deterministic quantity in the strict sense since it is likely to vary with time on a shared system. However systems such as the one used by Mutka [66], the NWS [100], or GIS [42] are able to predict the CPU availability and computation capacity with a fair degree of accuracy. This (determinism of parameters) assumption is supported by the fact that even in an analysis based on a stochastic framework, the expected values of the performance measures of interest are obtained with acceptable accuracy by replacing the random variables by their mean.

The assumption of a linear model for the computation time, rather than an affine model is justified under the circumstances because the total time for computation will be much larger than any constant computation overhead. However, as mentioned earlier, all analysis for the linear model can be easily extended to affine cost models, and linear models are assumed only for the sake of simplicity.

2.7 SUMMARY

This chapter presented the system model used in the rest of the thesis. The model is appealing in its simplicity and at the same time complete in its coverage. The assumptions are well justified for the applications and the environment targeted in the thesis — divisible loads on heterogeneous master-slave platforms.

The system model used is fairly standard in DLT literature to date. The most important constraints are:

- Communication and computation time is a linearly increasing function of the size of data.
- Size of result data is proportional to the size of allocated input load data.
- Communication and computation cannot overlap — a processor can do only one thing at a time.
- A processor can communicate with only one other processor at a time — the unidirectional single-port or one-port model is used.
- Data transmission and reception occurs in a single installment.

CHAPTER 3

ANALYSIS OF DLSRCHETS

3.1 INTRODUCTION

In the previous chapters the basic ideas of divisible loads, divisible load scheduling, result collection, and heterogeneous master-slave systems have been discussed. It is clear that one needs to somehow distribute, compute, and return results in the minimum amount of time. But what does it mean mathematically? What is a *schedule*? How can it be precisely defined? Moreover, what is the mathematical formulation of the DLSRCHETS problem? It has been noted that the complexity of DLSRCHETS is an open question. In that case, how can one analyze an optimal solution to DLSRCHETS? What information is it possible to obtain regarding the optimal solution? This chapter addresses these questions.

The two important contributions of this chapter are the proof of the allocation precedence condition and the proof of the idle time theorem. The proof of the idle time theorem shows how both non-degenerate and degenerate solutions to a linear program can be useful to offer insight into the problem structure. Another interesting part is the transition of DLSRCHETS from a traditional schedule to a linear program.

3.2 PROBLEM FORMULATION

DLSRCHETS stands for Divisible Load Scheduling with Result Collection on HETerogeneous Systems. The name of the problem suggests that there are three important components involved:

- divisible loads,
- result collection, and
- heterogeneous systems.

As mentioned previously, divisible loads can be partitioned into arbitrary sized fractions that can be processed independently of each other. After processing, these fractions generate some result data that is large enough to warrant explicitly scheduling it back to the master. The platform on which this execution takes place has computers that differ in their communication and computation performance. The master is aware of these performance differences, and must decide how to take advantage of the same.

In the DLSRCHETS problem, the master has to partition the load $\mathcal{J}$ into fractions $\alpha_1, \ldots, \alpha_m$, and manage the allocation of these fractions to, and collection of the results from the processors $p_1, \ldots, p_m$ in the minimum possible time. Let $\mathcal{T} = \{1, \ldots, m\}$ be the set of *tasks* corresponding to the m fractions that are allocated to, and $\mathcal{R} = \{1, \ldots, m\}$ be the set of *results* that are collected from the processors $p_1, \ldots, p_m$ respectively.

Though the load fractions (tasks) can be processed independently of each other on the respective processors, the single-port communication model implicitly induces a *precedence order* on the distribution of the tasks and collection of the results. Let $\prec_a$ and $\prec_c$ be *total orders* on the sets $\mathcal{T}$ and $\mathcal{R}$ respectively, such that $\prec_a$ represents the sequence (order) in which processors are allocated tasks, and $\prec_c$ is the sequence in which results are collected from the processors at the master. Then, $i \prec_a j$ implies that task i *precedes* task j (or equivalently task j *succeeds* task i) in the allocation sequence $\prec_a$, and $i \prec_c j$ signifies that result i precedes result j in the collection sequence $\prec_c$. If $\{k \in \mathcal{T} : i \prec_a k \prec_a j\} = \emptyset$, then task i is the *immediate predecessor* of task j in $\prec_a$, and is denoted as $i \lessdot_a j$. Similarly, if $\{k \in \mathcal{R} : i \prec_c k \prec_c j\} = \emptyset$, then result j is the *immediate successor* of result i in $\prec_c$, and is denoted as $i \lessdot_c j$. Define $B^i_{\prec_a} := \{j \in \mathcal{T} : j \prec_a i\} \cup \{i\}$ and $F^i_{\prec_a} := \{j \in \mathcal{T} : i \prec_a j\} \cup \{i\}$, i.e., $B^i_{\prec_a}$ is the set of task i and the tasks *before* i (*predecessors* of i) in $\prec_a$, while $F^i_{\prec_a}$ is the set of task i and the *followers* (*successors*) of task i in $\prec_a$. $B^i_{\prec_c}$ and $F^i_{\prec_c}$ are defined accordingly for $\prec_c$. The *minimal element* of $\prec_a$ is defined as $\prec_a^+ := \exists!\, i \in \mathcal{T} : B^i_{\prec_a} = \{i\}$ and the *maximal element* of $\prec_a$ is defined as, $\prec_a^- := \exists!\, i \in \mathcal{T} : F^i_{\prec_a} = \{i\}$, i.e., $\prec_a^+$ and $\prec_a^-$ are the first and last tasks allocated in $\prec_a$. $\prec_c^+$ and $\prec_c^-$ are similarly defined as the first and last results returned in $\prec_c$.

For a given load $\mathcal{J}$, the objective is to minimize the total processing time T, which is defined as the time taken from the point when the master first initiates the allocation of tasks, to the point when the master completes reception of all the results. From the system model in Chapter 2, there are two important constraints to consider while scheduling the tasks on the processors, viz. the exclusivity of the communication medium (single-port model), and the non-overlap of communication and computation.

The *schedule* $\mathcal{S}$ of DLSRCHETS for a given load distribution α, is a pair (t, r), where, $t : \mathcal{T} \mapsto \mathbb{R}_0^+$ is the task allocation start time function, and $r : \mathcal{R} \mapsto \mathbb{R}_0^+$ is the result collection start time function. In a *feasible* schedule, the start times in t and r must satisfy

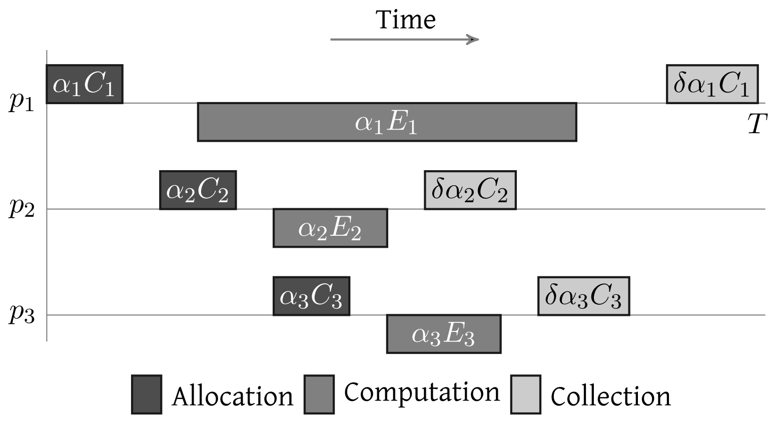


Figure 3.1 A possible schedule with $m = 3$. The three phases of each processor are atomic and satisfy the constraints (3.1) to (3.9).

the following constraints:

$$t_j - t_i \geq \alpha_i C_i \qquad \forall\, i \in \{1,\dots,m\}, i \preccurlyeq_a j \tag{3.1}$$

$$t_i \geq \sum_{j \in B^i_{\prec_a} \setminus \{i\}} \alpha_j C_j \qquad \forall\, i \in \{1,\dots,m\} \tag{3.2}$$

$$r_j - r_i \geq \delta\alpha_i C_i \qquad \forall\, i \in \{1,\dots,m\}, i \preccurlyeq_c j \tag{3.3}$$

$$T - r_i \geq \sum_{j \in F^i_{\prec_c}} \delta\alpha_j C_j \qquad \forall\, i \in \{1,\dots,m\} \tag{3.4}$$

$$r_i - t_i \geq \alpha_i C_i + \alpha_i E_i \qquad \forall\, i \in \{1,\dots,m\} \tag{3.5}$$

$$t_i \neq r_j \qquad \forall\, i,j \in \{1,\dots,m\} \tag{3.6}$$

$$r_j - t_i \geq \alpha_i C_i \qquad \forall\, j \in \{1,\dots,m\}, \forall\, t_i < r_j \tag{3.7}$$

$$t_i - r_j \geq \delta\alpha_j C_j \qquad \forall\, i \in \{1,\dots,m\}, \forall\, r_j < t_i \tag{3.8}$$

$$t_i, r_j \geq 0 \qquad \forall\, i,j \in \{1,\dots,m\} \tag{3.9}$$

The precedence constraints of $\prec_a$ are enforced by (3.1) and (3.2), while inequalities (3.3) and (3.4) impose the precedence constraints of $\prec_c$ and define the processing time T. The fact that the result collection cannot begin before the execution of the entire load fraction is complete is shown by (3.5). Constraints (3.6), (3.7), and (3.8) impose the single-port model so that no allocation and collection phase can overlap. The non-negativity of the start times is ensured by (3.9).

Figure 3.1 shows the timing diagram for a feasible schedule with $m = 3$. The time spent in communication with the master p_0 is shown above the horizontal axes, and time spent in computation by the individual processors below the horizontal axes. Since p_0

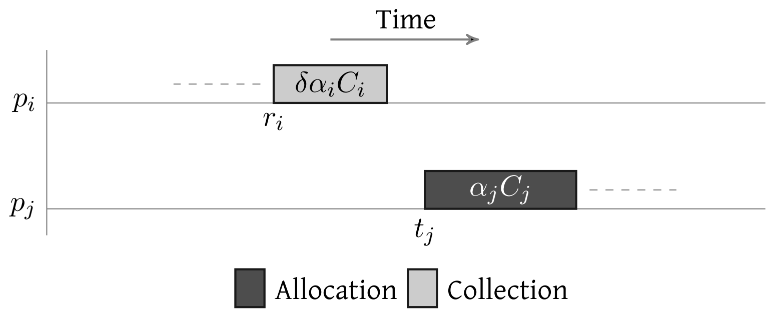


Figure 3.2 Interleaved result collection. There exists at least one pair of r_i and t_j that immediately follow each other.

does not retain any part of the load for itself, there is no p_0 axis. In this thesis, the timing diagrams are drawn so that the processors are numbered in the order of allocation sequence. This is done to keep the diagrams easy to understand, and there is no loss of generality, as it is just a question of renumbering the processors. It should be kept in mind this is not true in general, i.e., processor k is not necessarily the k^{th} processor in the allocation sequence.

In a LIFO or FIFO schedule, the order of distribution and collection of fractions is predefined, which explicitly determines t and r once α is known. However, in the general case this is not so, and to efficiently find optimal schedules, it is necessary to constrain the number of possible values that t and r can take. A lemma is stated based on the following condition that reduces the range of optimal solutions to a finite number.

Condition 3.1 (Allocation Precedence Condition)**.** The master should first allocate the entire load to the processors before receiving any results from the processors.

Lemma 3.1 (Allocation Precedence Lemma)**.** *There exists an optimal schedule for DLSRCHETS that satisfies the allocation precedence condition. (There may exist other optimal schedules that do not satisfy the allocation precedence condition.)*

Proof. Consider a feasible schedule with processing time T, that satisfies (3.1) to (3.9) for a load distribution α, and an arbitrary order of allocation and collection $\prec_a$ and $\prec_c$, such that some results are collected before the load is completely allocated first.

Then, there exists at least one pair (i, j) with $i \prec_a j$, such that the result collection starting at r_i is followed by a task allocation at t_j, without any other intermediate communication phase as shown in Fig. 3.2.

Suppose that all load fractions in α, and all other start times in t and r are maintained the same, and only the order of collection of result i and allocation of task j is exchanged, such that the new allocation start time of task j is $t'_j = r_i$, and the new collection start time of result i is $r'_i = r_i + \alpha_j C_j$.

Since the above exchange does not alter the order of allocation of different tasks, the precedence constraints of $\prec_a$ defined by (3.1) and (3.2) still hold. Similarly, the precedence constraints of $\prec_c$, imposed by (3.3) and (3.4) also hold after the exchange. The constraints (3.6), (3.7), and (3.8) are valid after the exchange because the single-port model is not violated by the exchange.

Only the conditions expressed by (3.5) require verification. Before the exchange, the conditions $r_i - t_i \geq \alpha_i C_i + \alpha_i E_i$ and $r_j - t_j \geq \alpha_j C_j + \alpha_j E_j$ are satisfied. After the exchange, the constraints (3.5) are still valid because $r'_i - t_i = r_i + \alpha_j C_j - t_i > r_i - t_i$, and $r_j - t'_j = r_j - r_i > r_j - t_j$.

From the above observations, it is clear that after the reordering, all conditions for feasibility are still satisfied. Moreover, the orders $\prec_a$ and $\prec_c$ are unchanged, and no additional processing time is required for the reordering.

If a similar reordering is carried out for all such pairs (i, j), then the allocation precedence condition is satisfied with no addition in total processing time T.

Now if there is an optimal schedule for DLSRCHETS that does not satisfy the allocation precedence condition, then a reordering can be performed as mentioned above so that the schedule satisfies the allocation precedence condition without an increase in the total processing time. That is, there always exists an optimal schedule that satisfies the allocation precedence condition, and only such schedules need be considered in the search for the optimal schedule. ■

Two other basic lemma are stated before the DLSRCHETS problem is formally defined.

Lemma 3.2. *There exists an optimal schedule for DLSRCHETS that has no idle time between any two consecutive allocation phases and any two consecutive result collection phases. (There may exist other optimal schedules that do not satisfy this condition.)*

Proof. Assume that a feasible schedule that obeys (3.1) to (3.9), and in addition also satisfies the allocation precedence condition, has idle time between the consecutive communication phases (see Fig. 3.1). Let the processing time be T, the load distribution be α, and $(\prec_a, \prec_c)$ be the orders of allocation and collection.

According to the assumptions in the system model, all processors are available continuously and exclusively during the entire execution process, and the master can only communicate with one processor at a time. For any $i \preccurlyeq_a j$, when processor p_i completes the reception of its allocated task at time $t_i + \alpha_i C_i$, processor p_j is already available and can start receiving data immediately at $t_j = t_i + \alpha_i C_i$. Because the schedule satisfies the allocation precedence condition, load is first distributed to all the processors sequentially before result collection begins. Thus the start time of each task $i \in \mathcal{T}$ can be brought forward so that $t_i = t_{\prec_a^+} + \sum_{j \in B^i_{\prec_a} \setminus \{i\}} \alpha_j C_j$, and the inequalities (3.1) and (3.2) are reduced to equalities without exceeding T.

Following a similar logic to the one above, the result collection of each result $i \in \mathcal{R}$ can be delayed to the extent necessary to make the result collection start time $r_i = T - \sum_{j \in F^i_{\prec_c}} \delta \alpha_j C_j$, with inequalities (3.3) and (3.4) reduced to equalities and no extra time added to T.

Since any feasible schedule can be reordered in this manner to eliminate the idle time between communication phases, it follows that an optimal schedule to DLSRCHETS also has no idle time between any two consecutive allocation and result collection phases. ■

Lemma 3.3. *There exists an optimal schedule for DLSRCHETS that has no idle time between the allocation and computation phases of each processor. (There may exist other optimal schedules that do not satisfy this condition.)*

Proof. Following an argument similar to the one used in Lemma 3.2, since all processors are always available, they can begin computing immediately upon receiving their load fractions in the allocation phase without affecting the schedule.

Thus, any processor p_i begins computing its allocated task at time $t_{\prec_a^+} + \sum_{j \in B^i_{\prec_a}} \alpha_j C_j$ without crossing the time interval T. Since any feasible schedule can be reordered in this manner, an optimal schedule to DLSRCHETS too has no idle time between the allocation and computation phases of each processor. ■

Theorem 3.1 (Feasible Schedule Theorem). *There exists an optimal schedule for DLSRCHETS that satisfies Lemmas 3.1 to 3.3.*

Proof. If there exists any optimal schedule that does not satisfy any of the Lemmas 3.1 to 3.3, it can always be reordered as explained in the respective proofs to satisfy the same. ■

From Theorem 3.1, it follows that only those schedules that satisfy Lemmas 3.1 to 3.3 need be considered in the search for the optimal solution to DLSRCHETS. A possible timing diagram for such a schedule is shown in Fig. 3.3.

From the preceding discussion, it can be concluded that the start times t and r in the optimal schedule for DLSRCHETS can be determined from the sequences $\prec_a$ and $\prec_c$, and the load distribution α that minimize the processing time T. Hence instead of finding t and r as in traditional scheduling practice, the DLSRCHETS problem is formulated as a linear programming problem, to find $\prec_a$, $\prec_c$, and α that minimize T. Once the optimal values of these variables are known, it is straightforward to find the optimal schedule.

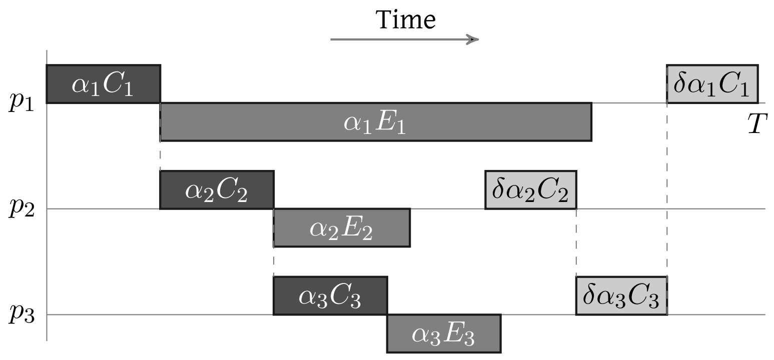


Figure 3.3 A schedule that satisfies the Feasible Schedule Theorem. Result collection starts only after load allocation to all processors is complete. Computation phase of each processor follows its allocation phase without delay. Each allocation and collection phase follows its predecessor without delay. Idle time may be present only between the computation and collection phases of each processor.

The constraints (3.1) to (3.9) and the allocation precedence condition are combined into a unified form, and for each processor p_i, constraints on T are written in terms of $B^i_{\prec_a}$ and $F^i_{\prec_c}$. The DLSRCHETS problem is defined in terms of a linear program as follows.

Definition 3.1 (Divisible Load Scheduling with Result Collection on HETerogeneous Systems).
Given a heterogeneous network $\mathcal{H} = (\mathcal{P}, \mathcal{L})$, a divisible load $\mathcal{J}$, unit communication and computation times $\mathcal{C}, \mathcal{E}$, find the sequence pair $(\prec^*_a, \prec^*_c)$, and load distribution $\alpha^* = \{\alpha^*_1, \ldots, \alpha^*_m\}$ that

Minimize T
Subject To:

$$\sum_{j \in B^k_{\prec_a}} \alpha_j C_j + \alpha_k E_k + \sum_{j \in F^k_{\prec_c}} \delta\alpha_j C_j \le T \qquad k = 1, \ldots, m \tag{3.10}$$

$$\sum_{j=1}^{m} \alpha_j C_j + \sum_{j=1}^{m} \delta\alpha_j C_j \le T \tag{3.11}$$

$$\sum_{j=1}^{m} \alpha_j = \mathcal{J} \tag{3.12}$$

$$T \ge 0, \quad \alpha_k \ge 0 \qquad k = 1, \ldots, m \tag{3.13}$$

In the above formulation, for a sequence pair $(\prec_a, \prec_c)$, and a load distribution α, the LHS (Left Hand Side) of constraint (3.10) indicates the total time spent in transmission of tasks to all the processors that must receive load before the processor p_i can begin processing its allocated task, the computation time on the processor p_i itself, and the time for transmission back to the master of results of processor p_i, and all its subsequent result transfers. For the no-overlap model to be satisfied, the processing time T should be greater than or equal to this time for all the m processors. The single-port communication model is enforced by (3.11) since its LHS represents the lower bound on the time for distribution and collection under this model. The fact that the entire load is distributed amongst the processors is imposed by (3.12). This is the *normalization equation*. The non-negativity of the decision variables is ensured by constraint (3.13).

The DLSRCHETS problem is defined similar to the *throughput maximization* problem addressed in [16–18]. The throughput maximization problem and the execution time minimization problem addressed in this thesis are *duals* of each other, and can be transformed from one form into the other. Because all equations are linear in the decision variables, an optimal solution to one form is also an optimal solution to the other form. However, the problem formulation given in [16–18] is applicable only for a single pair of allocation and collection sequences. LIFO and FIFO were selected as two instances of the problem and respective optimality results were derived. The formulation in this thesis is completely general and the scope of the problem is global, i.e. for all possible allocation and collection sequences. The optimality results for LIFO and FIFO presented in [16–18] can be easily derived as subsets of this generic formulation.

To keep the DLSRCHETS formulation as general as possible, the idle times in the definition of the problem as in [16, 18] are not included. In [16, 18], it is assumed in the system model itself that idle time always lies between the computation and result collection phase of a processor, when it may not always be so. The idle time can lie anywhere on the time-line. Lemmas 3.2 and 3.3 prove that idle time can be transferred to lie between computation and result collection phase of a processor.

Moreover, there is a discrepancy in the formulation used in [16, 18] because the constraints (2a) (corresponding to (3.10) here) are expressed as inequalities. These must actually be equalities since the idle times (x_i) are already considered in the equations.

The decision version of DLSRCHETS used to analyze the problem complexity is:

Definition 3.2 (DLSRCHETS — *Decision Variant*)**.**
Instance: Heterogeneous network $\mathcal{H} = (\mathcal{P}, \mathcal{L})$, divisible load $\mathcal{J}$, unit computation and communication times $\mathcal{E}, \mathcal{C}$, time interval T.
Question: Can load $\mathcal{J}$ be processed on $\mathcal{H}$, in at most T units of time?

Finding an optimal solution to the DLSRCHETS problem is surprisingly difficult. In fact, there is no known polynomial-time algorithm to find the optimal schedule for the general case considered in this paper, nor has the **NP**-completeness of DLSRCHETS been proved.

The problem is in **NP**, since the values of the two permutations and the load distribution can be guessed, and it can be checked if the answer to the decision question is true or false.

3.3 A PRIMER ON LINEAR PROGRAMMING

Since the analysis of the optimal solution in the next section depends entirely on the nature of linear programming, a brief introduction to the relevant aspects of linear programming is given. There is a lot of research and literature available on linear programming and combinatorial optimization. Some of the good books that were referred to during this work are [20, 35, 37, 69, 78, 94, 97].

3.3.1 *General Linear Programming Problem*

A linear programming problem searches for a vector $\boldsymbol{x} = (x_1, \ldots, x_d)^\top \in \mathbb{R}^d$ that maximizes (or equivalently, minimizes) a given linear function, among all vectors $\boldsymbol{x}$ that satisfy a given set of linear (in)equalities.

Definition 3.3 (General Linear Programing Problem)**.**
The general form of a linear programming problem is the following:
Maximize $\sum_{j=1}^{d} c_j x_j$
Subject to:

$$\begin{aligned} \sum_{j=1}^{d} a_{ij}x_j \le b_i \qquad & i = 1, \ldots, p \\ \sum_{j=1}^{d} a_{ij}x_j = b_i \qquad & i = p+1, \ldots, p+q \\ \sum_{j=1}^{d} a_{ij}x_j \ge b_i \qquad & i = p+q+1, \ldots, n \\ x_j \ge 0 \qquad & j = 1, \ldots, d \end{aligned}$$

Here, the input consists of a matrix $\boldsymbol{A} = (a_{ij}) \in \mathbb{R}^{n \times d}$, a vector $\boldsymbol{b} = (b_1, \ldots, b_d)^\top \in \mathbb{R}^d$, and a vector $\boldsymbol{c} = (c_1, \ldots, c_d)^\top \in \mathbb{R}^d$. Each coordinate of the vector $\boldsymbol{x}$ is called a *decision variable*. Each linear equality or inequality is called a *constraint*. The function $\boldsymbol{x} \to \boldsymbol{c}^\top \boldsymbol{x}$ is called the *objective function*. d denotes the number of variables, also known as the *dimension* of the problem. The number of constraints is usually denoted n.

Definition 3.4 (Canonical Form)**.**

A linear programming problem is in *canonical form* if it has the following structure:

Maximize $\sum_{j=1}^{d} c_j x_j$

Subject to:

$$\begin{aligned} \sum_{j=1}^{d} a_{ij} x_j \le b_i \qquad & i = 1, \dots, n \\ x_j \ge 0 \qquad & j = 1, \dots, d \end{aligned}$$

Or more compactly as

Maximize $\boldsymbol{c}^\top \boldsymbol{x}$

Subject to:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} &\le \boldsymbol{b} \\ \boldsymbol{x} &\ge \boldsymbol{0} \end{aligned}$$

Where, for any two vectors $\boldsymbol{u} = (u_1, \dots, u_d)^\top$ and $\boldsymbol{v} = (v_1, \dots, v_d)^\top$, the expression $\boldsymbol{u} \le \boldsymbol{v}$ means $u_i \le v_i$, $i = 1, \dots, d$.

The canonical form is important for analysis of optimal solutions. Any linear programming problem can be converted into an equivalent canonical form as follows:

- A minimization problem can be changed to a maximization problem by multiplying the objective function by -1.
- An equality constraint $\sum_{j=1}^{d} a_{ij} x_j = b_i$ can be replaced by two inequality constraints $\sum_{j=1}^{d} a_{ij} x_j \ge b_i$ and $\sum_{j=1}^{d} a_{ij} x_j \le b_i$.
- Constraints like $\sum_{j=1}^{d} a_{ij} x_j \ge b_i$ can be changed to the form $\sum_{j=1}^{d} (-a_{ij}) x_j \le -b_i$.
- Any unrestricted (unbounded) variable x_j can be replaced by the difference of two nonnegative variables, $x_j = x_j^+ - x_j^-$, where $x_j^+ \ge 0$ and $x_j^- \ge 0$.

3.3.2 Geometry of Linear Programming

Some definitions are necessary first before exploring the geometry of linear programming.[1]

Definition 3.5. The function to be maximized or minimized is called the *objective function.*

[1]My thanks to Prof. Jeff Erickson of the Dept. of Computer Science at the Univ. of Illinois at Urbana-Champaign, for sharing his lecture notes on this subject.

Definition 3.6. A vector $\boldsymbol{x} \in \mathbb{R}^d$ is *feasible* for an LP (Linear programming Problem) if it satisfies all the constraints. A set of feasible vectors is called the *constraint set*. The set of all feasible points is called the *feasible region* for that linear program.

Definition 3.7. A linear programming problem is said to be *feasible* if the constraint set is not empty; otherwise, it is said to be *infeasible*.

Definition 3.8. A feasible maximum (resp. minimum) problem is said to be *unbounded* if the objective function can assume arbitrarily large positive (resp. negative) values at feasible vectors; otherwise, it is said to be *bounded*. Thus there are three possibilities for a linear programming problem — it may be bounded feasible, it may be unbounded feasible, or it may be infeasible.

Definition 3.9. The *value* of a bounded feasible maximum (resp, minimum) problem is the maximum (resp. minimum) value of the objective function as the variables range over the constraint set.

Definition 3.10. A feasible $\boldsymbol{x} \in \mathbb{R}^d$ at which the objective function achieves the value is called *optimal*.

It is possible to interpret the constraints of a linear programming problem geometrically as follows:

- Any linear equation in d variables of the form $\boldsymbol{a}^\top \boldsymbol{x} = b$ defines a *hyperplane* in $\mathbb{R}^d$.
- Any linear inequality in d variables of the form $\boldsymbol{a}^\top \boldsymbol{x} \leq b$ or $\boldsymbol{a}^\top \boldsymbol{x} \geq b$ defines a *halfspace* in $\mathbb{R}^d$, i.e., the hyperplane divides $\mathbb{R}^d$ into two halfspaces, each of which is the set of points that satisfies a linear inequality.
- The set of feasible points (the feasible region) is the intersection of several hyperplanes (one for each equality constraint) and halfspaces (one for each inequality constraint). The intersection of a finite number of hyperplanes and halfspaces is called a *polyhedron*; in case of the canonical form, it is the intersection of $n + d$ halfspaces.
- Any halfspace and therefore any polyhedron is *convex*, i.e., if the polyhedron contains two points x and y, then it contains the entire line segment $\overline{xy}$.
- The problem of optimizing the objective function over all feasible vectors, is then the question of finding a point in the polyhedron that is farthest in the direction specified by the objective function. By appropriately rotating $\mathbb{R}^d$ (so that the objective function points downward), the linear program can be geometrically interpreted as looking for the lowest point in a convex polyhedron in $\mathbb{R}^d$

- In *non-degenerate* linear programs, at most d constraint hyperplanes pass through any point, and no constraint hyperplane is normal to the objective vector. In *degenerate* linear program, more than d constraint hyperplanes can pass through a point, and the feasible vector is not unique at that point.

3.3.3 *Bases, Feasibility, and Local Optimality*

A *basis* is a subset of d constraints, which for non-degenerate linear programs must be linearly independent. The *location* of a basis is the unique point $\boldsymbol{x}$ that satisfies all d constraints with equality; geometrically, $\boldsymbol{x}$ is the unique intersection point of the d hyperplanes. The *value* of a basis is $\boldsymbol{c}^\top \boldsymbol{x}$, where $\boldsymbol{x}$ is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of $\mathbb{R}^d$ into convex polyhedra; this cell decomposition is called the *arrangement* of the hyperplanes. Every d-tuple of hyperplanes (i.e., every basis) defines a *vertex* of this arrangement (the location of the basis). Thus "vertex" and "basis" can be used interchangeably.

A basis is *feasible* if its location $\boldsymbol{x}$ satisfies all the d linear constraints, or geometrically, if the point $\boldsymbol{x}$ is a vertex of the polyhedron.

A basis is *locally optimal* if its location $\boldsymbol{x}$ is the optimal solution to the linear program with the same objective function, and *only* the constraints in the basis. Geometrically, a basis is locally optimal, if its location $\boldsymbol{x}$ is the lowest point in the intersection of those d halfspaces.

It can be proved that the value of every feasible basis is less than or equal to the value of every locally optimal basis, i.e., every feasible vertex is higher than every locally optimal vertex. If a linear program has an optimal solution, it is the *unique* vertex that is *both* feasible and locally optimal.

There are several algorithms to find the optimal solution to a linear programming problem, the most popular ones based on the *Simplex method* [37] and the *Interior-Point method* [55]. However, an explanation regarding these methods is beyond the scope of this work.

3.4 ANALYSIS OF OPTIMAL SOLUTION

The processors allocated non-zero load fractions are called *participating processors* or *participants.*

Theorem 3.2 (Idle Time Theorem)**.** *There exists an optimal solution to the DLSRCHETS problem, in which irrespective of whether load is allocated to all available processors, at the most one of the participating processors has idle time, and the idle time exists only when the result collection begins immediately after the completion of load distribution.*

Proof. For a pair $(\prec_a, \prec_c)$, the DLSRCHETS problem defined by (3.10) to (3.13) always has a feasible solution. This is because, for any load distribution α that satisfies (3.12), T can be made arbitrarily large to satisfy the inequalities (3.10) and (3.11). It implies that the polyhedron formed by the constraints of the DLSRCHETS problem, $P := \{\boldsymbol{x} \in \mathbb{R}^{m+1} : A\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{x} \geq 0\} \neq \emptyset$.

According to the theory of linear programming, the optimal solution to DLSRCHETS is obtained at some vertex of this polyhedron [37, 94]. As the DLSRCHETS problem has $m + 1$ decision variables and $2m + 3$ constraints, in a *non-degenerate* optimal solution, at the optimal vertex, $m+1$ constraints out of these must be *tight*, i.e., satisfied with equality. In a *degenerate* optimal solution, more than $m + 1$ constraints are tight.

It is clear that in an optimal solution, the normalization constraint (3.12) will always be tight, and T will always be greater than zero. This means that m constraints out of the remaining $2m+1$ constraints will be tight in a non-degenerate optimal solution. There are two possible ways to proceed with the analysis at this point depending on the assumption regarding the allocated load fractions in the optimal solution.

1. $\boldsymbol{\forall\, k \in \{1, \ldots, m\} : \alpha_k > 0}$.

   In this case, all the load fractions are assumed to be always greater than zero, i.e. number of participants is m. Since all decision variables are positive, there can be no degeneracy [94, Chapter 3].

   It leaves only $m + 1$ constraints (3.10) and (3.11), out of which m will be tight in the optimal solution. Hence, in the optimal solution, either,

   (a) the m constraints (3.10) are tight, and the (3.11) constraint is not, or

   (b) the (3.11) constraint is tight and one of the (3.10) constraints is not.

   If any constraint from (3.10) and (3.11) is not tight in the optimal solution, it implies a *shortfall* in the LHS as compared to the optimal processing time. In constraints (3.10) this shortfall represents idle time in a processor, while in (3.11) it represents the intervening time interval between completion of load distribution from the master and the start of result transfer to the master.

   Thus, if the option (a) above is true, then none of the processors have any idle time in the optimal solution. If the option (b) is true, then one of the processors has idle time, and since this happens only when constraint (3.11) is tight, it means that idle time in a processor exists only when result transfer to the master begins immediately after completion of load allocation is completed. This is similar to the analysis in [16, 18].

2. $\boldsymbol{\exists\, k \in \{1, \ldots, m\} : \alpha_k = 0}$.

   In this case, some of the processors can be allocated zero load in the optimal solution.

   The analysis has two parts — for non-degenerate and degenerate optimal solutions.

   Non-degenerate Optimal Solution

   If there are p $(p \leq m)$ participants in the optimal solution, then $m - p$ constraints of (3.13) are necessarily tight. This means that out of the $m + 1$ constraints (3.10) and (3.11), only p constraints will be tight in the optimal solution. Hence, in an optimal solution, either,

   (a) p of the (3.10) constraints are tight, $m - p$ of the (3.10) constraints are not tight, and the (3.11) constraint is not tight, or
   (b) the (3.11) constraint is tight, $p - 1$ of the (3.10) constraints are tight, and $m - p + 1$ of the (3.10) constraints are not tight.

   In the optimal solution, if the option (a) is true, then $m - p$ processors have idle time, while if the option (b) is true, then $m - p + 1$ processors have idle time.

   Since $m-p$ processors are not allocated load, it is obvious that they are idle throughout in either of the above two options. The additional processor with idle time if the option (b) is true has to be one of the participating processors. This means that idle time in a participating processor exists only when the result collection begins immediately upon completion of load allocation.

   Degenerate Optimal Solution

   Similar to the non-degenerate case, if there are p $(p \leq m)$ participants in the optimal solution, then $m-p$ constraints of (3.13) are necessarily tight. Since the optimal solution is degenerate, *more than* p constraints out of the $m + 1$ constraints (3.10) and (3.11) will be tight.

   This means that in the optimal solution, irrespective of whether the (3.11) constraint is tight, *at least* p of the (3.10) constraints are tight, and *less than* $m - p$ of the (3.10) constraints are not tight. Since $m - p$ processors are necessarily idle, some of the (3.10) constraints corresponding to the processors allocated zero load are tight in the degenerate solution.

   Since $\forall\, k \in \{1, \ldots, m\},\ B^k_{\prec_a}, F^k_{\prec_c} \subseteq \{1, \ldots, m\}$, it implies that,

$$\sum_{j \in B^k_{\prec_a}} \alpha_j C_j \leq \sum_{j=1}^{m} \alpha_j C_j \qquad\qquad k \in \{1, \ldots, m\}$$

and

$$\sum_{j \in F^k_{\prec_c}} \delta\alpha_j C_j \leq \sum_{j=1}^{m} \delta\alpha_j C_j \qquad\qquad k \in \{1, \ldots, m\}$$

It follows that,

$$\sum_{j \in B^k_{\prec_a}} \alpha_j C_j + \sum_{j \in F^k_{\prec_c}} \delta\alpha_j C_j \leq \sum_{j=1}^{m} \alpha_j C_j + \sum_{j=1}^{m} \delta\alpha_j C_j \qquad k \in \{1, \ldots, m\} \tag{3.14}$$

If (3.11) is not tight, then the RHS (Right Hand Side) of (3.14) is strictly less than T. That is,

$$\sum_{j \in B^k_{\prec_a}} \alpha_j C_j + \sum_{j \in F^k_{\prec_c}} \delta\alpha_j C_j < T \quad k \in \{1, \ldots, m\} \tag{3.15}$$

If $\exists\ k \in \{1, \ldots, m\} : \alpha_k = 0$, then $\alpha_k E_k = 0$, and from (3.15), it immediately follows that the corresponding constraint from (3.10) can never be tight.

Thus, a constraint corresponding to a processor p_k allocated zero load is tight in the optimal solution only if

$$\sum_{j \in B^k_{\prec_a}} \alpha_j C_j + \sum_{j \in F^k_{\prec_c}} \delta\alpha_j C_j - T = 0 \tag{3.16}$$

or equivalently if (3.14) is satisfied with an equality, *and* the RHS of (3.14) is equal to T, i.e, the (3.11) constraint is tight.

It is now clear that a degenerate optimal solution exists only when the (3.11) constraint is tight, and the condition (3.16) is satisfied. To find when the condition is satisfied, consider the case where for some pair $(\prec_a, \prec_c)$, one or more of the processors allocated zero load follow each other at the end of the allocation sequence and the start of the result collection sequence in the optimal solution.

For example, if $\alpha_i, \alpha_j, \alpha_k = 0$, and one or more of the following occur (the list is not exhaustive):

- $\prec_a^- = i$ and $\prec_c^+ = i$
- $i \preccurlyeq_a j$, $\prec_a^- = j$ and $\prec_c^+ = i$
- $i \preccurlyeq_a j$, $\prec_a^- = j$, $\prec_c^+ = k$ and $k \preccurlyeq_c i$

Only if such tail-end zero-load processors exist, then (3.14) is satisfied with an equality. Finally, if constraint (3.11) is tight in the optimal solution, then it follows that

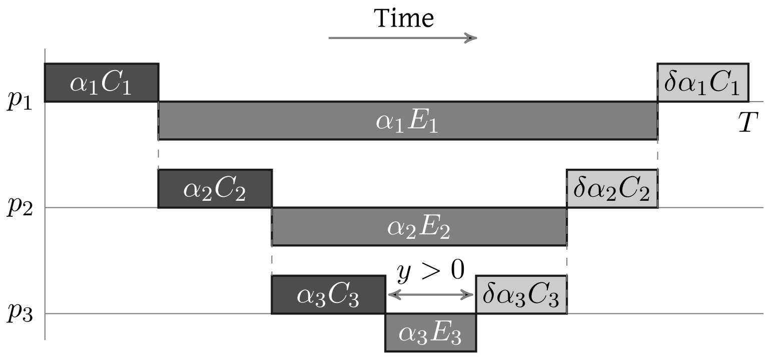


Figure 3.4 Timing diagram for the optimal LIFO schedule. There is no idle time in any processor because the intervening time interval between the end of the last allocation phase and the start of the first result collection phase, y, is always greater than zero.

the constraints corresponding to these processors are tight.

The linear program obtained after eliminating the redundant constraints corresponding to the tail-end zero-load processors has a non-degenerate optimal solution. This is because, the feasible region defined by the constraints of the non-degenerate problem does not change after addition of the redundant constraints. Hence only a single participant processor has idle time in the degenerate optimal solution.

From the preceding discussion on the optimal solution to the linear program for a pair $(\prec_a, \prec_c)$, it follows that in the optimal solution to the DLSRCHETS problem, $(\prec_a^*, \prec_c^*, \alpha^*)$, at the most one participating processor can have idle time. The idle time occurs *only when* the result collection from processor $\prec_c^+$ starts immediately after completion of load allocation to processor $\prec_a^-$. ■

There are $m!$ possible permutations each of $\prec_a$ and $\prec_c$, and the linear program has to be evaluated $(m!)^2$ times to determine the globally optimum solution $(\prec_a^*, \prec_c^*, \alpha^*)$ for DLSRCHETS. Since the solution to the linear program is completely determined by the values of $\delta, \mathcal{C}$ and $\mathcal{E}$, along with the pair $(\prec_a, \prec_c)$, it is not possible at this stage to predict which of the processors or how many processors will be allocated zero load.

Two corollaries to the optimal schedule theorem follow immediately.

Corollary 3.1 (Optimal LIFO Schedule). *There exists an optimal LIFO schedule in which no processor has idle time.*

Proof. Figure 3.4 shows an optimal LIFO schedule for three slave processors. In a LIFO schedule, the intervening time interval between the end of the last allocation phase and the start of the first result collection phase, $y \geq \alpha_m E_m$, the computation time of the last

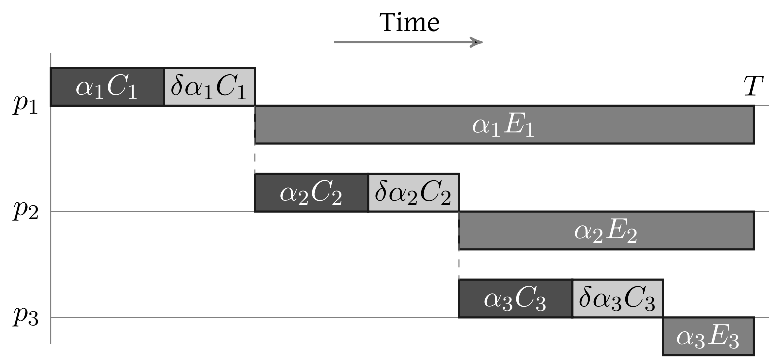


Figure 3.5 The optimal LIFO schedule rearranged so that the communication phases of each processor are grouped together. The resulting constraints are mathematically equivalent to the ones for Fig. 3.4. This rearrangement converts the schedule in to a form similar to the one in Fig. 1.3 where no results are returned to the master.

processor in the allocation sequence. It is trivial to prove that the equality occurs in an optimal schedule.

Since the intervening time interval, $y > 0$ always, it implies that constraint (3.11) can never be tight. From the idle time theorem, it follows that no processor ever has idle time in the optimal LIFO schedule. ■

An interesting addition to this proof is provided by Beaumont et al. [17]. It is proved that the optimal LIFO schedule always uses all available processors. A brief sketch of the proof is as follows. The LIFO schedule is rearranged so that the allocation and collection phases of each processor are grouped together, followed by the computation phase. For processor p_k, this gives a combined communication time of $(1 + \delta)\alpha_k C_k$ followed by a computation phase of length $\alpha_k E_k$ as shown in Fig. 3.5. The resulting constraint equations are mathematically equivalent to the form where the allocation and collection phases are on either side of the computation phase as in Fig. 3.4.

This reordering of the timing diagram converts the schedule in to a form similar to the case in Fig. 1.3, where no results are returned to the master, and all optimality results for this form are applicable to the LIFO schedule. It is a well known result in DLT literature that when results are not returned to the master, the optimal solution uses all available processors [24]. Hence the proof.

Corollary 3.2 (Optimal FIFO Schedule)**.** *There exists an optimal FIFO schedule in which*

1. *either no participating slaves have idle time, or*
2. *at most one participating slave may have idle time.*

There may exists other optimal solutions that do not satisfy these conditions.

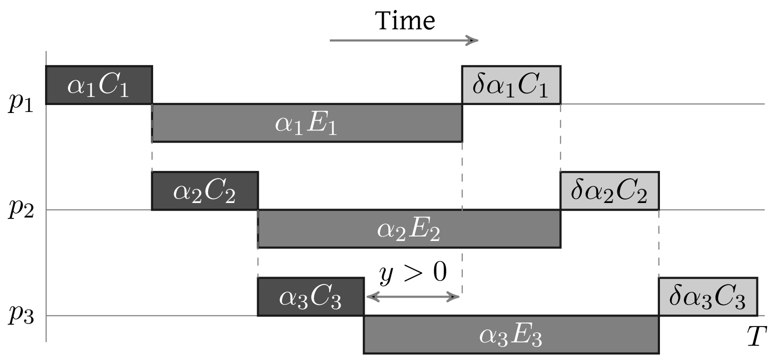


Figure 3.6 Timing diagram for an optimal FIFO schedule. In this schedule, there is no idle time as constraint (3.11) is not tight, i.e., the intervening time interval between the end of the last allocation phase and the start of the first result collection phase, $y > 0$.

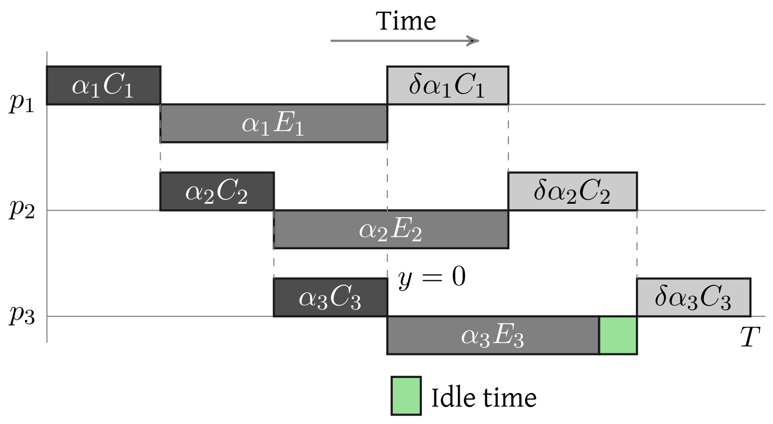


Figure 3.7 Timing diagram for an optimal FIFO schedule with idle time. In this schedule, the intervening time interval between the end of the last allocation phase and the start of the first result collection phase, $y = 0$, i.e., constraint (3.11) is tight. This necessitates idle time to be present.

Proof. In a FIFO schedule, the intervening time interval between the end of the last allocation phase and the start of the first result collection phase, y, can be either greater than or equal to zero as shown in Figs. 3.6 and 3.7. Note that $y < 0$ is not possible as it violates the unidirectional one-port condition.

If the intervening time interval $y > 0$ in an optimal solution, then from the idle time theorem, it follows that no participating processor will have idle time. On the other hand if constraint (3.11) is tight, and $y = 0$ in an optimal solution, then there can be a single processor with idle time.

There may be other optimal solutions when $y = 0$ in which more than one processor has idle time. But, it can be guaranteed that at least one optimal solution exists in which there is a single processor with idle time. ∎

In addition to this corollary, in [16, 18], two other results are proved:

1. The optimal order of allocation (and result collection) in a FIFO schedule is when processors $p_1, \ldots, p_m$ are arranged such that $C_1 \leq C_2 \leq \ldots \leq C_m$.

2. For such an optimal FIFO schedule, if a processor has idle time, then this processor can always be chosen to be processor $\prec_a^-$, i.e., the last processor in the allocation sequence. In this case it is also the processor with the slowest network link.

It is important to note however, that both these results may not be true for the general case considered in this thesis.

3.5 THE IMPORTANCE OF IDLE TIME

As mentioned earlier, traditional divisible load theory both with and without result collection, usually only considers schedules in which a processor has no idle time once it begins reception of its allocated load fraction. Two factors were responsible for the attitude towards idle time:

1. usually only homogeneous systems were considered, and

2. when heterogeneous systems were considered, it was assumed that the computation speed of a processor is slower than the communication speed of its network link.

But both these assumptions are not always valid in the new distributed computing models that use open and shared resources.

In this work, because of the one-port communication restriction, the load allocation and result collection necessarily proceeds in a sequential manner, and a processor has to be idle up to the point where load transfer to it from the master begins. This idle time is minimized by having each allocation and collection phase follow its predecessor without delay, as proved in Lemma 3.2. Similarly, any idle time between the allocation phase and computation phase of a processor only adds to the makespan T, as shown in Lemma 3.3. But the same reasoning cannot be extended to the idle time between the computation phase and result collection phase in a processor. In fact, the example in this section shows that sometimes there can be no schedule if idle time is not considered to be present before the result collection phase.

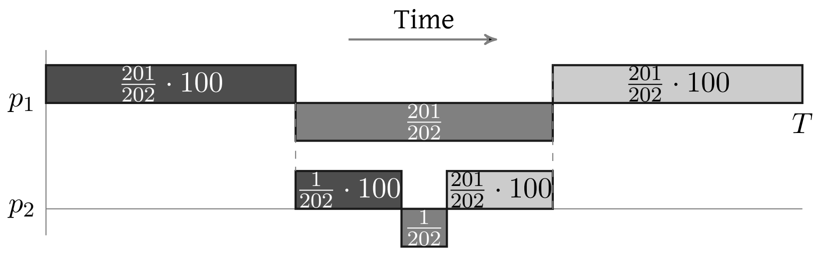

Time
p_1
$\frac{201}{202} \cdot 100$
$\frac{201}{202}$
$\frac{201}{202} \cdot 100$
T
p_2
$\frac{1}{202} \cdot 100$
$\frac{1}{202}$
$\frac{201}{202} \cdot 100$

Figure 3.8 The optimal LIFO schedule for the system configuration in Example 3.1. There is no idle time, and the makespan $T = \frac{201}{202}(100 + 1 + 100) \approx 200.005$ units.

Example 3.1. Consider the following system configuration: $m = 2$, $C_1 = C_2 = 100$, $E_1 = E_2 = 1$, and $\delta = 1$. It is a system with two identical slave processors where the communication speed is 100 times slower than the computation speed. Admittedly, the parameters are a bit exaggerated to simplify explanation, but similar results can be easily obtained with other values.

For a system with two slaves, there are three possible cases:

Individual schedule If the entire load were to be distributed to either p_1 or p_2, the processing time would be $T = 100 + 1 + 100 = 201$ units.

LIFO schedule For the optimal LIFO schedule, the load fractions and processing time can be calculated easily using the formulae in Section 5.4 as $\alpha_1 = \frac{201}{202}$, $\alpha_2 = \frac{1}{202}$, and $T = \frac{201}{202}(100 + 1 + 100) \approx 200.005$ units. The resulting timing diagram is shown in Fig. 3.8. The result is an improvement over the earlier case where the entire load is distributed to a single processor.

FIFO schedule An optimal FIFO schedule without idle time *cannot be constructed* for this system. There are at least two optimal FIFO schedules for this system when idle time is considered to be present between the computation phase and result collection phase of each processor.

The first one is obtained by using the formulae in Section 5.3. The load fractions are computed as $\alpha_1 = \alpha_2 = \frac{101}{202} = 0.5$. The timing diagram for this case is shown in Fig. 3.9. It is observed that idle time of 49.5 units is present in both processors, and the total processing time is $T = (50 + 0.5 + 49.5 + 50 + 50) = 200$ units, which is better than the LIFO schedule.

Similarly, using the formulae in Section 5.5, the load fractions are obtained as $\alpha_1 = \frac{100}{101}$ and $\alpha_2 = \frac{1}{101}$. The timing diagram for this schedule is as shown in Fig.3.10. The idle time of $\frac{9999}{101} = 99$ units is transferred to the single processor p_2. The total processing time is still $T = \frac{100}{101}(100 + 1 + 100 + 1) = 200$ units.

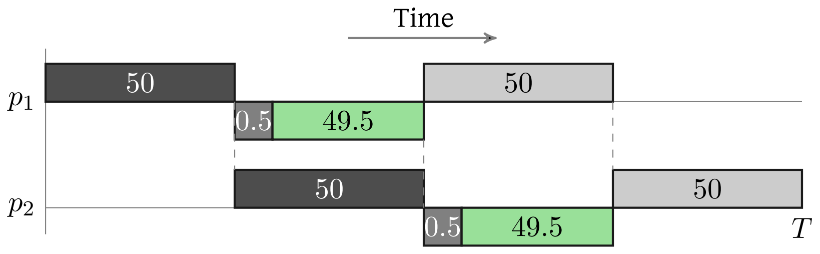


Figure 3.9 An optimal FIFO schedule with idle time in both processors for Example 3.1. The makespan is $T = (50 + 0.5 + 49.5 + 50 + 50) = 200$ units.

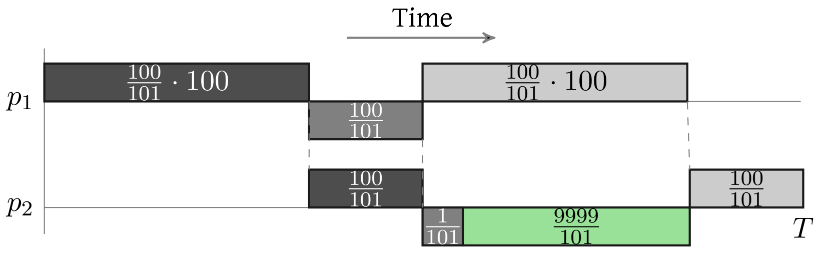


Figure 3.10 Optimal FIFO schedule with idle time in one processor for Example 3.1. The makespan is $T = \frac{100}{101}(100 + 1 + 100) = 200$ units. This schedule satisfies the idle time theorem and Corollary 3.2.

The above example clearly shows that when result collection is considered together with slow communication speeds, idle time is an important factor when minimizing the total processing time.

It is important to note that neither having two optimal FIFO solutions, nor having an optimal FIFO solution with idle time in both processors is a violation of idle time theorem or Corollary 3.2. Both the idle time theorem and Corollary 3.2 state that there is at least one optimal solution with idle time in a single processor. It does not necessarily mean that there is only one optimal solution.

3.6 SUMMARY

This chapter primarily presented the analysis of the optimal solution to DLSRCHETS. Two important proofs were given — one for the allocation precedence condition and the other for the idle time theorem. The allocation precedence condition is necessary to limit the number of possible schedules of DLSRCHETS to a finite number. It argues that there always exists an optimal solution to DLSRCHETS in which the entire load is first distributed to the slaves before the master starts to receive results from the slaves. The proof uses simple rearrangement of the timing diagram to prove the claim.

The proof of the idle time theorem is a bit more complicated. It uses the theory behind the geometry of linear programming. A brief introduction to linear programming is also included in the chapter for this reason. The idle time theorem makes a very interesting claim — that not all slaves may be allocated load in the optimal solution, and irrespective of the number of slaves that are allocated load, at most one slave can have idle time in the optimal solution.

The assumption that all processors are allocated load can greatly simplify analysis, but it is not realistic. Instead of making this assumption without justification, the case when all processors are not assumed to be allocated load in the optimal solution is considered. The analysis is not so simple in this case. In linear programming, there is a possibility of some solutions being degenerate. Hence the analysis is carried out for both non-degenerate and degenerate cases. It is proved that the idle time theorem is true for both cases.

CHAPTER 4

THE ITERLP ALGORITHM

4.1 INTRODUCTION

If the network $\mathcal{H}$ has m slave processors, then $(m!)^2$ linear programs have to be solved to find the optimal solution for DLSRCHETS. This is practically impossible to carry out for $m \geq 7$. For example, it takes 80 minutes to find the optimal solution to DLSRCHETS for $m = 6$ on a Power Mac G5, with 2GB of memory. Simple linear extrapolation yields that to compute the optimal solution for $m = 7$, it will take 3920 minutes or approximately 65 hours; that is if the computer does not first run out of memory.

The complexity of DLSRCHETS is an open problem and finding the optimal solution is difficult. Thus, one has to resort to heuristic algorithms under the circumstances. The logical approach to solving a combinatorial optimization problem by approximation is *pruning*. That is, to find some criterion that can be used to reduce the number of possible output combinations. This is the approach taken by *branch and bound* and *genetic* algorithms for example.

In this chapter, the ITERLP algorithm is proposed. It uses a well-known result in DLT to prune the number of linear programs to be solved. The complexity of ITERLP is polynomial in m, and in the worst case $O(m^3)$ linear programs have to be solved. This is still quite expensive for large values of m, but as the simulation results show, ITERLP can be used as a benchmark to compare other heuristic algorithms when it is impossible to find the optimal solution.

4.2 BRIEF INTRODUCTION TO PERMUTATIONS

In Chapter 3, it was explained how the allocation and collection sequences define precedence orders. The entire analysis was carried out using the sequences as *total orders*. For further analysis, it is important to understand the allocation and collection sequences as $permutations$. A brief explanation of permutations is given in this section. Landin [61] serves as a good introductory reference.

Definition 4.1. A *permutation* f, also called an *arrangement number*, is a rearrangement of the elements of an ordered list (set) V into a one-to-one correspondence onto V itself, i.e., $f : V \mapsto V$. This means that for any $v \in V$, there exists a $u \in V$, such that $f(u) = v$.

A permutation can be considered as a way of *reindexing* the set $V = \{v_1, \ldots, v_n\}$. For the sake of convenience, when discussing permutations, symbols for the elements in a set (e.g. v) are omitted and only the indices are considered. Then permutations just involve operations with the *index set* $\{1, \ldots, n\}$. The permutation f is represented by listing its values at $i = 1, \ldots, n$ as $\{f(1), \ldots, f(n)\}$.

The number of permutations on a set of n elements is given by $n!$ (n factorial).

Example 4.1. For example, there are $2! = 2$ permutations of $\{1, 2\}$, namely $\{1, 2\}$ and $\{2, 1\}$, and $3! = 6$ permutations of $\{1, 2, 3\}$, namely $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and $\{3, 2, 1\}$.

Definition 4.2. The *symmetric group* S_n of degree n, is the *group* of all permutations on n symbols.

S_n is therefore a *permutation group* of order $n!$ and contains as subgroups every group of order n.

Example 4.2. $S_3 = \{\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}\}$.

Definition 4.3. An *inverse permutation* is a permutation in which each number and the number of the place which it occupies are exchanged.

The inverse permutation is represented as $\{f^{-1}(1), \ldots, f^{-1}(n)\}$. Inverse permutations are sometimes also called *conjugate* or *reciprocal permutations*. Inverse permutations are important because they allow finding the position of a slave processor in the allocation or collection sequence.

Example 4.3. For example, $f_1 = \{3, 1, 4, 2\}$ and $f_2 = \{2, 4, 1, 3\}$ are inverse permutations, since the positions of 1, 2, 3, and 4 in f_1 are f_2, and the vice versa.

The number of ways of obtaining an *ordered subset* of k elements from a set of n elements is given by

$$_nP_k = \frac{n!}{(n-k)!}$$

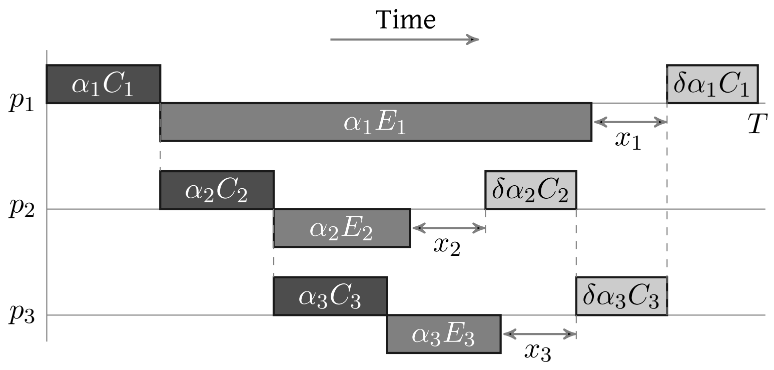


Figure 4.1 A schedule for $m = 3$ that satisfies the Feasible Schedule Theorem. Result collection begins only after the entire load is distributed. Each allocation and result collection phase follows its predecessor without delay. The computation phase of each processor follows its allocation phase without delay. Idle time may be present in each processor between the end of its computation phase and the start of the result collection phase.

Example 4.4. For example, there are $4!/2! = 12$ ordered 2-subsets of $\{1, 2, 3, 4\}$, namely $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 1\}$,$\{2, 3\}$, $\{2, 4\}$, $\{3, 1\}$, $\{3, 2\}$, $\{3, 4\}$, $\{4, 1\}$, $\{4, 2\}$, and $\{4, 3\}$.

Definition 4.4. The *unordered subsets* containing k elements are known as the *k-subsets* of a given set.

The number of k-subsets on n elements is given by the *binomial coefficient* $\binom{n}{k}$.

Example 4.5. There are $\binom{3}{2} = 3$, 2-subsets of $\{1, 2, 3\}$, namely $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3\}$.

The allocation and collection sequences can be considered as permutations on the index set of the set of slave processors $\{p_1, \ldots, p_m\}$ of a heterogeneous network $\mathcal{H}$. In the remainder of this thesis, the allocation sequence is represented by σ_a, and the collection sequence is represented by σ_c, both of which are permutations of the index set $K = \{1, \ldots, m\}$ of slave processors.

To differentiate from the representation of σ_a as a *function*, the values in the allocation and collection permutations are accessed using the standard square bracket array index notation: $\sigma_a[i]$ and $\sigma_c[i]$, $i = 1, \ldots, m$. The inverse permutations σ_a^{-1} and σ_c^{-1} act as query or lookup functions, so that $\sigma_a^{-1}[i]$ and $\sigma_c^{-1}[i]$, $i = 1, \ldots, m$, indicate the position of processor p_i in the allocation and collection sequence respectively.

Example 4.6. For the schedule shown in Fig: 4.1, $\sigma_a = \{1, 2, 3\}$, $\sigma_c = \{2, 3, 1\}$, $\sigma_a^{-1} = \{1, 2, 3\}$, and $\sigma_c^{-1} = \{3, 1, 2\}$.

Using this new convention, after application of the *Feasible Schedule Theorem* (Theorem 3.1), for a sequence pair (σ_a, σ_c) and load distribution $\alpha = \{\alpha_1, \ldots, \alpha_m\}$, a slave processor p_i

- starts receiving its data at $t_i^{\text{recv}} = \sum_{j=1}^{\sigma_a^{-1}[i]-1} \alpha_{\sigma_a[j]} C_{\sigma_a[j]}$
- starts execution at $t_i^{\text{exec}} = t_i^{\text{recv}} + \alpha_i C_i$
- stops execution at $t_i^{\text{stop}} = t_i^{\text{recv}} + \alpha_i C_i + \alpha_i E_i = t_i^{\text{exec}} + \alpha_i E_i$
- starts sending results at $t_i^{\text{send}} = T - \sum_{j=\sigma_c^{-1}[i]}^{m} \delta\alpha_{\sigma_c[j]} C_{\sigma_c[j]}$
- may have idle time $x_i = t_i^{\text{send}} - t_i^{\text{stop}} \geq 0$

4.3 PROPOSED ALGORITHM

The ITERLP (ITERative Linear Programming) heuristic algorithm finds a solution by iteratively solving linear programs. The rationale behind ITERLP is as follows. All optimality results to date for DLS on heterogeneous systems, those ignoring result collection [22, 24, 27, 28, 56, 59] as well as those considering result collection [1, 9, 16–18], have advocated load allocation in the order of decreasing communication link bandwidth. Hence processors are initially sorted in that order in ITERLP. Since neither LIFO nor FIFO schedule is always optimal, the new processor being introduced in an iteration could potentially be interleaved in any position in the optimal sequence. So the ITERLP heuristic tests all possible positions for the newly introduced processor. To build the sequences at reasonable (polynomial) cost, ITERLP assumes that the relative positions of the processors already determined are not modified by the additional processor.

In the algorithm below, the allocation and collection sequences σ_a and σ_c are indexed by superscripts $k \in \{1, \ldots, m\}$, to indicate the number of slave processors in the sequence.

Algorithm 1 (ITERLP).

```
1: Sort processors $p_1, \ldots, p_m$ such that $C_1 \leq \ldots \leq C_m$
2: Select processors $p_1$ and $p_2$
3: Find optimal allocation and collection sequence pair $(\sigma_a^2, \sigma_c^2)$
4: Add processor $p_3$
5: Find optimal sequence pair $(\sigma_a^3, \sigma_c^3)$ such that the relative order of $\sigma_a^2$ and $\sigma_c^2$ is preserved
6: Similarly add one processor at a time, and for $k \leq m$ processors, find optimal sequence pair
   $(\sigma_a^k, \sigma_c^k)$ such that the relative orders of $\sigma_a^{k-1}$ and $\sigma_c^{k-1}$ are preserved
```

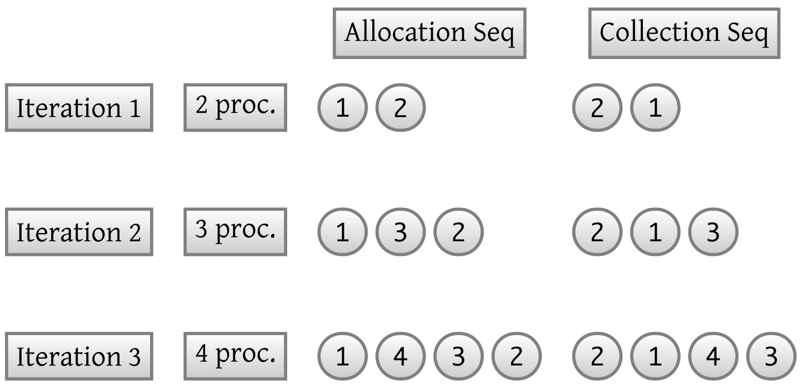


Figure 4.2 ITERLP progress illustrated for 4 processors. The number of processors at each iteration increases by one. The relative positions of the processors do not change from one iteration to the next. With k processors, there are k possible allocation sequences and k possible collection sequences, requiring k^2 LPs to be solved. The worst case complexity of ITERLP is of the order of $\sum_{i=1}^{m} k^2 = m^3$.

7: In any iteration, if any processor is allocated zero load, then the algorithm terminates

4.4 ALGORITHM EXPLANATION

Processors are first sorted by increasing value of C_k (i.e., decreasing value of communication link bandwidth). The first two processors are selected and the optimal (σ_a, σ_c) pair (the one with the lowest processing time for the two processors) is determined by solving the linear program defined by the constraints (3.10) to (3.13) four times — once for each permutation of the allocation and collection sequence. The next processor in the sequence is added in the next iteration and the linear program is solved again. The new processor can be interleaved at any position in (σ_a, σ_c), but with an additional constraint that the relative positions of processors already determined are maintained. In any iteration, if processor k is allocated zero load, then the algorithm terminates and does not proceed to the next iteration with $k + 1$ processors. As the optimal sequence pair is obtained by solving linear programs at each step, the load distribution α corresponding to the sequence pair is simultaneously obtained as an artifact of the solution to the linear program. Thus, though the emphasis is on determining the optimal sequence pair, in reality, it is not just the sequence pair but the 3-tuple $(\sigma_a, \sigma_c, \alpha)$ that is determined.

Example 4.7. If the optimal sequences at the end of the first iteration are $\sigma_a^2 = \{1, 2\}$ and $\sigma_c^2 = \{2, 1\}$, then in the second iteration, the set of possible allocation sequences is $\Sigma_a^3 = \{\{3, 1, 2\}, \{1, 3, 2\}, \{1, 2, 3\}\}$, and the set of possible collection sequences is $\Sigma_c^3 = \{\{3, 2, 1\}, \{2, 3, 1\}, \{2, 1, 3\}\}$. So $3 \times 3 = 9$ linear programs one for each sequence pair are solved and the optimal sequence pair (out of these nine pairs) is determined.

4.5 FINDING CANDIDATE SEQUENCES

The important part of the ITERLP algorithm is finding the set of possible allocation and collection sequences in every iteration. This set is a subset of the symmetric group S_k for k processors. The procedure to determine the set of candidate sequences Σ_a^k of the allocation sequence for $k \geq 3$ processors is given below. The procedure for finding Σ_c^k is analogous.

procedure `find_candidate_sequence`

```
1: define candidate sequence set $\Sigma_a^k \leftarrow \emptyset$
2: find the symmetric group $S_k$
3: for $\forall s \in S_k$ do
4:     find set $\Lambda_{k-1}$ of $(k-1)$-subsets of $s$
5:     if $\exists \lambda \in \Lambda_{k-1}$ such that $\lambda = \sigma_a^{k-1}$ do
6:         $\Sigma_a^k \leftarrow \Sigma_a^k \bigcup s$
7:     end if
8: end for
9: return
```

The '$\bigcup$' operator signifies that the sequence s is appended to the set Σ_a^k to obtain the updated set Σ_a^k.

The procedure `find_candidate_sequence` is explained in Example 4.8 to find the allocation sequence candidates for $k = 3$.

Example 4.8. It is assumed that already $\sigma_a^2 = \{2, 1\}$ has been found optimal in the first iteration. $S_3 = \{\{1,2,3\},\{1,3,2\},\{2,1,3\},\{2,3,1\},\{3,1,2\},\{3,2,1\}\}$, the symmetric group for $k = 3$ is first obtained. Then start with $s = \{1, 2, 3\}$. The set of 2-subsets for this is $\Lambda_2 = \{\{1,2\},\{1,3\},\{2,3\}\}$. Because no element of Λ_2 is equal to σ_a^2, s cannot be a candidate sequence. Continuing in this manner, the first candidate sequence is obtained as $\{2, 1, 3\}$, because one of its 2-subsets, namely $\{2, 1\}$, is equal to the σ_a^2 obtained previously. Similarly, the sequences $\{2, 3, 1\}$ and $\{3, 2, 1\}$ are determined to be candidate sequences.

Table 4.1 Results for all algorithms for $\mathcal{C} = \{10, 15\}$, $\mathcal{E} = \{10, 10\}$, $\delta = 0.5$. In this case, ITERLP results agree with the optimal.

Algorithm	σ_a	σ_c	α	T
OPT	$\{1,2\}$	$\{1,2\}$	$\{0.625, 0.375\}$	18.4375
ITERLP	$\{1,2\}$	$\{1,2\}$	$\{0.625, 0.375\}$	18.4375
LIFOC	$\{1,2\}$	$\{2,1\}$	$\{0.765, 0.235\}$	19.1176
FIFOC	$\{1,2\}$	$\{1,2\}$	$\{0.625, 0.375\}$	18.4375

4.6 COMPLEXITY AND DISCUSSION

In every iteration of ITERLP for k processors, the order (sequence) of allocation and collection of $k-1$ processors is already fixed in the previous iteration. When a new processor k is added, the number possible positions at which it can be placed is given by the binomial coefficient $\binom{k}{k\text{-}1} = k$. Thus the procedure `find_candidate_sequence` always generates k candidate sequences as output for k processors, even though the symmetric group has $k!$ possible sequences. This implies that:

- For k processors, σ_a and σ_c are restricted to k permutations each.
- For k processors, k^2 linear programs have to be solved to find the optimal sequence pair (σ_a^k, σ_c^k).
- In the worst case, $\sum_{k=1}^{m} k^2 = O(m^3)$ linear programs have to be solved in ITERLP.

To compare performance with the *brute force* method of finding the optimal solution, ITERLP can find the solution for about 65 processors in 80 minutes on a Power Mac G5 with 2GB of memory. When m is increased to 100, it takes around 15 hours. Of course, this is much too expensive to be practically used for large values of m. However, it is found that ITERLP generates significantly better schedules than traditional algorithms (see Sect. 4.7) and it can be used as a benchmark to compare other heuristic algorithms.

ITERLP asserts that maintaining the order of processors found in the previous iteration in the current iteration generates schedules that are close to optimal. The following examples with three processors proves that this is not true in general.

Example 4.9. Let $\mathcal{C} = \{10, 15, 20\}$, $\mathcal{E} = \{10, 10, 1\}$, and $\delta = 0.5$. The results obtained for the different algorithms are given in Tables 4.1 and 4.2. Details of the algorithms used are given in Sect 4.7. It is observed that after the first iteration, the optimal sequences found by ITERLP are $\sigma_a^2 = \{1, 2\}$ and $\sigma_c^2 = \{1, 2\}$. In the second iteration, when processor p_3 is added, ITERLP returns the sequences as $\sigma_a^3 = \{1, 2, 3\}$ and $\sigma_c^3 = \{3, 1, 2\}$. However, the optimal sequences for the three processors obtained by brute force, are $\sigma_a^* = \{1, 2, 3\}$

Table 4.2 Results for all algorithms for $\mathcal{C} = \{10, 15, 20\}$, $\mathcal{E} = \{10, 10, 1\}$, $\delta = 0.5$. In this case, the collection sequence detected by ITERLP is different from optimal. Moreover, the addition of a third processor reverses the optimal order detected for two processors in Table 4.1.

Algorithm	σ_a	σ_c	α	T
OPT	$\{1,2,3\}$	$\{3,2,1\}$	$\{0.7108, 0.2187, 0.0705\}$	17.7690
ITERLP	$\{1,2,3\}$	$\{3,1,2\}$	$\{0.6126, 0.3676, 0.0198\}$	18.0371
LIFOC	$\{1,2,3\}$	$\{3,2,1\}$	$\{0.7108, 0.2187, 0.0705\}$	17.7690
FIFOC	$\{1,2,3\}$	$\{1,2,3\}$	$\{0.6061, 0.3636, 0.0303\}$	18.1818

and $\sigma_c^* = \{3, 2, 1\}$. That is, the optimal collection sequence for the first two processors is reversed by the addition of the third processor.

Example 4.10. Another interesting example occurs when $\mathcal{C} = \{10, 15, 20\}, \mathcal{E} = \{5, 10, 15\}$, and $\delta = 0.5$. The sequences found by ITERLP are: $\sigma_a^2 = \{1, 2\}, \sigma_c^2 = \{1, 2\}, \sigma_a^3 = \{1, 3, 2\}$ and $\sigma_c^3 = \{1, 2, 3\}$. The optimal schedule is $\sigma_a^* = \{1, 2, 3\}$ and $\sigma_c^* = \{2, 3, 1\}$.

In Example 4.10, as in Example 4.9, the order of collection sequence is reversed. As the optimal collection sequence is not included in the set of candidate collection sequences for ITERLP, it causes ITERLP to detect the wrong allocation sequence too.

In this work, to date *a lot* of simulations have been carried out, and no set of values of $\mathcal{C}$, $\mathcal{E}$, and δ has been found that reverses the order of processors' allocation sequence in the optimal schedule obtained by brute force. The allocation sequence in the order of decreasing communication bandwidth is always found optimal. Given this fact, the above example in which ITERLP returns an incorrect allocation sequence, just goes to show the extreme dependence of the optimal solution on the values of the parameters $\mathcal{C}$, $\mathcal{E}$, and δ. Even though a fairly large number of sequences are evaluated in ITERLP, a small perturbation is sufficient for a wrong solution to be detected.

4.7 SIMULATION RESULTS AND ANALYSIS

On open networks, it is usual for processors to have wide variation in values of E_k and C_k [45]. In a network $\mathcal{H}$, it is possible that the ratios $\min \mathcal{E} : \max \mathcal{E}$ and $\min \mathcal{C} : \max \mathcal{C}$ reach 1:100. Further, they can appear in any combination. For example, a fast processor may have a very slow network connection, while a processor with a fast link may be overloaded and not have enough computation speed. Along with system heterogeneity, it is important to verify the effect of the application (δ) on the algorithms. To rigorously test the performance of ITERLP, several simulations were performed with different ranges for E_k, C_k, and δ.

4.7.1 *Algorithm Variants for Comparison*

The performance of ITERLP was compared to three algorithms, viz. OPT, FIFOC, and LIFOC, which are explained below. In all, more than 300,000 simulation runs were carried out using parameter values that cover most situations observed in practice.

The globally optimal schedule OPT is obtained after evaluation of the linear program for all possible $(m!)^2$ permutations of (σ_a, σ_c). The MATLAB™ linear program solver `lin-prog` is used to determine the optimal solution to the linear program defined by constraints (3.10) to (3.13) for each permutation pair. The processing time for each pair is calculated, and the sequence pair and load distribution that results in the minimum processing time is selected as the OPT solution. This ensures that the minimal set of processors is used and the optimal processing time is found.

LIFOC and FIFOC heuristics are as follows. In FIFOC, processors are allocated load and result are collected in the order of decreasing communication link bandwidth of the processors. In LIFOC, load allocation is in the order of decreasing communication link bandwidth of the processors, while result collection is the reverse order of increasing communication link bandwidth of the processors. Example 4.11 shows how σ_a and σ_c are obtained for FIFOC and LIFOC.

Example 4.11. Let $\mathcal{C} = \{20, 10, 15\}$, $\mathcal{E} = \{5, 15, 10\}$. The processors are first sorted in the order of decreasing communication link bandwidth (i.e. by increasing value of C_k). The sorted processor numbers give the allocation sequence $\sigma_a = \{2, 3, 1\}$ for both FIFOC and LIFOC. For FIFOC, the result collection sequence, is the same as σ_a, i.e. $\sigma_c|_{FIFOC} = \{2, 3, 1\}$, and for LIFOC, the result collection sequence is the reverse of σ_a, i.e. $\sigma_c|_{LIFOC} = \{1, 3, 2\}$.

For FIFOC, the sequence pair (σ_a, σ_c) so obtained, along with the sets $\mathcal{C}$ and $\mathcal{E}$, and δ, are used to construct the linear program defined by constraints (3.10) to (3.13), which is passed to the linear program solver `linprog`, that determines the optimal FIFOC solution. For LIFOC, using the transformation explained in [17], the optimal solution is found by using the closed form equations given in [85].

Preliminary simulations for other heuristic algorithms, viz. FIFO, LIFO, FIFOE, LIFOE, and SUMCE, revealed such large errors in favor of ITERLP, that it was decided not to pursue them further. The solutions to FIFO and LIFO are calculated similar to FIFOC and LIFOC except for the fact that the processors are not initially sorted. FIFOE and LIFOE distribute load fractions in the order of decreasing computation speed (i.e., increasing value of computation parameter, E_k). SUMCE distributes and collects load fractions in the order of increasing value of the sum $C_k + E_k + \delta C_k$ (equivalent to sorting by the sum $C_k + E_k$).

Table 4.3 Parameter values used for ITERLP simulations. There are 25 cases with different values of unit communication and computation times. In each case, the intervals are uniformly sampled to generate the sets $\mathcal{C}$ and $\mathcal{E}$. The intervals cover a wide range of parameter values such that all combinations of slow and fast communication and computation speeds are covered. The maximum min to max ratio is $1 : 100$. The intervals not only change in range (ratio), but also in the absolute value.

Case	$C_k \in$	$E_k \in$	Case	$C_k \in$	$E_k \in$
1	[1,10]	[1,10]	14	[10,100]	[1,100]
2	[1,10]	[10,100]	15	[10,100]	[10,1000]
3	[1,10]	[100,1000]	16	[10,1000]	[1,10]
4	[1,10]	[1,100]	17	[10,1000]	[10,100]
5	[1,10]	[10,1000]	18	[10,1000]	[100,1000]
6	[1,100]	[1,10]	19	[10,1000]	[1,100]
7	[1,100]	[10,100]	20	[10,1000]	[10,1000]
8	[1,100]	[100,1000]	21	[100,1000]	[1,10]
9	[1,100]	[1,100]	22	[100,1000]	[10,100]
10	[1,100]	[10,1000]	23	[100,1000]	[100,1000]
11	[10,100]	[1,10]	24	[100,1000]	[1,100]
12	[10,100]	[10,100]	25	[100,1000]	[10,1000]
13	[10,100]	[100,1000]			

4.7.2 *Simulation Method*

Simulations were carried out for $m = 4, 5$ and $\delta = 0.2, 0.5, 0.8$. For each variant algorithm, viz. OPT, LIFOC, FIFOC, and ITERLP, at each value of m and δ, 100 simulation runs were carried out for the 25 cases in Table 4.3. The values of E_k and C_k were obtained by sampling continuous uniform distributions in the regions specified in Table 4.3. The total processing time for each variant algorithm, T_{VAR}, was calculated in each run.

For example, Fig. 4.3 shows the execution times normalized with respect to T_{OPT} for $m = 4$, $\delta = 0.2$, case number 9. The solid line indicates the performance of ITERLP. As can be observed, ITERLP has the best performance, followed by LIFOC and FIFOC.

Fig. 4.3 also distinctly shows the dependence of processing time on the system parameters. To quantify the performance of the algorithms, the percentage deviation from the optimal processing time for each variant in each of the 25 cases was calculated as:

$$\Delta T_{\text{VAR}} = \frac{T_{\text{VAR}} - T_{\text{OPT}}}{T_{\text{OPT}}} * 100\% \tag{4.1}$$

Mean deviation from optimal, $\langle \Delta T_{\text{VAR}} \rangle$, for each variant was calculated by averaging ΔT_{VAR} over 100 simulation runs. Values of $\langle \Delta T_{\text{VAR}} \rangle$ were then plotted.

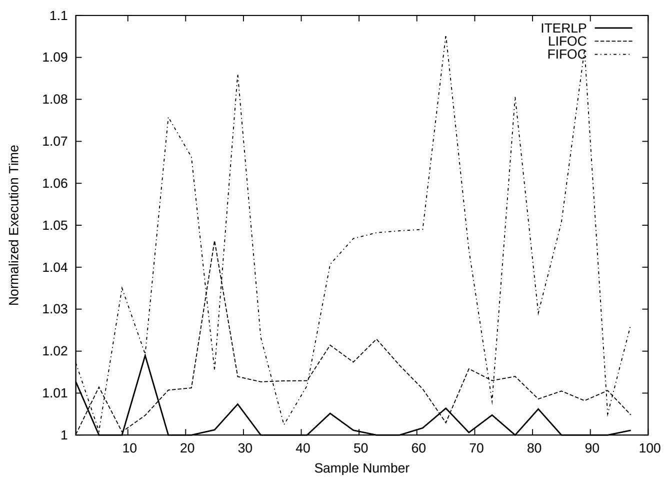


Figure 4.3 Execution time normalized w.r.t. optimal for $m = 4, \delta = 0.2$, Case 9. The 100 results are sub-sampled by 4 for plotting. A value of 1 means the execution time is equal to the optimal time. ITERLP has the best performance, followed by LIFOC and FIFOC. This figure appears in [46].

4.7.3 Result Plots and Analysis

The plot of $\langle \Delta T \rangle$ for $m = 4, \delta = 0.2$ is shown in Fig. 4.4. It can be observed that ITERLP consistently outperforms FIFOC and LIFOC in all the cases. As the value of δ increases, it is observed that the performance of LIFOC and ITERLP becomes very similar, while the error of FIFOC increases, as the plot for $m = 5$, $\delta = 0.8$ in Fig. 4.5 shows. Not only does the performance become similar, but also it gets very close to optimal. It can be concluded that for heterogeneous systems, where result collection time is large (comparable to the load allocation time), the performance of LIFOC and ITERLP is almost equal and optimal.

For intermediate values of δ, the performance of ITERLP is moderately better than LIFOC, and largely better than FIFOC as seen in Fig. 4.6 for $m = 5, \delta = 0.5$.

Though the algorithms show a clear dependence on the value of δ, the reason for the variation in performance can only be hypothesized at this juncture. In the case of LIFOC for example, when δ is large ($\delta \gg 1$), and especially when $\delta \to +\infty$, the load allocation and result collection looks similar to the case when $\delta = 0$, only in the reverse. LIFOC is still optimal in this case (if it was optimal earlier), while for FIFOC it would be the worst possible sequence. For the case when $\delta = 1$, by using the schedule transformation explained in [17], it can be seen that LIFOC processes exactly twice the amount of load in half the time, as would be processed if there would not have been any result collection phase. No such statement can be made about FIFOC.

Table 4.4 gives the maximum values of $\langle \Delta T \rangle$ for FIFOC, the case numbers when they occur, and the corresponding values of $\langle \Delta T \rangle$ for ITERLP for those cases. Table 4.5 gives

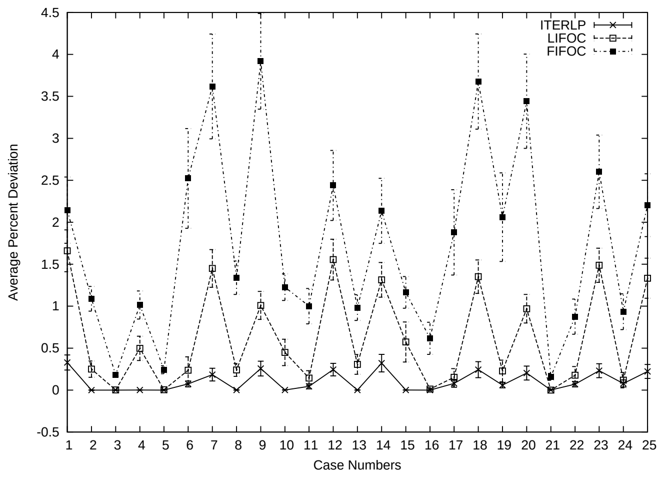


Figure 4.4 Average percent deviation $\langle \Delta T \rangle$ for $m = 4$, $\delta = 0.2$. ITERLP has the lowest deviation followed by LIFOC and FIFOC. This figure appears in [46].

Table 4.4 Maximum $\langle \Delta T \rangle$ of FIFOC. FIFOC error increases with the increase in m and δ. ITERLP error is virtually unchanged.

m	$\delta = 0.2$			$\delta = 0.5$			$\delta = 0.8$		
	FIFOC	case	ITERLP	FIFOC	case	ITERLP	FIFOC	case	ITERLP
4	3.91	9	0.25	10.29	7	0.24	12.31	18	0.35
5	4.40	7	0.32	10.96	7	0.35	14.33	7	0.36

Table 4.5 Maximum $\langle \Delta T \rangle$ of LIFOC. LIFOC error decreases slightly with the increase in m and δ. ITERLP error is virtually unchanged. So ITERLP and LIFOC performance becomes similar.

m	$\delta = 0.2$			$\delta = 0.5$			$\delta = 0.8$		
	LIFOC	case	ITERLP	LIFOC	case	ITERLP	LIFOC	case	ITERLP
4	1.66	1	0.33	1.39	23	0.45	1.13	23	0.53
5	1.24	23	0.37	1.27	4	0.08	1.63	4	0.12

similar values for LIFOC. The value of $\langle \Delta T \rangle$ for FIFOC for $m = 4$, $\delta = 0.5$, case number 7, is 43 times that of ITERLP for that case. Similarly, $\langle \Delta T \rangle$ for LIFOC for $m = 5$, $\delta = 0.8$, case number 4, is 13.5 times that of ITERLP for that case. That is, ITERLP generates good schedules for cases that cause FIFOC and LIFOC to perform poorly.

The maximum values of $\langle \Delta T \rangle$ of ITERLP and the case numbers when they occur are given in Table 4.6. The maximum $\langle \Delta T \rangle$ of ITERLP is 0.68%, and it occurs at case number 1 of $m = 5$, $\delta = 0.5$. It is observed that the error remains below 1% irrespective of the value of m or δ.

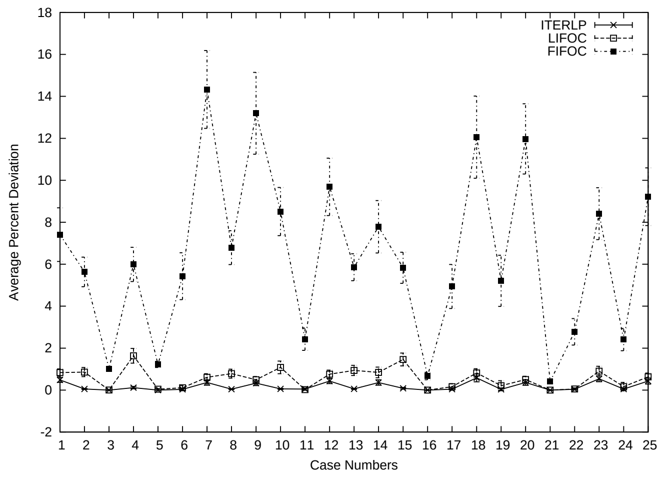


Figure 4.5 Average percent deviation $\langle \Delta T \rangle$ for $m = 5, \delta = 0.8$. ITERLP has the lowest deviation followed by LIFOC and FIFOC. LIFOC performance is almost the same as ITERLP while FIFOC error is quite large. This figure appears in [46].

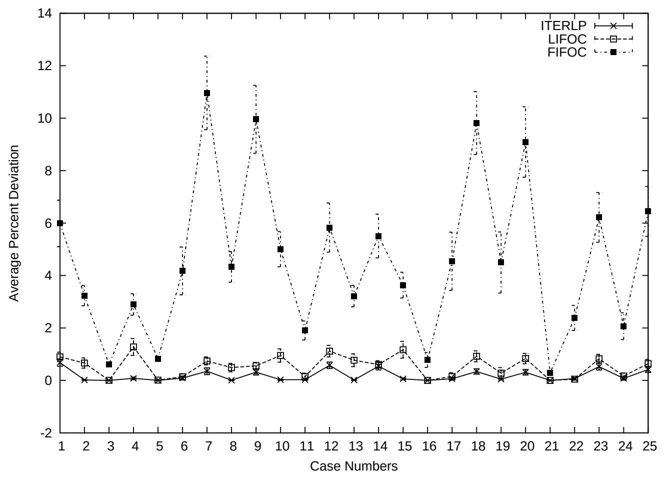


Figure 4.6 Average percent deviation $\langle \Delta T \rangle$ for $m = 5, \delta = 0.5$. ITERLP has the lowest deviation followed by LIFOC and FIFOC. LIFOC error is moderately improved while FIFOC performance deteriorates. This figure appears in [46].

To evaluate the performance of the algorithms with the increase in number of nodes, the processing times of FIFOC and LIFOC were compared with ITERLP. This is because, OPT cannot be practically carried out beyond $m = 5$. 100 simulation runs were carried out for $m = 10, 20, 30$, $\delta = 0.2, 0.5, 0.8$ for each of the 25 cases listed in Table 4.3.

As the number of processors and the value of δ increase, the performance of ITERLP and LIFOC becomes very similar, while there is an increase in the error of FIFOC. The 95% confidence interval bars indicate that the $\langle \Delta T \rangle$ of FIFOC with respect to ITERLP varies

Table 4.6 Maximum $\langle \Delta T \rangle$ of ITERLP. ITERLP error is virtually unchanged with increase in m and δ. FIFOC error increases, while LIFOC error slightly decreases with the increase in m and δ.

m	$\delta = 0.2$				$\delta = 0.5$				$\delta = 0.8$			
	ITERLP	case	LIFOC	FIFOC	ITERLP	case	LIFOC	FIFOC	ITERLP	case	LIFOC	FIFOC
4	0.33	1	1.66	2.14	0.45	23	1.39	5.81	0.53	23	1.13	7.35
5	0.51	12	1.13	2.41	0.68	1	0.90	5.99	0.58	18	0.82	12.06

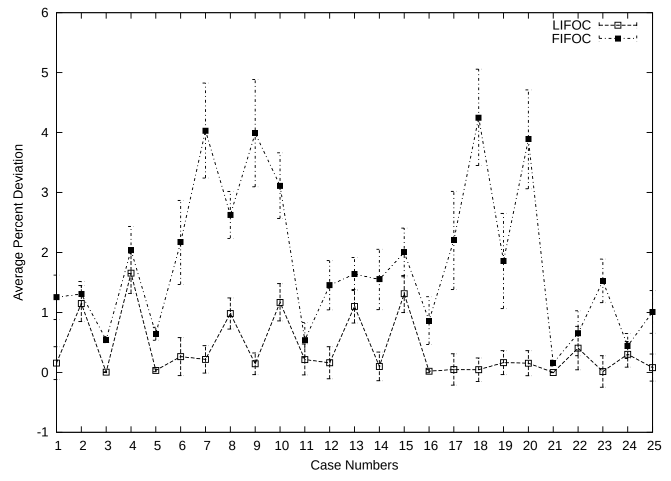


Figure 4.7 Average percent deviation $\langle \Delta T \rangle$ with respect to ITERLP at $m = 10, \delta = 0.2$. With increased m but low value of δ, ITERLP performance is much better than LIFOC and FIFOC. This figure appears in [46].

widely. The progression of performance is clearly reflected in Figs. 4.7 to 4.9 that plot the values of $\langle \Delta T \rangle$ with respect to ITERLP for (m, δ) pairs $(10, 0.2)$, $(20, 0.5)$, and $(30, 0.8)$ respectively.

However, at small values of δ, ITERLP performs better than both LIFOC and FIFOC even with large number of processors as can be seen in Fig. 4.10 that plots the value of $\langle \Delta T \rangle$ with respect to ITERLP for $m = 30, \delta = 0.2$.

4.8 SUMMARY

Obtaining an optimal solution to DLSRCHETS by enumerating all possible permutations of the allocation and collection sequences is impractical for more than six slave processors. LIFO and FIFO are two of the simplest possible schedules for DLSRCHETS. However, they suffer from a few problems:

- It is not specified how to obtain the optimal number of processors to use. In practice

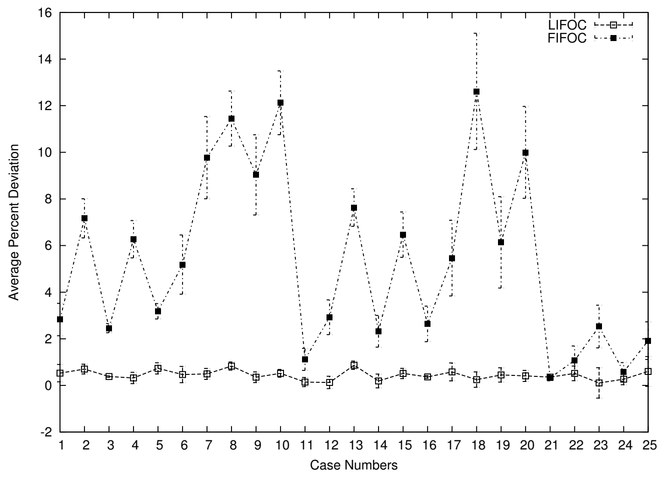


Figure 4.8 Average percent deviation $\langle \Delta T \rangle$ with respect to ITERLP at $m = 20, \delta = 0.5$. With increased value of δ, LIFOC performance improves but is still moderately worse than ITERLP, but FIFOC performance degrades noticeably. This figure appears in [46].

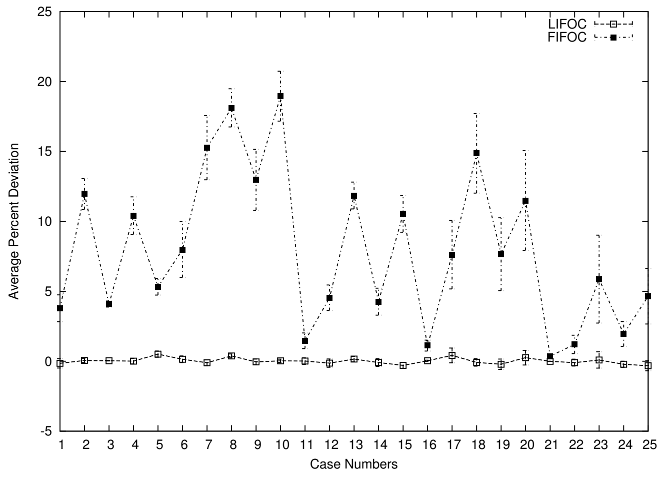


Figure 4.9 Average percent deviation $\langle \Delta T \rangle$ with respect to ITERLP at $m = 30, \delta = 0.8$. With even higher values of m and δ, LIFOC performance is almost equal to that of ITERLP, while FIFOC error is very large. This figure appears in [46].

one would resort to techniques such as *thresholding* (e.g., all processors for which the ratio of load allocated to the largest load fraction exceeds 1:100 will be dropped), but the algorithm itself does not include a way to determine this.

- It is known that LIFO and FIFO are not always optimal. Thus, there always exists a possibility that some other sequence is optimal.

- It is not clear when LIFO or FIFO is to be used. Previous studies have indicated that

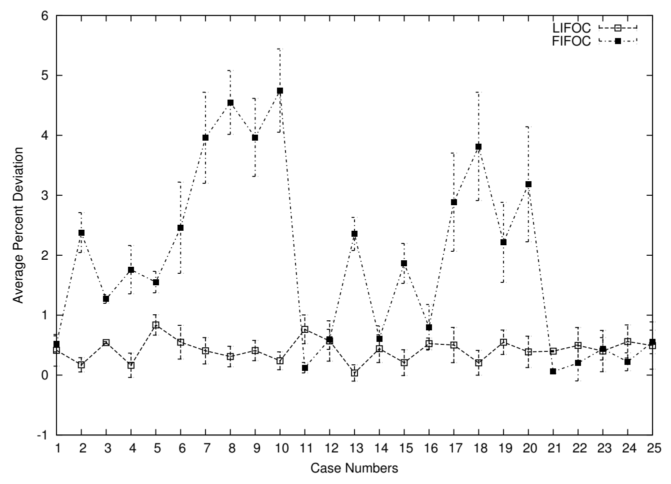


Figure 4.10 Average percent deviation $\langle \Delta T \rangle$ with respect to ITERLP at $m = 30, \delta = 0.2$. With high value of m and low value of δ, ITERLP performance is better than that of LIFOC and FIFOC. This figure appears in [46].

LIFO is better for heterogeneous systems while FIFO is better for more homogeneous environments. But there are no clear guidelines as to what constitutes heterogeneity.

Thus there is a need for a polynomial time algorithm that is robust to heterogeneity, that specifies the number of processors to use, and that generates good schedules.

As the *brute force* approach is not viable, one has to resort to a heuristic and prune the number of solutions. The proposed ITERLP algorithm reduces the number of possible allocation and collection sequences to m each instead of the usual $m!$. The idea is to generate a piecewise optimal solution by adding one processor at a time to the set of available processors. At the same time, the number of possible permutations is constrained by limiting it to only those permutations that do not change the order of processors already determined.

The computation cost of ITERLP is still quite high — in the worst case $O(m^3)$ linear programs have to be solved. The simulations show that ITERLP performance is much better than LIFO and FIFO over a wide range of parameter values. The performance of the algorithm is quite stable; schedules generated by ITERLP have execution time close to the optimal in most of the cases. Thus even though computation cost is high, it allows comparison of other heuristic algorithms.

The ITERLP algorithm forms a solution to the DLSRCHETS problem especially when it is impossible to find an optimal solution for large number of processors.

CHAPTER 5

ANALYSIS OF TWO-SLAVE SYSTEM

5.1 INTRODUCTION

The ITERLP algorithm presented in Chapter 4 generates good schedules for DLSRCHETS, but was seen to be computationally expensive for large number of processors. Another algorithm is needed that has comparable performance but is fast to compute.

The idea behind ITERLP was to construct a schedule in a piecewise manner by adding one processor at a time and finding the optimal schedule. The order of allocation and collection sequences found in one iteration were preserved in the next iteration. This limited the number of possible permutations of the allocation and collection sequences to k permutations for k processors. The selection of processors to add at every iteration is not random. There is a very definite order in which processors are added — in the order of decreasing communication link bandwidth. It was observed that this order is preserved in the allocation sequences in a large number of cases. This order of load allocation has been recommended by all DLS literature to date, those ignoring result collection [22, 24, 27, 28, 56, 59] as well as those considering result collection [1, 9, 16–18]. Considering these facts, the ITERLP algorithm could be potentially improved by:

- Maintaining the allocation sequence in the order of decreasing communication link bandwidth.
- Further restraining the number of possible collection order permutations.
- Finding the optimal schedule out of the possible schedules by some method other than by solving a linear program.
- Finding the load distribution without solving a linear program.

These points are the basis for the SPORT algorithm that is proposed in Chapter 6. One of the problems with ITERLP was that in iteration number k it still had to solve a linear program involving k processors. SPORT uses an ingenious way to construct a piecewise optimal solution using only two processors at a time. There are three important innovations in SPORT as listed below.

1. The crux of the SPORT technique is the *equivalent processor* and *equivalent system* concept. Basically, it is a way to reduce a two-slave system into its equivalent *virtual* single-processor system.

2. The second important concept involved is to find the optimal load distribution and schedule for two processors using simple if-then statements and closed-form formulae involving the system communication and computation parameters.

3. The third issue is that of limiting the possible collection sequence permutations. In ITERLP, the new processor added in an iteration number k can be placed in k different positions. In SPORT on the other hand, the new processor can be placed in only two positions — either at the beginning of the collection sequence or at the end. The position is decided by a simple comparison of parameter values.

It is clear that the two-slave system plays a pivotal role in the SPORT algorithm. This chapter introduces the various concepts mentioned above. First the derivation of the optimal schedule and load distribution for two processors is explained in Sections 5.2 to 5.6. Next the concept of equivalent processor is explained and the values of equivalent computation and communication parameters are derived in Section 5.7. The derivation of optimal schedule includes the description of the condition for optimality of LIFO and FIFO schedules, and the condition for the presence of idle time in a FIFO schedule for two processors. Section 5.12 shows how the condition for idle time can be extended to an arbitrary number of processors.

5.2 TWO-SLAVE SYSTEM CONFIGURATION

Traditional DLT assumes that load is always distributed to all the processors in a system, and that a processor is idle only up to the point where it starts receiving its load fraction. In the case of a heterogeneous system with result collection, these assumptions may not always be true.

Let the allocation sequence be represented by σ_a, and the collection sequence is represented by σ_c, both of which are permutations of the index set $K = \{1, \ldots, m\}$ of slave processors in the heterogeneous system $\mathcal{H}$.

To recapitulate, as shown in Fig. 5.1, for a sequence pair (σ_a, σ_c) and load distribution $\alpha = \{\alpha_1, \ldots, \alpha_m\}$, a slave processor p_i,

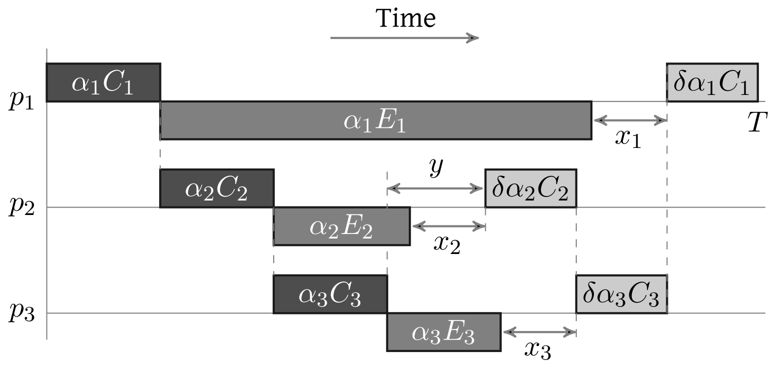


Figure 5.1 A schedule for $m = 3$ that satisfies the Feasible Schedule Theorem. Result collection begins only after the entire load is distributed. Each allocation and result collection phase follows its predecessor without delay. The computation phase of each processor follows its allocation phase without delay. Idle time may be present in each processor between the end of its computation phase and the start of the result collection phase.

- starts receiving its data at $t_i^{\text{recv}} = \sum\limits_{j=1}^{\sigma_a^{-1}[i]-1} \alpha_{\sigma_a[j]} C_{\sigma_a[j]}$
- starts execution at $t_i^{\text{exec}} = t_i^{\text{recv}} + \alpha_i C_i$
- stops execution at $t_i^{\text{stop}} = t_i^{\text{recv}} + \alpha_i C_i + \alpha_i E_i = t_i^{\text{exec}} + \alpha_i E_i$
- starts sending results at $t_i^{\text{send}} = T - \sum\limits_{j=\sigma_c^{-1}[i]}^{m} \delta\alpha_{\sigma_c[j]} C_{\sigma_c[j]}$
- may have idle time $x_i = t_i^{\text{send}} - t_i^{\text{stop}} \geq 0$

Thus idle time x_i may potentially be present in each processor p_i because it may have to wait for another processor to release the communication medium for result transfer. It has been proved that in the optimal solution to DLSRCHETS, $\forall i \in \{1 \dots m\}$, $x_i = 0$, if and only if $y > 0$, and that there exists a unique $x_i > 0$ if and only if $y = 0$, where y is the intervening time interval between the end of allocation phase of processor $\sigma_a[m]$ and the start of result collection from processor $\sigma_c[1]$. For the FIFO schedule in particular, processor $\sigma_a[m]$ can always be selected to have idle time when $y = 0$, i.e., in the FIFO schedule, $x_{\sigma_a[m]} > 0$ if and only if $y = 0$. In the LIFO schedule, since $y > 0$ always, no processor has idle time, i.e., $\forall i \in \{1 \dots m\}$, $x_i = 0$ always [16–18].

In the general case considered in this paper, for a pair (σ_a, σ_c), the solution to the linear program defined by (3.10) to (3.13) is completely determined by the values of δ, $\mathcal{E}$, $\mathcal{C}$, and it is not possible at this stage to predict which processor is the one that has idle time in the optimal solution. In fact, it is possible that not all processors are allocated load

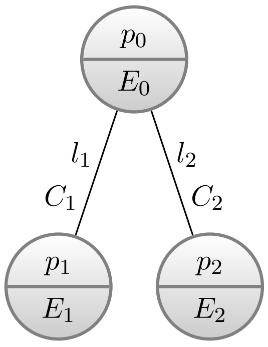


Figure 5.2 A heterogeneous two-slave system. Both slaves have different computation speeds and network bandwidths. The two-slave system is analyzed as it forms the basic building block in the SPORT algorithm.

in the optimal solution, in which case some processors are idle throughout. The processors that are allocated load for computation are known as the *participating processors* (or *participants*).

Thus any heuristic algorithm for DLSRCHETS must find both — the number of participants, and the load fractions allocated to them. In the next chapter, a polynomial time heuristic algorithm, SPORT is proposed that does this simultaneously. The foundation of the SPORT algorithm is laid first by analyzing the case of a system with two slave processors.

The heterogeneous system $\mathcal{H} = (\mathcal{P}, \mathcal{L})$ with $m = 2$ is shown in Fig. 5.2. It is defined by $\mathcal{P} = \{p_0, p_1, p_2\}$ and $\mathcal{L} = \{l_1, l_2\}$. The unit computation and communication times are defined by the sets $\mathcal{E} = \{E_1, E_2\}$, and $\mathcal{C} = \{C_1, C_2\}$. Without loss of generality, it is assumed that the total load to be processed available at the master is $\mathcal{J} = 1$. Also it is assumed that $C_1 \leq C_2$. No assumptions are possible regarding the relationship between E_1 and E_2, or $C_1 + E_1 + \delta C_1$ and $C_2 + E_2 + \delta C_2$ (or equivalently $C_1 + E_1$ and $C_2 + E_2$).

An important parameter, ρ_k, known as the *network parameter* is introduced, which indicates for a slave p_k, how fast (or slow) its computation parameter E_k is with respect to the communication parameter C_k of its network link:

$$\rho_k = \frac{E_k}{C_k} \qquad k = 1, \ldots, m \tag{5.1}$$

The master p_0 distributes the load $\mathcal{J}$ between the two slave processors p_1 and p_2 so as to minimize the processing time T. Depending on the values of δ, $\mathcal{E}$ and $\mathcal{C}$, there are three possibilities:

1. **Entire load is distributed to p_1 only.**

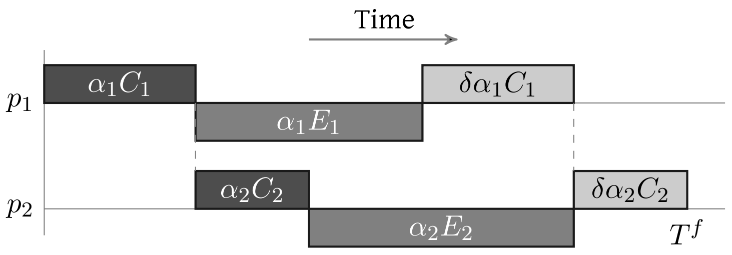


Figure 5.3 Timing diagram for Schedule f. This is the FIFO schedule without idle time for two slaves.

The total processing time is given by

$$T^1 = C_1 + E_1 + \delta C_1 = C_1(1 + \delta + \rho_1) \tag{5.2}$$

2. **Entire load is distributed to p_2 only.**

   The total processing time in this case is

$$T^2 = C_2 + E_2 + \delta C_2 = C_2(1 + \delta + \rho_2) \tag{5.3}$$

3. **Load is distributed to both p_1 and p_2.**

   It can be proved that as long as $C_1 \leq C_2$, only the schedules shown in Figs. 5.3, 5.4, and 5.5 can be optimal for a two-slave system. These schedules are the FIFO schedule, the LIFO schedule, and the FIFO schedule with idle time in processor p_2 respectively. There can be no LIFO schedule with idle time [16–18].

   In this thesis, these schedules are referred to as Schedule f, Schedule l, and Schedule g respectively. Superscripts f, l, and g are used to distinguish the three schedules. The equations for load fractions, processing times, and the conditions for optimality of Schedules f, l, and g are derived in brief in the following Sections 5.3, 5.4, and 5.5.

5.3 SCHEDULE f

This is the FIFO schedule without idle time for two slave processors. As mentioned above, it is assumed that $C_1 \leq C_2$. In this section, the equations defining Schedule f are derived.

From Fig. 5.3,

$$\alpha_1^f(E_1 + \delta C_1) = \alpha_2^f(C_2 + E_2) \tag{5.4}$$

Using $\alpha_1^f + \alpha_2^f = 1$ gives

$$\begin{aligned}\alpha_1^f &= \frac{C_2 + E_2}{E_1 + \delta C_1 + C_2 + E_2} \\ &= \frac{C_2(1+\rho_2)}{C_1(\delta+\rho_1) + C_2(1+\rho_2)} \\ &= \frac{C_2 r_2^f}{C_1 r_1^f + C_2 r_2^f} \end{aligned} \tag{5.5}$$

$$\begin{aligned}\alpha_2^f &= \frac{E_1 + \delta C_1}{E_1 + \delta C_1 + C_2 + E_2} \\ &= \frac{C_1(\delta+\rho_1)}{C_1(\delta+\rho_1) + C_2(1+\rho_2)} \\ &= \frac{C_1 r_1^f}{C_1 r_1^f + C_2 r_2^f} \end{aligned} \tag{5.6}$$

Where

$$\begin{aligned} r_1^f &= \delta + \rho_1 \\ r_2^f &= 1 + \rho_2 \end{aligned}$$

It is interesting to see α_1^f and α_2^f as weighted ratios of C_2 and C_1. The weights are functions of the network computation and communication parameters as well as the application under consideration. The processing time of Schedule f is

$$T^f = \alpha_1^f C_1(1+\delta+\rho_1) + \delta\alpha_2^f C_2$$

Using (5.5) and (5.6),

$$\begin{aligned} &= \frac{C_2 r_2^f C_1(1+r_1^f)}{C_1 r_1^f + C_2 r_2^f} + \frac{C_1 r_1^f \delta C_2}{C_1 r_1^f + C_2 r_2^f} \\ &= \frac{C_1 C_2}{C_1 r_1^f + C_2 r_2^f}(\delta r_1^f + r_2^f + r_1^f r_2^f) \\ &= \frac{C_1 C_2}{C_1 r_1^f + C_2 r_2^f}\Big((1+r_1^f)(\delta + r_2^f) - \delta\Big) \end{aligned} \tag{5.7}$$

It is advantageous to distribute load to the two processors in Schedule f instead of pro-

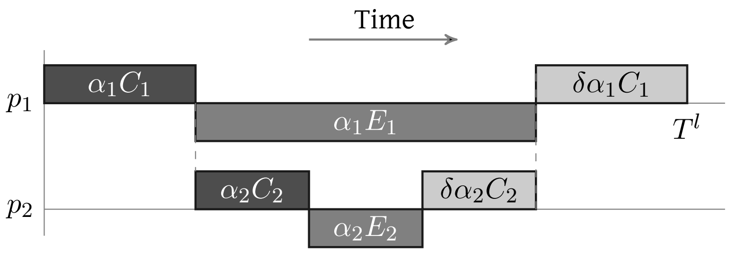


Figure 5.4 Timing diagram for Schedule l for the two-slave system. This is the LIFO schedule for two slaves.

cessing it entirely on p_1, if $T^f \leq T^1$. From (5.7),

$$\begin{aligned}
T^f \leq T^1 &\Leftrightarrow \frac{C_1C_2}{C_1r_1^f + C_2r_2^f}(\delta r_1^f + r_2^f + r_1^f r_2^f) \leq C_1(1 + r_1^f) \\
&\Leftrightarrow C_2(\delta r_1^f + r_2^f + r_1^f r_2^f) \leq (C_1r_1^f + C_2r_2^f)(1 + r_1^f) \\
&\Leftrightarrow \delta C_2 r_1^f \leq C_1 r_1^f(1 + r_1^f) \\
&\Leftrightarrow \delta C_2 \leq C_1(1 + r_1^f) \\
&\Leftrightarrow \delta C_2 \leq C_1(1 + \delta + \rho_1) \qquad (5.8)
\end{aligned}$$

Similarly, it is advantageous to distribute load to the two processors in Schedule f instead of processing it entirely on p_2, if $T^f \leq T^2$. Again using (5.7),

$$\begin{aligned}
T^f \leq T^2 &\Leftrightarrow \frac{C_1C_2}{C_1r_1^f + C_2r_2^f}(\delta r_1^f + r_2^f + r_1^f r_2^f) \leq C_2(\delta + r_2^f) \\
&\Leftrightarrow C_1(\delta r_1^f + r_2^f + r_1^f r_2^f) \leq (C_1r_1^f + C_2r_2^f)(\delta + r_2^f) \\
&\Leftrightarrow \delta C_1 r_2^f \leq C_2 r_2^f(\delta + r_2^f) \\
&\Leftrightarrow \delta C_1 \leq C_2(\delta + r_2^f) \\
&\Leftrightarrow \delta C_1 \leq C_2(1 + \delta + \rho_2) \qquad (5.9)
\end{aligned}$$

Equation (5.9) is always true for $C_1 \leq C_2$.

5.4 SCHEDULE l

This is the LIFO schedule for two slave processors. In this section, the equations defining Schedule l are derived.

From Fig. 5.4,

$$\alpha_1^l E_1 = \alpha_2^l(C_2 + E_2 + \delta C_2) \qquad (5.10)$$

Using $\alpha_1^l + \alpha_2^l = 1$ gives

$$\begin{aligned}\alpha_1^l &= \frac{C_2 + E_2 + \delta C_2}{E_1 + C_2 + E_2 + \delta C_2} \\ &= \frac{C_2(1+\delta+\rho_2)}{C_1\rho_1 + C_2(1+\delta+\rho_2)} \\ &= \frac{C_2 r_2^l}{C_1 r_1^l + C_2 r_2^l} \qquad (5.11)\end{aligned}$$

$$\begin{aligned}\alpha_2^l &= \frac{E_1}{E_1 + C_2 + E_2 + \delta C_2} \\ &= \frac{C_1\rho_1}{C_1\rho_1 + C_2(1+\delta+\rho_2)} \\ &= \frac{C_1 r_1^l}{C_1 r_1^l + C_2 r_2^l} \qquad (5.12)\end{aligned}$$

Where

$$r_1^l = \rho_1 \qquad (5.13)$$
$$r_2^l = 1 + \delta + \rho_2 \qquad (5.14)$$

We note that,

$$\begin{aligned} r_1^l &= r_1^f - \delta \qquad (5.15) \\ r_2^l &= r_2^f + \delta \qquad (5.16) \\ r_1^l + r_2^l &= r_1^f + r_2^f = 1 + \delta + \rho_1 + \rho_2 \qquad (5.17)\end{aligned}$$

The processing time of Schedule l is

$$T^l = \alpha_1^l C_1(1+\delta+\rho_1)$$

Using (5.11),

$$= \frac{C_1 C_2 r_2^l (1+\delta+r_1^l)}{C_1 r_1^l + C_2 r_2^l} \qquad (5.18)$$

Using (5.15) and (5.16)

$$= \frac{C_1 C_2 (r_2^f + \delta)(1+\delta+r_1^f - \delta)}{C_1(r_1^f - \delta) + C_2(r_2^f + \delta)}$$

$$= \frac{C_1C_2(1+r_1^f)(\delta+r_2^f)}{C_1r_1^f + C_2r_2^f + \delta(C_2-C_1)} \tag{5.19}$$

Load distribution in Schedule l is advantageous instead of distributing load entirely to p_1, if $T^l \leq T^1$. From (5.18),

$$\begin{aligned} T^l \leq T^1 &\Leftrightarrow \frac{C_1C_2r_2^l(1+\delta+r_1^l)}{C_1r_1^l + C_2r_2^l} \leq C_1(1+\delta+r_1^l) \\ &\Leftrightarrow C_2r_2^l \leq C_1r_1^l + C_2r_2^l \\ &\Leftrightarrow 0 \leq C_1\rho_1 = C_1\rho_1 = E_1 \end{aligned} \tag{5.20}$$

Equation (5.20) is always true. Similarly, Schedule l is advantageous as compared to processing entire load on p_2, if $T^l \leq T^2$. Again using (5.18),

$$\begin{aligned} T^l \leq T^2 &\Leftrightarrow \frac{C_1C_2r_2^l(1+\delta+r_1^l)}{C_1r_1^l + C_2r_2^l} \leq C_2r_2^l \\ &\Leftrightarrow C_1(1+\delta+r_1^l) \leq C_1r_1^l + C_2r_2^l \\ &\Leftrightarrow C_1(1+\delta) \leq C_2r_2^l \\ &\Leftrightarrow C_1(1+\delta) \leq C_2(1+\delta+\rho_2) \end{aligned} \tag{5.21}$$

Equation (5.21) is always true for $C_1 \leq C_2$.

To find the limiting condition for the optimality of Schedules f and l, the equations for T^f and T^l are compared. From (5.7) and (5.19),

$$\begin{aligned} T^f \leq T^l &\Leftrightarrow \frac{(1+r_1^f)(\delta+r_2^f)-\delta}{C_1r_1^f + C_2r_2^f} \leq \frac{(1+r_1^f)(\delta+r_2^f)}{C_1r_1^f + C_2r_2^f + \delta(C_2-C_1)} \\ &\Leftrightarrow -\delta(C_1r_1^f + C_2r_2^f) + \delta(C_2-C_1)\Big((1+r_1^f)(\delta+r_2^f)-\delta\Big) \leq 0 \\ &\Leftrightarrow (C_2-C_1)\Big((1+r_1^f)(\delta+r_2^f)-\delta\Big) \leq (C_1r_1^f + C_2r_2^f) \\ &\Leftrightarrow \frac{C_1C_2}{(C_1r_1^f + C_2r_2^f)}\Big((1+r_1^f)(\delta+r_2^f)-\delta\Big) \leq \frac{C_1C_2}{(C_2-C_1)} \\ &\Leftrightarrow T^f \leq \frac{C_1C_2}{(C_2-C_1)} \end{aligned} \tag{5.22}$$

Conversely, it can be easily proved that

$$T^l \geq \frac{C_1C_2}{(C_2-C_1)} \Leftrightarrow T^l \geq T^f \tag{5.23}$$

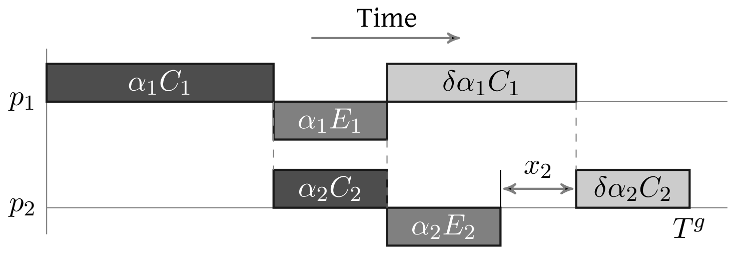


Figure 5.5 Timing diagram for Schedule g. This is the FIFO schedule with idle time for two slaves.

5.5 SCHEDULE g

This is the FIFO schedule with idle time for two slave processors. In this section, the equations defining Schedule g are derived.

From Fig. 5.5,

$$\alpha_1^g E_1 = \alpha_2^g C_2 \tag{5.24}$$

Using $\alpha_1^g + \alpha_2^g = 1$ gives

$$\alpha_1^g = \frac{C_2}{E_1 + C_2} = \frac{C_2}{C_1\rho_1 + C_2} \tag{5.25}$$
$$\alpha_2^g = \frac{E_1}{E_1 + C_2} = \frac{C_1\rho_1}{C_1\rho_1 + C_2} \tag{5.26}$$

The processing time of Schedule g is

$$T^g = \alpha_1^g C_1(1 + \delta + \rho_1) + \delta\alpha_2^g C_2$$

Using (5.25) and (5.26),

$$= \frac{C_1 C_2(1 + \delta + \rho_1)}{C_1\rho_1 + C_2} + \frac{\delta\rho_1 C_1 C_2}{C_1\rho_1 + C_2}$$
$$= \frac{C_1 C_2}{C_1\rho_1 + C_2}(1 + \delta)(1 + \rho_1) \tag{5.27}$$

Idle time occurs in processor p_2 (i.e., $x_2 \geq 0$), only when

$$\alpha_2^g E_2 \leq \delta\alpha_1^g C_1 \tag{5.28}$$

From (5.25) and (5.26),

$$\begin{aligned}\alpha_2^g E_2 \leq \delta\alpha_1^g C_1 &\Leftrightarrow \frac{C_1\rho_1}{C_1\rho_1 + C_2}\rho_2 C_2 \leq \frac{C_2}{C_1\rho_1 + C_2}\delta C_1 \\ &\Leftrightarrow \rho_1\rho_2 \leq \delta \end{aligned} \tag{5.29}$$

This is one of the most interesting conditions regarding idle time, derived for the first time in DLT. The condition is appealing because of its simplicity; in the way it summarizes the complex relationship between the presence of idle time in a two slave network with the system computation and communication parameters and the application under consideration. Further confirmation of this condition is obtained in Sect. 5.7.

It is advantageous to distribute load to the two processors in Schedule g instead of processing it entirely on p_1, if $T^g \leq T^1$. From (5.27),

$$\begin{aligned}T^g \leq T^1 &\Leftrightarrow \frac{C_1C_2}{C_1\rho_1 + C_2}(1+\delta)(1+\rho_1) \leq C_1(1+\delta+\rho_1) \\ &\Leftrightarrow C_2(1+\delta)(1+\rho_1) \leq (C_1\rho_1 + C_2)(1+\delta+\rho_1) \\ &\Leftrightarrow C_2(1+\delta)\rho_1 \leq C_1\rho_1(1+\delta+\rho_1) + C_2\rho_1 \\ &\Leftrightarrow \delta C_2 \leq C_1(1+\delta+\rho_1)\end{aligned} \tag{5.30}$$

Similarly, it is advantageous to distribute load to the two processors in Schedule g instead of processing it entirely on p_2, if $T^g \leq T^2$. Again using (5.27),

$$\begin{aligned}T^g \leq T^2 &\Leftrightarrow \frac{C_1C_2}{C_1\rho_1 + C_2}(1+\delta)(1+\rho_1) \leq C_2(1+\delta+\rho_2) \\ &\Leftrightarrow C_1(1+\delta)(1+\rho_1) \leq (C_1\rho_1 + C_2)(1+\delta+\rho_2) \\ &\Leftrightarrow C_1(1+\delta)(1+\rho_1) - C_1\rho_1(1+\delta+\rho_2) \leq C_2(1+\delta+\rho_2) \\ &\Leftrightarrow C_1(1+\delta) - C_1\rho_1\rho_2 \leq C_2(1+\delta+\rho_2) \\ &\Leftrightarrow C_1(1+\delta-\rho_1\rho_2) \leq C_2(1+\delta+\rho_2)\end{aligned} \tag{5.31}$$

Equation (5.31) is always true for $C_1 \leq C_2$.

To find the limiting condition for optimality of Schedules g and l, the equations for T^g and T^l are compared. From (5.27) and (5.18),

$$\begin{aligned}T^g \leq T^l &\Leftrightarrow \frac{C_1C_2}{C_1\rho_1 + C_2}(1+\delta)(1+\rho_1) \leq \frac{C_1C_2}{C_1r_1^l + C_2r_2^l}(1+\delta+r_1^l)r_2^l \\ &\Leftrightarrow \frac{(1+r_1^l)(r_2^l-\rho_2)}{C_1r_1^l + C_2} \leq \frac{(1+\delta+r_1^l)r_2^l}{C_1r_1^l + C_2r_2^l} \\ &\Leftrightarrow -C_1r_1^l(1+r_1^l)\rho_2 + C_2r_2^l(1+r_1^l)(r_2^l-\rho_2) \leq C_1r_1^l\delta r_2^l + C_2r_2^l(1+\delta+r_1^l)\end{aligned}$$

$$\Leftrightarrow C_2 r_2^l \Big((1+r_1^l)(r_2^l-\rho_2)-(1+\delta+r_1^l)\Big) \le C_1 r_1^l \Big(\delta r_2^l + (1+r_1^l)\rho_2\Big)$$

Using (5.13),

$$\Leftrightarrow C_2 r_2^l \Big((1+\rho_1)(1+\delta)-(1+\delta+\rho_1)\Big) \le C_1 \rho_1 \Big(\delta r_2^l + (1+\rho_1)\rho_2\Big)$$
$$\Leftrightarrow C_2 r_2^l \delta \rho_1 \le C_1 \rho_1 (\delta r_2^l + \rho_2 + \rho_1\rho_2)$$

Using (5.14),

$$\Leftrightarrow \delta C_2(1+\delta+\rho_2) \le C_1 \Big(\delta(1+\delta+\rho_2) + \rho_2 + \rho_1\rho_2\Big)$$
$$\Leftrightarrow C_2 \le C_1 \Big(1 + \frac{(1+\rho_1)\rho_2}{\delta(1+\delta+\rho_2)}\Big) \tag{5.32}$$

5.6 OPTIMAL SCHEDULE IN TWO-SLAVE SYSTEM

A few lemmas to determine the optimal schedule for a two-slave system are now stated.

Lemma 5.1. *It is always advantageous to distribute the load to both the processors, rather than execute it on the individual processors (for the system model under consideration).*

Proof. From (5.8), (5.9), (5.20), (5.21), (5.30), and (5.31), it can be concluded that:

1. $\delta C_2 \le C_1(1+\delta+\rho_1) \;\Rightarrow\; T^f \le T^1$
2. $C_1 \le C_2 \;\Rightarrow\; T^f \le T^2$
3. $E_1 \ge 0 \;\Rightarrow\; T^l \le T^1$
4. $C_1 \le C_2 \;\Rightarrow\; T^l \le T^2$
5. $\delta C_2 \le C_1(1+\delta+\rho_1) \;\Rightarrow\; T^g \le T^1$
6. $C_1 \le C_2 \;\Rightarrow\; T^g \le T^2$

By assumption, $C_1 \le C_2$. Hence, from points 2, 4, and 6 above, execution time of Schedules f, l, and g is always smaller than T^2.

By definition, $E_1 > 0$. From points 1, 3, and 5 above, as long as $\delta C_2 \le C_1(1+\delta+\rho_1)$, execution time of Schedules f, l, and g is less than T^1.

Finally, if $\delta C_2 > C_1(1+\delta+\rho_1)$, then T^f and T^g are greater than T^1, but since T^l is always less than T^1, load can be distributed in Schedule l to reduce the processing time.

Thus, it is always advantageous to distribute load to two processors instead of one under the system model under consideration. ■

Lemma 5.1 is important because it helps SPORT determine the number of participants in a general schedule. Details are given in Sect. 6.3.

From Lemma 5.1, if $\delta C_2 \leq C_1(1+\delta+\rho_1)$, then any of the Schedules f, g, or l could be optimal. The limiting condition between Schedule f and Schedule g is stated in the following lemma.

Lemma 5.2 (Idle Indicator Lemma). *$\rho_1\rho_2 \leq \delta$ is a necessary and sufficient condition to indicate the presence of idle time in the FIFO schedule (i.e. Schedule g).*

Proof. If the values of δ, $\mathcal{E}$, and $\mathcal{C}$, are such that they necessitate the presence of idle time in a FIFO schedule, then the schedule can be reduced to Schedule g as shown in Fig. 5.5.

In that case, idle time in processor p_2 occurs *only when* $\alpha_2^g E_2 \leq \delta\alpha_1^g C_1$. From (5.29), this condition reduces to $\rho_1\rho_2 \leq \delta$. ■

The simplicity of the condition to detect the presence of idle time in the FIFO schedule is both pleasing and surprising, and has been derived for the first time ever. Further confirmation of this condition is obtained in Sect. 5.7.

The following theorem can now be stated.

Theorem 5.1 (Optimal Schedule Theorem). *The optimal schedule for a two-slave system can be found as follows:*

1. *If $\delta C_2 > C_1(1+\delta+\rho_1)$, then Schedule l is optimal.*
2. *If $\delta C_2 \leq C_1(1+\delta+\rho_1)$, and both* (5.29) *and* (5.32) *hold, then Schedule g is optimal. Else if* (5.32) *does not hold, then Schedule l is optimal.*
3. *If $\delta C_2 \leq C_1(1+\delta+\rho_1)$,* (5.29) *does not hold, and condition* (5.22) *holds, then Schedule f is optimal. Else if* (5.22) *does not hold, then Schedule l is optimal.*

Proof. The proof follows directly from Lemmas 5.1 and 5.2, and (5.22) and (5.32). ■

All the conditions use only the data provided in the definition of the problem, with the exception of (5.22) which requires the computation of T^f.

It can be argued that the optimal schedule can be determined by directly computing the values of T^f, T^l, and T^g using (5.7), (5.18), and (5.27) respectively. While this fact cannot be denied, it defeats the ultimate purpose of this research, which is to identify

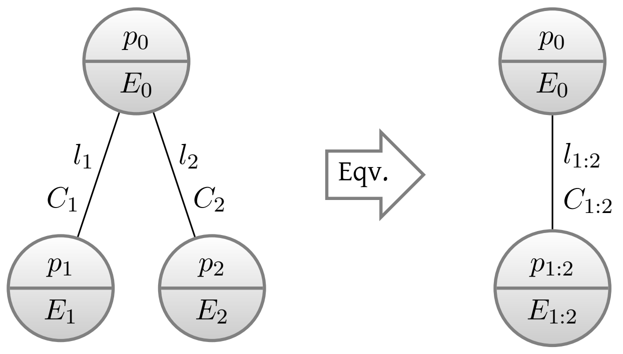


Figure 5.6 The concept of equivalent processor for the heterogeneous two-slave system. The two processors p_1 and p_2 are replaced by an equivalent virtual processor $p_{1:2}$. The two network links l_1 and l_2 are replaced by an equivalent virtual link $l_{1:2}$. As far as the master p_0 is concerned, there is no difference in the time it takes for the equivalent processor to execute a task.

relationships between the system parameters that influence the optimality of different schedules.

Another very interesting insight into the problem is provided by (5.22). The value $C_1C_2/(C_2 - C_1)$ forms a limiting condition between the optimality of Schedules f and l. It can be seen that as long as T^f is smaller than this value, it is also smaller than T^l.

As the network links become homogeneous, the difference $(C_2 - C_1)$ becomes small, and Schedule f is likely to be optimal because T^f would be less than the large value $C_1C_2/(C_2 - C_1)$. On the other hand, as the network links become heterogeneous, the value $C_1C_2/(C_2 - C_1)$ becomes small, and Schedule l would tend to be optimal because T^f can easily exceed this small value.

Rosenberg reached a similar conclusion with the help of simulations [77]. However this condition is analytically derived for the first time in DLT literature.

Once the optimal schedule (i.e., σ_a^* and σ_c^*) is known, it is simple to calculate the optimal load distribution α^* using the equations in Sects. 5.3, 5.4, and 5.5.

The optimal solution to DLSRCHETS, $(\sigma_a^*, \sigma_c^*, \alpha^*)$, for a system with two slave processors is a function of the system parameters and the application under consideration, because of which, no particular sequence of allocation and collection can be defined *a priori* as the optimal sequence. The optimal solution can only be determined once all the parameters become known.

5.7 THE CONCEPT OF EQUIVALENT PROCESSOR

To extend the above result to the general case with m slave processors, the concept of an *equivalent processor* is introduced. Consider the system in Fig. 5.6. The processors p_1 and

p_2 are replaced by a single equivalent processor $p_{1:2}$ with computation parameter $E_{1:2}$, connected to the root by an equivalent link $l_{1:2}$ with communication parameter $C_{1:2}$. The resulting system is called the *equivalent system* and the resulting schedule is known as the *equivalent schedule*. The values of the parameters for the three equivalent schedules are defined below.

If the initial load distribution is $\alpha = \{\alpha_1, \alpha_2\}$, and the processing time is T, then the equivalent system satisfies the following properties:

- The load processed by $p_{1:2}$ is $\alpha_{1:2} = \alpha_1 + \alpha_2 = 1$.
- The processing time is unchanged and equal to T.
- The time spent in load distribution and result collection is unchanged, i.e., for all three schedules,
  - $\alpha_{1:2}C_{1:2} = \alpha_1 C_1 + \alpha_2 C_2$, and
  - $\delta\alpha_{1:2}C_{1:2} = \delta\alpha_1 C_1 + \delta\alpha_2 C_2$.
- The time spent in load computation is equal to the intervening time interval between the end of allocation phase and the start of result collection phase, i.e.,
  - For Schedule f, $\alpha_{1:2}E^f_{1:2} = \alpha_1 E_1 - \alpha_2 C_2 = \alpha_2 E_2 - \delta\alpha_1 C_1$.
  - For Schedule l, $\alpha_{1:2}E^l_{1:2} = \alpha_2 E_2 = \alpha_1 E_1 - \alpha_2 C_2 - \delta\alpha_2 C_2$.
  - For Schedule g, $\alpha_{1:2}E^g_{1:2} = 0$.

5.8 EQUIVALENT PROCESSOR FOR SCHEDULE f

In Fig. 5.7, the top half shows Schedule f for the two processors in the original system, while the bottom half shows the corresponding equivalent Schedule f for the equivalent processor. In the equivalent Schedule f, the total communication time remains the same as the original two processors. The equivalent computation time is equal to the interval between the end of allocation to p_2 and the start of result collection from p_1.

To derive the equation for equivalent processor of Schedule f, algebraic manipulation of (5.7) gives

$$T^f_{1:2} = \frac{C_1C_2}{C_1r^f_1 + C_2r^f_2}\Big(r^f_1 + r^f_2 + \delta r^f_1 + \delta r^f_2 + r^f_1 r^f_2 - r^f_1 - \delta r^f_2 + \delta - \delta\Big)$$
$$= \frac{C_1C_2(r^f_1 + r^f_2)}{C_1r^f_1 + C_2r^f_2} + \frac{\delta C_1C_2(r^f_1 + r^f_2)}{C_1r^f_1 + C_2r^f_2} + \frac{C_1C_2\Big((r^f_1 - \delta)(r^f_2 - 1) - \delta\Big)}{C_1r^f_1 + C_2r^f_2}$$

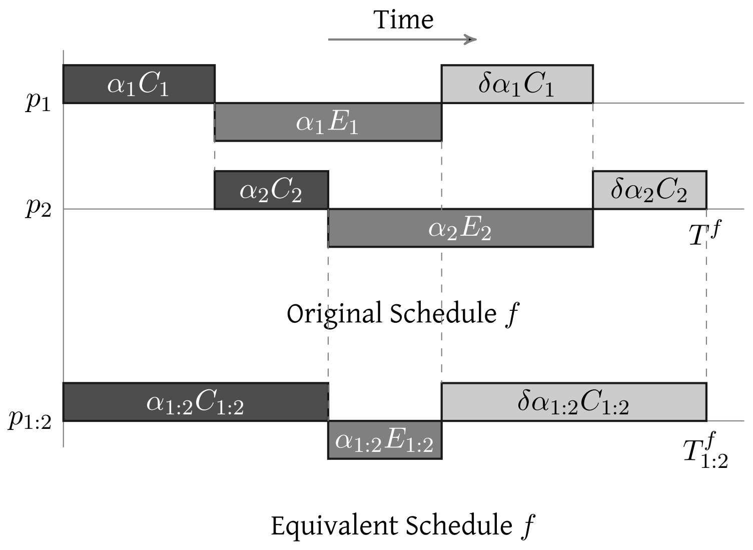


Figure 5.7 Equivalent processor in Schedule f. The total communication time remains the same as the original two processors. The equivalent computation time is equal to the interval between the end of allocation to p_2 and the start of result collection from p_1.

$$= \frac{C_1C_2(r_1^f + r_2^f)}{C_1r_1^f + C_2r_2^f} + \frac{C_1C_2(\rho_1\rho_2 - \delta)}{C_1r_1^f + C_2r_2^f} + \frac{\delta C_1C_2(r_1^f + r_2^f)}{C_1r_1^f + C_2r_2^f} \tag{5.33}$$

5.9 EQUIVALENT PROCESSOR FOR SCHEDULE l

Fig. 5.8 shows the timing diagram for equivalent Schedule l. As for Schedule f, the upper half shows the original schedule and the lower half shows the equivalent schedule. In the equivalent Schedule l, the total communication time remains the same as the original two processors. The equivalent computation time is equal to the computation time of p_2.

To derive the equation for equivalent processor for Schedule l, algebraic manipulation of (5.19) gives

$$\begin{aligned} T^l_{1:2} &= \frac{C_1C_2}{C_1r_1^f + C_2r_2^f + \delta(C_2 - C_1)}\left(r_1^f + r_2^f + \delta r_1^f + \delta r_2^f + r_1^f r_2^f - r_1^f - \delta r_2^f + \delta\right) \\ &= \frac{C_1C_2(r_1^f + r_2^f)}{C_1r_1^f + C_2r_2^f + \delta(C_2 - C_1)} + \frac{C_1C_2(r_1^f - \delta)(r_2^f - 1)}{C_1r_1^f + C_2r_2^f + \delta(C_2 - C_1)} \\ &\quad + \frac{\delta C_1C_2(r_1^f + r_2^f)}{C_1r_1^f + C_2r_2^f + \delta(C_2 - C_1)} \end{aligned}$$

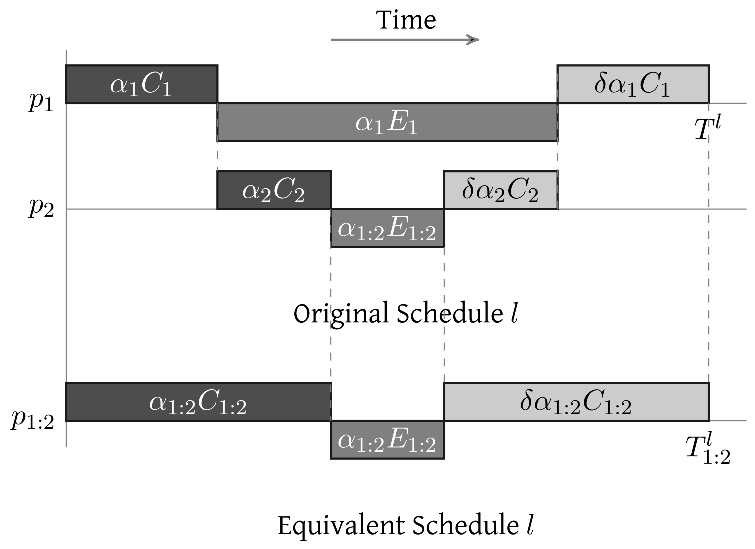


Figure 5.8 Equivalent processor in Schedule l. The total communication time remains the same as the original two processors. The equivalent computation time is equal to the computation time of p_2.

Using (5.15), (5.16), and (5.17),

$$= \frac{C_1C_2(r_1^l + r_2^l)}{C_1r_1^l + C_2r_2^l} + \frac{C_1C_2\rho_1\rho_2}{C_1r_1^l + C_2r_2^l} + \frac{\delta C_1C_2(r_1^l + r_2^l)}{C_1r_1^l + C_2r_2^l} \tag{5.34}$$

5.10 EQUIVALENT PROCESSOR FOR SCHEDULE g

Fig. 5.9 shows the timing diagram for equivalent Schedule g. In the equivalent Schedule g, the total communication time remains the same as the original two processors. The equivalent computation time is equal to zero as the result collection begins immediately after the allocation phase ends.

From (5.27), the processing time of Schedule g can be written as

$$T^g_{1:2} = \frac{C_1C_2(1+\rho_1)}{C_1\rho_1 + C_2} + \frac{\delta C_1C_2(1+\rho_1)}{C_1\rho_1 + C_2} \tag{5.35}$$

5.11 THE EQUIVALENT PROCESSOR THEOREM

This leads to the following theorem:

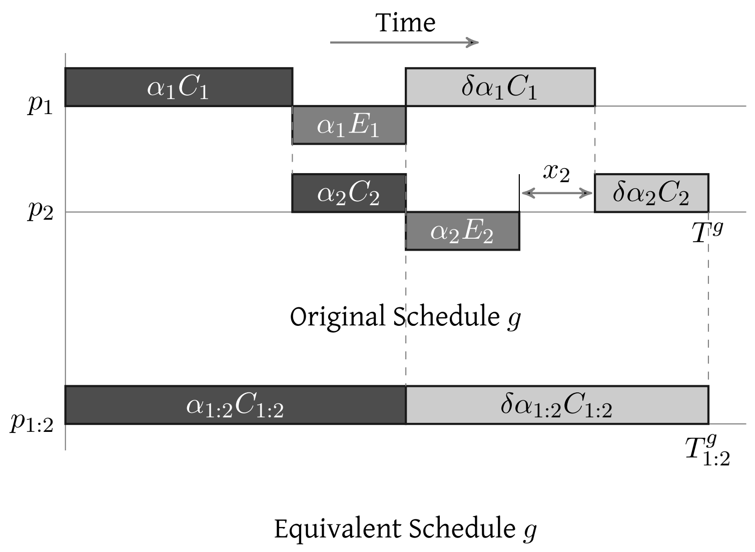


Figure 5.9 Equivalent processor in Schedule g. The total communication time remains the same as the original two processors. The equivalent computation time is equal to zero as the result collection begins immediately after the allocation phase ends.

Theorem 5.2 (Equivalent Processor Theorem)**.** *In a heterogeneous system $\mathcal{H}$ with $m = 2$, the two slave processors p_1 and p_2 can be replaced without affecting the processing time T, by a single (virtual) equivalent processor $p_{1:2}$ with equivalent parameters $C_{1:2}$ and $E_{1:2}$, such that $C_1 \leq C_{1:2} \leq C_2$ and $E_{1:2} \leq E_1, E_2$.*

Proof. From (5.33), the processing time of Schedule f can be written as,

$$T^f = \alpha_{1:2} C^f_{1:2} + \alpha_{1:2} E^f_{1:2} + \delta\alpha_{1:2} C^f_{1:2}$$

where

$$\alpha_{1:2} = \alpha^f_1 + \alpha^f_2 = 1$$

$$C^f_{1:2} = \frac{C_1 C_2 (r^f_1 + r^f_2)}{C_1 r^f_1 + C_2 r^f_2} \tag{5.36}$$

$$E^f_{1:2} = \frac{C_1 C_2 (\rho_1 \rho_2 - \delta)}{C_1 r^f_1 + C_2 r^f_2} \tag{5.37}$$

Similarly, from (5.34), the processing time of Schedule l can be written as,

$$T^l = \alpha_{1:2} C^l_{1:2} + \alpha_{1:2} E^l_{1:2} + \delta\alpha_{1:2} C^l_{1:2}$$

where

$$\begin{aligned} \alpha_{1:2} &= \alpha_1^l + \alpha_2^l = 1 \\ C_{1:2}^l &= \frac{C_1 C_2 (r_1^l + r_2^l)}{C_1 r_1^l + C_2 r_2^l} && (5.38) \\ E_{1:2}^l &= \frac{C_1 C_2 \rho_1 \rho_2}{C_1 r_1^l + C_2 r_2^l} && (5.39) \end{aligned}$$

From (5.35), the processing time of Schedule g can be written as

$$T^g = \alpha_{1:2} C_{1:2}^g + \alpha_{1:2} E_{1:2}^g + \delta \alpha_{1:2} C_{1:2}^g$$

where,

$$\begin{aligned} \alpha_{1:2} &= \alpha_1^g + \alpha_2^g = 1 \\ C_{1:2}^g &= \frac{C_1 C_2 (1 + \rho_1)}{C_1 \rho_1 + C_2} && (5.40) \\ E_{1:2}^g &= 0 && (5.41) \end{aligned}$$

It can be easily verified that these representations satisfy the properties of equivalent processor mentioned above.

For Schedule f, l, and g,

$$\begin{aligned} \alpha_{1:2} C_{1:2} &= \alpha_1 C_1 + \alpha_2 C_2, && (5.42) \\ \alpha_1 &= 1 - \alpha_2, && (5.43) \end{aligned}$$

and

$$\alpha_{1:2} = 1. \qquad (5.44)$$

Using (5.43) and (5.44) in (5.42), the equivalent communication parameter can be written as

$$C_{1:2} = C_1 + \alpha_2 (C_2 - C_1). \qquad (5.45)$$

Since by definition, $0 \leq \alpha_2 \leq 1$, it immediately follows that $C_1 \leq C_{1:2} \leq C_2$.

Similarly, from the definition of equivalent system,

$$E_{1:2}^f = \alpha_1 E_1 - \alpha_2 C_2$$

$$= \alpha_2 E_2 - \delta \alpha_1 C_1 \tag{5.46}$$

and

$$\begin{aligned} E^l_{1:2} &= \alpha_2 E_2 \\ &= \alpha_1 E_1 - \alpha_2 C_2 (1+\delta). \end{aligned} \tag{5.47}$$

Since $0 \leq \alpha_1, \alpha_2 \leq 1$, it follows from (5.46) and (5.47) that $E^f_{1:2} \leq E_1, E_2$ and $E^l_{1:2} \leq E_1, E_2$.

From (5.41), $E^g_{1:2} = 0$ implies that $E^g_{1:2} \leq E_1, E_2$, since by definition, $E_1, E_2 > 0$. ■

The equivalent processor enables replacement of two processors by a single processor with communication parameter with a value that lies between the values of communication parameters of the original two links. Because of this property, if the processors are arranged so that $C_1 \leq C_2 \leq \ldots \leq C_m$, and two processors are combined at a time sequentially starting from the fastest two, then the resultant equivalent processor does not disturb the order of the sequence. This property is exploited in the SPORT algorithm, which is described in the next chapter.

The equivalent processor for Schedule f provides additional confirmation of the condition for the presence of idle time in a FIFO schedule i.e. use of Schedule g (5.29). It is known that idle time can exist in a FIFO schedule only when the intervening time interval $y = 0$. According to the definition of equivalent processor, this interval corresponds to the equivalent computation capacity $E^f_{1:2}$. From (5.37), this value becomes zero only when $\rho_1\rho_2 - \delta = 0$. Thus, if $\rho_1\rho_2 < \delta$, then idle time must exist in the FIFO schedule.

5.12 EXTENDING THE EQUIVALENT PROCESSOR CONCEPT

Consider the heterogeneous system $\mathcal{H}$ with m processors. Two special case situations arise if one wishes to know:

- If load is distributed to the m processors in a FIFO schedule, will it have idle time?
- Can load be allocated to all processors in a FIFO schedule?
- If load is distributed to the m processors in a LIFO schedule, will it be faster than distributing it in a FIFO schedule?

Of course one wishes to know the answers without explicitly calculating the processing times and load distributions. Using the equivalent processor concept, these questions can be answered by simple calculations on the system parameters itself.

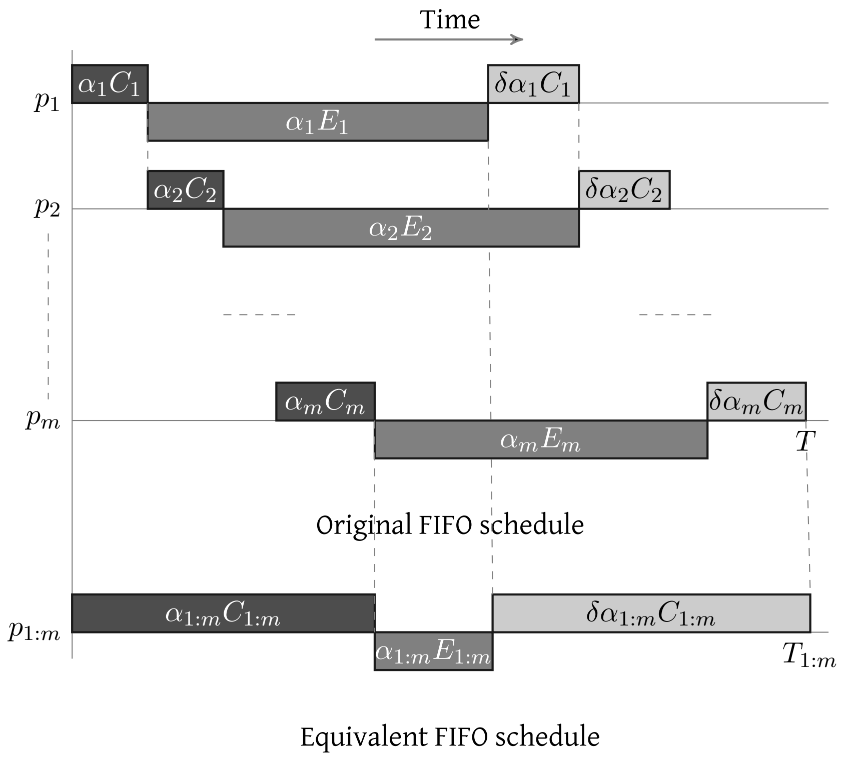


Figure 5.10 General FIFO schedule and its equivalent processor. The total communication time remains the same as the original m processors. The equivalent computation time is equal to the interval between the end of the allocation phase to p_m and the start of the result collection phase of p_1.

The closed-form formulae for the FIFO and LIFO schedules and their equivalent processors are first derived in brief.

5.12.1 The General FIFO Schedule

The closed-form formulae for load fractions of the general FIFO schedule can be obtained in any standard text on DLS. They are reproduced here for the sake of completeness, and so that they can be used in the formulae for the general FIFO equivalent processor.

From Fig. 5.10, for $k = 1, \ldots, m-1$,

$$\alpha_k E_k + \delta\alpha_k C_k = \alpha_{k+1} C_{k+1} + \alpha_{k+1} E_{k+1}$$

Which can be written as

$$\alpha_k = f_k\, \alpha_{k+1} \qquad k = 1, \ldots, m-1 \tag{5.48}$$

where

$$f_k = \left(\frac{C_{k+1} + E_{k+1}}{E_k + \delta C_k}\right) \qquad k = 1, \ldots, m-1 \tag{5.49}$$

Let $f_m = 1$. From the normalization equation (3.12),

$$\sum_{j=1}^{m} \alpha_j = 1 \tag{5.50}$$

The recursive equations (5.48) can be solved by expressing α_k $(k = 1, \ldots, m-1)$ in terms of α_m as

$$\alpha_k = \left(\prod_{j=k}^{m} f_j\right) \alpha_m \qquad k = 1, \ldots, m-1 \tag{5.51}$$

From (5.50) and (5.51), and using the fact that $f_m = 1$,

$$\alpha_m = \frac{1}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.52}$$

Using (5.52) in (5.51),

$$\alpha_k = \frac{\prod_{j=k}^{m} f_j}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \quad k = 1, \ldots, m-1 \tag{5.53}$$

The processing time is,

$$T^f_{1:m} = \sum_{k=1}^{m} \alpha_k C_k + \alpha_m E_m + \delta \alpha_m C_m \tag{5.54}$$

From (5.52) and (5.53),

$$T^f_{1:m} = \frac{\sum_{k=1}^{m} \left(\prod_{j=k}^{m} f_j\right) C_k + E_m + \delta C_m}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.55}$$

5.12.2 General FIFO Equivalent Processor

The set of processors $p_1, \ldots, p_m$ in figure 5.10 can be replaced by a single equivalent processor denoted by $p_{1:m}$ with parameter $E_{1:m}$ connected to the master by an equivalent link denoted by $l_{1:m}$ with parameter $C_{1:m}$. The equivalent system satisfies the following properties.

If the initial load distribution $\alpha = \{\alpha_1, \ldots, \alpha_m\}$ is such that the total processing time is T, then

- The processing time of the equivalent system with the new load distribution $\alpha_{1:m}$ is also T. The equivalent system now has one slave processor and the load processed by the equivalent processor is $\alpha_{1:m} = \alpha_1 + \cdots + \alpha_m$.
- The total delay in communicating the load fractions $\alpha_1, \ldots, \alpha_m$ from the master and the result fractions $\delta\alpha_1, \ldots, \delta\alpha_m$ to the master is the same as the delay in communicating the load $\alpha_{1:m}$ and $\delta\alpha_{1:m}$ in the equivalent system.
- The time taken by the equivalent processor to compute the load $\alpha_{1:m}$ is equal to the difference between the point when processor p_1 starts to send its result data to the master and the point at which processor p_m completes reception of its input load fraction.

From the conditions imposed on the equivalent system,

$$\alpha_{1:m} = \sum_{k=1}^{m} \alpha_k = \Big(\sum_{i=1}^{m} \prod_{j=i}^{m} f_j\Big)\alpha_m = 1 \tag{5.56}$$

and

$$\alpha_{1:m} C^f_{1:m} = \sum_{k=1}^{m} \alpha_k C_k \tag{5.57}$$

Using (5.51) and (5.56) in (5.57),

$$C^f_{1:m} = \frac{\sum_{k=1}^{m} \Big(\prod_{j=k}^{m} f_j\Big) C_k}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.58}$$

Now, by definition,

$$\alpha_{1:m} E^f_{1:m} = \alpha_m E_m - \sum_{k=1}^{m-1} \delta\alpha_k C_k \tag{5.59}$$

Again using (5.56) and (5.58) in (5.59)

$$E^f_{1:m} = \frac{E_m - \delta \sum_{k=1}^{m-1} \Big(\prod_{j=k}^{m} f_j\Big) C_k}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.60}$$

To find the range of values for $C^f_{1:m}$ and $E^f_{1:m}$, it is assumed that the processors $p_1, \ldots, p_{m-1}$ are represented by their equivalent processor $p_{1:m-1}$ with parameters $C^f_{1:m-1}$ and $E^f_{1:m-1}$,

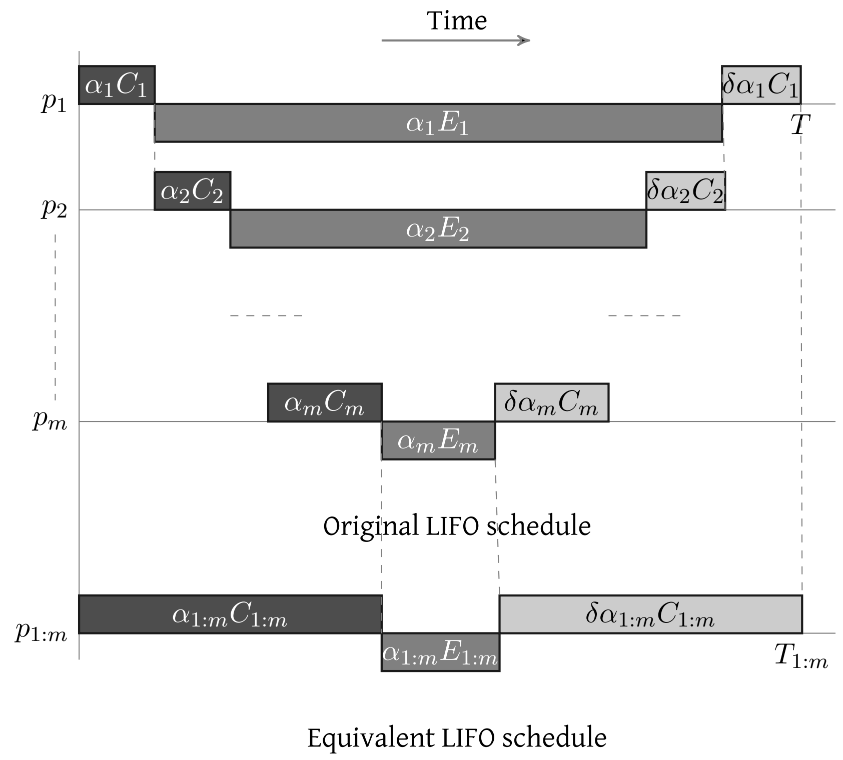


Figure 5.11 General LIFO schedule and its equivalent processor. The total communication time remains the same as the original m processors. The equivalent computation time is equal to the computation time of processor p_m.

the values of which are given by equations (5.58) and (5.60) respectively. By setting $p_1 \equiv p_{1:m-1}$ and $p_2 \equiv p_m$ in Theorem 5.2, if follows that,

$$C^f_{1:m-1} \leq C^f_{1:m} \leq C_m \tag{5.61}$$

and

$$E^f_{1:m} \leq E_1, \ldots, E_m \tag{5.62}$$

5.12.3 The General LIFO Schedule

The closed-form equations for load fractions for the general LIFO schedule are now derived. The derivation is based on similar lines as that for FIFO.

From Fig. 5.11, for $k = 1, \ldots, m-1$,

$$\alpha_k E_k = \alpha_{k+1} C_{k+1} + \alpha_{k+1} E_{k+1} + \delta\alpha_{k+1} C_{k+1}$$

or,

$$\alpha_k = f_k\, \alpha_{k+1} \qquad k = 1, \ldots, m-1 \tag{5.63}$$

where

$$f_k = \left(\frac{C_{k+1} + E_{k+1} + \delta C_{k+1}}{E_k}\right) \tag{5.64}$$

and $f_m = 1$. Proceeding similar to FIFO,

$$\alpha_m = \frac{1}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.65}$$

and

$$\alpha_k = \frac{\prod_{j=k}^{m} f_j}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \qquad k = 1, \ldots, m-1 \tag{5.66}$$

The processing time is given as,

$$\begin{aligned} T^l_{1:m} &= \alpha_1 C_1 + \alpha_1 E_1 + \delta \alpha_1 C_1 \\ &= \frac{\prod_{j=2}^{m} f_j (C_1 + E_1 + \delta C_1)}{1 + \sum_{i=1}^{m-1} \prod_{j=i}^{m} f_j} \end{aligned} \tag{5.67}$$

5.12.4 General LIFO Equivalent Processor

The conditions for equivalent processor for the LIFO schedule are similar to that of FIFO except for the fact that the time taken by the equivalent processor to compute the load $\alpha_{1:m}$ is equal to the time taken by processor p_m to compute its load fraction, α_m (refer figure 5.11). Similar to equations for FIFO schedule, it can be derived,

$$\alpha_{1:m} = \sum_{k=1}^{m} \alpha_k = \Big(\sum_{i=1}^{m-1} \prod_{j=i}^{m} f_j\Big)\alpha_m = 1 \tag{5.68}$$

$$C^l_{1:m} = \frac{\sum_{k=1}^{m} \Big(\prod_{j=k}^{m} f_j\Big) C_k}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.69}$$

Now, by definition,

$$\alpha_{1:m} E^l_{1:m} = \alpha_m E_m \tag{5.70}$$

Using (5.68) in (5.70)

$$E^l_{1:m} = \frac{E_m}{\sum_{i=1}^{m} \prod_{j=i}^{m} f_j} \tag{5.71}$$

Further, by proceeding in a similar way as schedule FIFO and using Theorem 5.2, it can be shown that

$$C^l_{1:m-1} \leq C^l_{1:m} \leq C_m \tag{5.72}$$

and

$$E^l_{1:m} < E_1, \ldots, E_m \tag{5.73}$$

5.12.5 Results Using General Equivalent Processor

Using the derivations above, the questions posed at the beginning of Section 5.12 can be answered.

Whether LIFO or FIFO is faster for a heterogeneous system $\mathcal{H}$ with m processors can easily be computed using (5.55) and (5.67).

Lemma 5.3. *$\rho^f_{1:m-1}\rho_m < \delta$ is a necessary and sufficient condition to indicate the presence of idle time in a general optimal FIFO schedule with m slave processors, where $\rho^f_{1:m-1}$ is the equivalent network parameter for the first $m-1$ processors in the allocation sequence.*

Proof. The conditions for optimality (5.22) or (5.23) and for the existence of idle time (5.29) can be easily extended to the general FIFO and LIFO schedules with an arbitrary number of processors using the equivalent processor concept. The conditions place no restriction on the processors that can be compared, and so one or both the processors may be virtual (equivalent) processors.

It is known that idle time in an optimal FIFO schedule can always be transferred to the last processor in the allocations sequence, i.e., the processor with the slowest communication link. This means that the first $m-1$ processors in case of a system with m processors have no idle time irrespective of whether idle time exists or not. Thus if these processors are collected together and represented by their equivalent parameters $C^f_{1:m-1}$ and $E^f_{1:m-1}$, then the condition for presence of idle time (5.29) reduces to simply checking if

$$\rho^f_{1:m-1}\rho_m < \delta. \tag{5.74}$$

When (5.74) is satisfied with an equality, the computation time of p_m is exactly equal to the result collection time of $p_{1:m-1}$. The intervening time interval between the end of load allocation of $p_{1:m-1}$ and the start of result collection of p_m is still zero. ∎

An interesting and important corollary results from Lemma 5.3. It enables to determine the number of processors to use in a general FIFO schedule as well as the SPORT algorithm.

Corollary 5.1. *In a heterogeneous system $\mathcal{H}$ with m slave processors, assume that the processors $p_1, \ldots, p_m$ are arranged such that $C_1 \leq C_2 \leq \ldots \leq C_m$. Then load is allocated to processors k and above, $k \in \{1, \ldots, m\}$, if and only if $\rho_{1:k-2}\rho_{k-1} > \delta$.*

Proof. Scan the processors from left to right, i.e., from the one with the fastest communication link to the slowest. Let p_k, $k \in \{1, \ldots, m\}$ be the first processor to satisfy the condition (5.74) with either an equality or inequality, i.e. $\rho_{1:k-1}\rho_k \leq \delta$.

It is known that when condition (5.74) is satisfied with either an equality or inequality, the intervening time interval y between the end of the allocation sequence and the start of result collection is zero. This means that processors p_{k+1} and above cannot be allocated load in the FIFO sequence.

Thus, equivalently, load is allocated to processor k and above, $k \in \{1, \ldots, m\}$, if and only if the condition $\rho^f_{1:k-2}\rho_{k-1} > \delta$ it true.

This condition is valid irrespective of whether the equivalent processor is obtained by combining processors in a FIFO schedule, or a LIFO schedule, or a combination of both. ∎

5.13 SUMMARY

This is arguably the most important chapter in the thesis, because it lays the foundation of the two-slave system that forms the basis for the SPORT algorithm. Several important concepts were introduced in this chapter, namely,

- The three types of possible optimal schedules in a two-slave system and the related derivations. With two slaves, it would seem natural to have four possible processor orderings. While the number of orderings is four, there are in fact *six* possible schedules — two of FIFO type, two of LIFO type, and two of FIFO type with idle time. Out of these six only three (one of each type) are possibly optimal because they are just images of each other. The three schedules were explored and formulae for the load fractions and processing times were derived.

- Derivation of optimal schedule for two processors using simple if-then clauses and closed-form equations. It is one of the objectives of this research to find relationships between system parameters and the optimal schedule. Thus the optimal schedule is determined by a series of if-then clauses that use processor computation and communication parameters only.

- The condition for optimality of the LIFO and FIFO schedules in a two-slave system. LIFO and FIFO type of schedules are the simplest possible for divisible load scheduling with result collection. A condition was derived to check which of the two is faster for a two-slave system. The result shows that this is independent of the computation speeds of the processors as well as the application under consideration, i.e., the size of result data as indicated by δ. Whether LIFO (resp. FIFO) is faster for a two-slave system depends only on the communication speeds of the processor links.

- The condition for the existence of idle time in a FIFO schedule. The simplicity of this condition is very appealing. It neatly shows a relationship between the computation and communication speeds of the two processors and the value of δ. Since $0 \leq \delta \leq 1$, at least one of the network parameters (ρ_k) should be less than one for idle time to exist in a schedule. That is, idle time may exist when the communication speed is lower than the computation speed of the processors.

- Equivalent processor for LIFO and FIFO schedules and related derivations. The equivalent processor enables the combination of two processors into a single virtual processor. When applied repeatedly to a system, the communication and computation capacity of the system can be easily summarized. Because of this summarization, when constructing a piecewise solution, at each step (iteration):

  (a) the optimal schedule has to be determined for only two processors, and

  (b) the possible collection sequence permutations are limited to only two.

  This in turn enables a huge reduction in the computation complexity of an iterative solution because the number of possibilities is constant.

- The extension of the equivalent processor concept to an arbitrary number of processors and its applications. The LIFO and FIFO solutions are special cases where the equivalent processor concept can be extended to an arbitrary number of processors. In case of a general schedule this is not always possible. Closed form equations were derived for the LIFO and FIFO case and it was shown how some of the results such as the condition for idle time in a two-slave system can be extended to a general system with m processors. As a corollary to the condition for idle time, the method to determine the number of processors to allocate load was derived.

In the next chapter all these concepts finally coalesce into the SPORT algorithm.

CHAPTER 6

THE SPORT ALGORITHM

6.1 INTRODUCTION

From the analysis in previous chapters, some desirable characteristics of an algorithm to solve DLSRCHETS are:

- Along with the allocation and collection sequences the algorithm should find:
  (a) the number of processors to use for computation (i.e. the number of *participants*), and
  (b) the load fractions to be allocated to the participants.
- As far as possible this should be done without solving a linear program, because solving a linear program is time consuming.
- The algorithm should limit the number of possible allocation and collection sequences to a few, potentially close to optimal permutations.
- The algorithm should be robust to system heterogeneity; it should provide good schedules for both homogeneous and heterogeneous types of systems.

The SPORT algorithm proposed in this chapter fulfills all of these requirements. For example,

- SPORT uses Theorem 5.1 to determine the optimal schedule for two processors.
- SPORT reduces two processors into single virtual processor using Theorem 5.2.
- Lemma 5.2 and Corollary 5.1 are used to find the number of participants in SPORT.
- SPORT determines the load fractions using a simple binary tree traversal.

- The allocation sequence is maintained in the order of decreasing communicaiton bandwidth as recommended by previous research results [1, 9, 16–18, 22, 24, 27, 28, 56, 59].
- The collection sequence is limited to two possible permutations in each iteration.
- Because SPORT builds a piecewise optimal schedule, it is able to adapt to changes in system parameters over a wide range of values.

The chapter is organized as follows. Section 6.2 details the proposed algorithm and Sections 6.3 and 6.4 give the explanation of the algorithm and discuss about the computation complexity. SPORT is tested using simulations over a wide range of system and application parameters in Sections 6.5 and 6.6. Finally, Section 6.7 summarizes the results presented in this chapter.

6.2 PROPOSED ALGORITHM

The proposed SPORT algorithm is as follows.

Algorithm 2 (SPORT)**.**

```
1:  arrange $p_1, \ldots, p_m$ such that $C_1 \le C_2 \le \ldots \le C_m$
2:  $\sigma_a \leftarrow 1, \sigma_c \leftarrow 1, \alpha_1 \leftarrow 1$
3:  for $k := 2$ to $m$ do
4:    $C_1 \leftarrow C_{1:k-1}, E_1 \leftarrow E_{1:k-1}, C_2 \leftarrow C_k, E_2 \leftarrow E_k$
5:    if $\delta C_2 > C_1(1 + \delta + \rho_1)$ then
6:      /* $T^l < T^f, T^g$, use Schedule $l$ */
7:      call schedule_lifo
8:    else
9:      /* Need to check other conditions */
10:     if $\rho_1 \rho_2 \le \delta$ then
11:       /* Possibility of idle time */
12:       if $C_2 \le C_1\Big(1 + \frac{(1+\rho_1)\rho_2}{\delta(1+\delta+\rho_2)}\Big)$ then
```

```
13:         /* $T^g < T^l$, use Schedule $g$ */
14:         call schedule_idle
15:         break for
16:       else
17:         /* $T^l < T^g$, use Schedule $l$ */
18:         call schedule_lifo
19:       end if
20:     else
21:       /* No idle time present */
22:       if $T^f \le \dfrac{C_1 C_2}{C_2 - C_1}$ then
23:         /* $T^f < T^l$, use Schedule $f$ */
24:         call schedule_fifo
25:       else
26:         /* $T^l < T^f$, use Schedule $l$ */
27:         call schedule_lifo
28:       end if
29:     end if
30:   end if
31: end for
32: $n \leftarrow$ numberOfProcessorsUsed
33: /* Update load fractions from stored values */
```

$$\alpha_k \leftarrow \begin{cases} \alpha_k \cdot \prod_{j=2}^{n} \alpha_{1:j} & \text{if } k = 1 \\ \alpha_k \cdot \prod_{j=k}^{n} \alpha_{1:j} & \text{if } k = 2, \ldots, n \end{cases}$$

```
34: $T \leftarrow C_{1:n} + E_{1:n} + \delta\, C_{1:n}$
```

The procedures in the algorithm are given below:

procedure `schedule_idle`

```
1: $\alpha_{1:k-1} \leftarrow \dfrac{C_2}{C_1\rho_1 + C_2}$
2: $\alpha_k \leftarrow \dfrac{C_1\rho_1}{C_1\rho_1 + C_2}$
3: /* Update sequences for FIFO */
4: $\sigma_a \leftarrow \{\sigma_a,\ k\}$
5: $\sigma_c \leftarrow \{\sigma_c,\ k\}$
6: /* Compute equivalent processor parameters */
7: $C_{1:k} \leftarrow \dfrac{C_1 C_2(1+\rho_1)}{C_1\rho_1 + C_2}$
8: $E_{1:k} \leftarrow 0$
9: numberOfProcessorsUsed $\leftarrow k$
10: return
```

procedure `schedule_lifo`

```
1: $r_1^l \leftarrow \rho_1$
2: $r_2^l \leftarrow 1 + \delta + \rho_2$
3: $\alpha_{1:k-1} \leftarrow \dfrac{C_2 r_2^l}{C_1 r_1^l + C_2 r_2^l}$
4: $\alpha_k \leftarrow \dfrac{C_1 r_1^l}{C_1 r_1^l + C_2 r_2^l}$
5: /* Update sequences for LIFO */
6: $\sigma_a \leftarrow \{\sigma_a,\ k\}$
7: $\sigma_c \leftarrow \{k,\ \sigma_c\}$
8: /* Compute equivalent processor parameters */
9: $C_{1:k} \leftarrow \dfrac{C_1 C_2(r_1^l + r_2^l)}{C_1 r_1^l + C_2 r_2^l}$
```

```
10: $E_{1:k} \leftarrow \dfrac{C_1 C_2 \rho_1 \rho_2}{C_1 r_1^l + C_2 r_2^l}$
11: numberOfProcessorsUsed $\leftarrow k$
12: **return**
```

procedure `schedule_fifo`

```
1: $r_1^f \leftarrow \delta + \rho_1$
2: $r_2^f \leftarrow 1 + \rho_2$
3: $\alpha_{1:k-1} \leftarrow \dfrac{C_2 r_2^f}{C_1 r_1^f + C_2 r_2^f}$
4: $\alpha_k \leftarrow \dfrac{C_1 r_1^f}{C_1 r_1^f + C_2 r_2^f}$
5: /* Update sequences for FIFO */
6: $\sigma_a \leftarrow \{\sigma_a,\ k\}$
7: $\sigma_c \leftarrow \{\sigma_c,\ k\}$
8: /* Compute equivalent processor parameters */
9: $C_{1:k} \leftarrow \dfrac{C_1 C_2 (r_1^f + r_2^f)}{C_1 r_1^f + C_2 r_2^f}$
10: $E_{1:k} \leftarrow \dfrac{C_1 C_2 (\rho_1 \rho_2 - \delta)}{C_1 r_1^f + C_2 r_2^f}$
11: numberOfProcessorsUsed $\leftarrow k$
12: **return**
```

The working of the algorithm is explained in the next section.

6.3 ALGORITHM EXPLANATION

At the start, the processors are arranged so that $C_1 \le C_2 \le \ldots \le C_m$, and two processors with the fastest communication links are selected. The optimal schedule and load distribution for the two processors are found according to Theorem 5.1. If Schedule f or l is found optimal, then the two processors are replaced by their equivalent processor. In either case, since $C_1 \le C_{1:2} \le C_2$, the ordering of the processors does not change. In

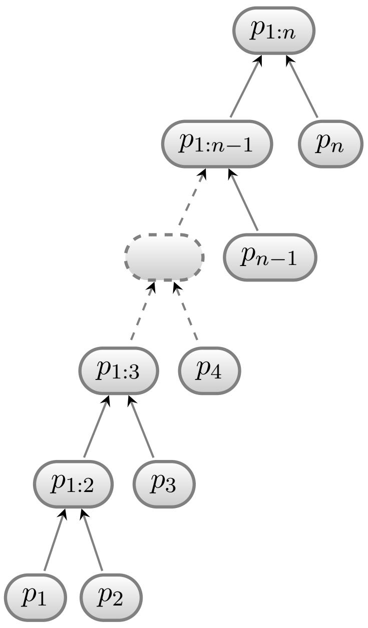


Figure 6.1 The building of SPORT solution. At each step only two processors are involved (the state space remains constant). The optimal schedule for two processors can be easily computed in constant time using simple if-then-else statements in Theorem 5.1.

the subsequent iteration, the equivalent processor and the processor with the next fastest communication link are selected and the steps are repeated until either all processors are used up, or Schedule g is found to be optimal. If Schedule g is found to be optimal in any iteration, then the algorithm exits after finding the load distribution for that iteration.

Example 6.1. An example of how the SPORT algorithm works for a network with $m = 3$ is given in Fig. 6.2. In Fig. 6.2, Schedule l is found to be optimal for processors p_1 and p_2, and Schedule f is optimal for their equivalent processor $p_{1:2}$ and the processor p_3. The resulting timing diagrams are as shown.

Now assume that this network has several more processors. Since SPORT adds processors one by one to the set of participants, Lemma 5.1 implies that the addition should be *greedy*, i.e. as many processors as possible should be used to minimize the processing time. If in the third iteration, Schedule g is found to be optimal for the processors $p_{1:3}$ and p_4, then in that case, the intervening time interval between the end of load allocation to p_4 and the start of result collection from p_2 would be zero. Since additional processors are always inserted (allocated load) within this interval, it would not be possible to allocate load to any more processors, and the algorithm exits. Thus Schedule g forms the logical termination criterion for the algorithm unless all processors in the network are used. When Schedule g is found to be the optimal schedule, all remaining processors are

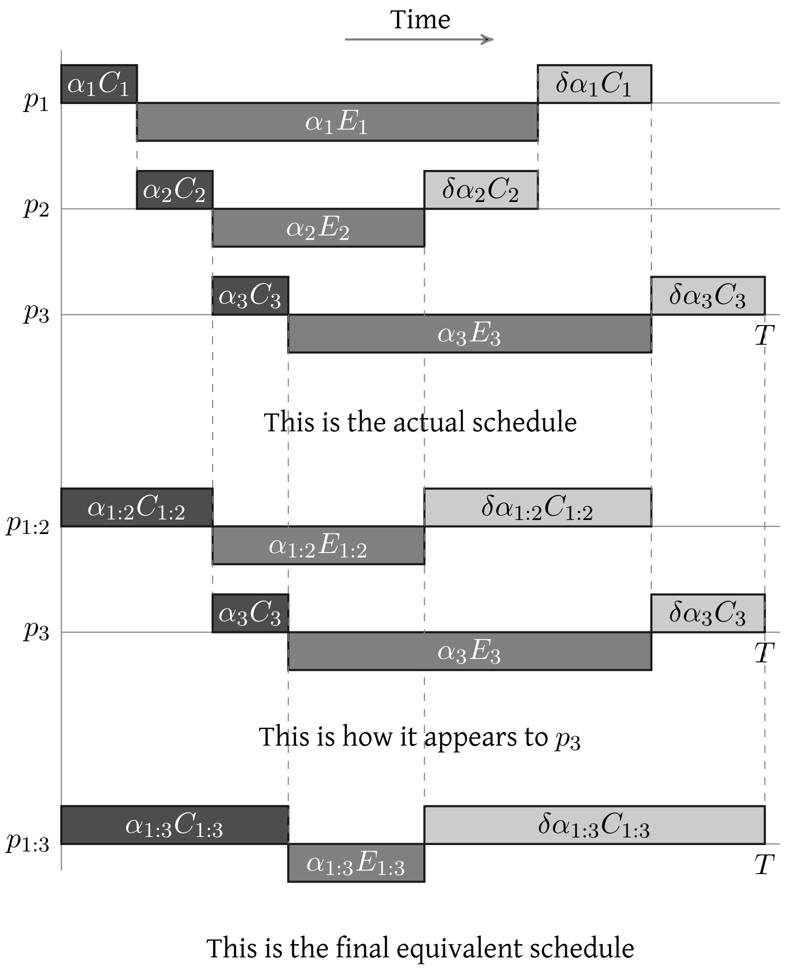


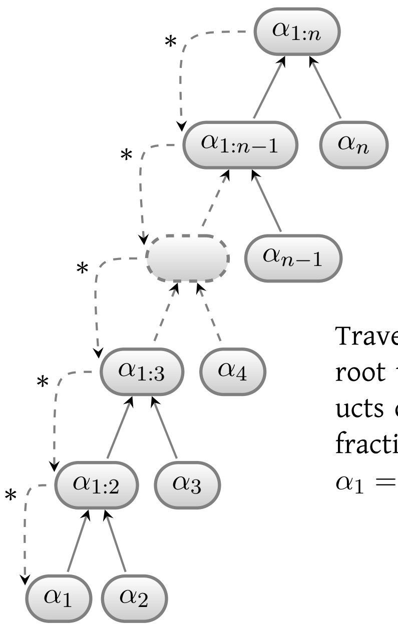
Figure 6.2 An example of SPORT algorithm for three processors. Schedule l is optimal for p_1 and p_2, and schedule f is optimal for their equivalent processor $p_{1:2}$ and p_3. p_3 *sees* only $p_{1:2}$. The final equivalent processor $p_{1:3}$ determines the makespan T.

allocated zero load, and their index in the allocation and collection sequence is assigned zero value.

The computation of the allocation and collection sequences is straightforward. The allocation sequence σ_a is maintained in the order of decreasing communication link bandwidth of the processors. Irrespective of the schedule found optimal in iteration k, k is always appended to σ_a. The collection sequence σ_c is constructed in the following manner:

- If Schedule f or g is found optimal in iteration k, k is appended to σ_c.
- If Schedule l is found optimal in iteration k, k is prepended to σ_c.

The calculation of load distribution to the processors occurs simultaneously with the



Traverse the binary tree from the root to the leaves, and take products of the values to compute load fractions. For example,
$\alpha_1 = \alpha_{1:n} \cdot \alpha_{1:n-1} \cdots \alpha_{1:2} \cdot \alpha'_1$

Figure 6.3 Calculating the load fractions in SPORT. α'_1 is the initial value of α_1. It is multiplied by the product term in (6.1) to get the final value of α_1. This is equivalent to traversing the binary tree from the root to the leaf nodes and taking the product of all nodes (values) encountered. This calculation can be implemented in $O(m)$ time by starting with α_m and storing the intermediate values.

search for the optimal schedule. As shown in Fig. 6.3, the algorithm creates a *one-sided binary tree* of load fractions. If the number of processors participating in the computation is n, $2 \le n \le m$, the root node of the binary tree is $\alpha_{1:n}$ and the leaf nodes represent the final load fractions allocated to the processors. The value of the root node need not be calculated as it is equal to one. The individual load fractions, α_k, are initially assigned value α'_k (say), and then updated at the end as:

$$\alpha_k = \begin{cases} \alpha'_k \cdot \prod_{j=2}^{n} \alpha_{1:j} & \text{if } k = 1 \\ \alpha'_k \cdot \prod_{j=k}^{n} \alpha_{1:j} & \text{if } k = 2, \dots, n \end{cases} \tag{6.1}$$

This is equivalent to traversing the binary tree from the root to each leaf node and taking the product of the nodes encountered (see Fig. 6.3). This calculation can be easily implemented in $O(m)$ time by starting with the computation of α_n, and storing the values of the product terms (i.e. $\prod \alpha_{1:j}$) for each processor and then using that value for the next processor.

Once the sequences (σ_a, σ_c) and load distribution α are found, calculating the processing time is straightforward. The processing time is simply the sum of the (equivalent) parameters of the equivalent processor $p_{1:n}$, i.e., $T = C_{1:n} + E_{1:n} + \delta\, C_{1:n}$.

6.4 COMPLEXITY AND DISCUSSION

In SPORT, defining the allocation sequence by sorting the values of C_k requires $O(m \log m)$ time, while finding the collection sequence and load distribution requires $O(m)$ time in the worst case. Thus, if sorted values of C_k are given, then the overall complexity of the algorithm is polynomial in m and is equal to $O(m)$.

The equivalent processor method works because:

- The equivalent processor maintains the positions of the processors in the allocation sequence in order of decreasing communication bandwidth (increasing value of communication parameter).
- Two processors in each iteration are kept immediately successive in the final ordering.
- All equations are linear. So irrespective of the final time interval T, the respective positions and load fractions are not changed.

It follows that if overheads are considered (i.e. computation and communication costs are affine functions of the size of data) in the system model, then the equivalent processor method may not work. However most applications of DLS where result collection is important, involve transfers of large volumes of data, and overheads can be safely ignored.

As noted earlier in Sect. 5.6, when result collection phase is considered along with heterogeneous networks, at this stage at least, it is not possible to *a priori* define any single sequence as *the* optimum sequence for allocation or collection. Nor is it possible to determine a criterion for the optimal number of processors to be used in the computation. The number of processors used and the optimal sequences depend on the communication and computation parameters of the processors and the application under consideration (δ). Because of this, it is also not possible to derive closed-form equations for the load fractions or the total processing time.

SPORT *does not guarantee* a globally optimal solution to the DLSRCHETS problem as seen in Example 6.2. However, since the solution (SPORT) is built on locally optimal values by considering two processors at a time, the error as compared to the global optimum is reduced to some degree.

Example 6.2. If $\mathcal{C} = \{10, 20, 30\}$, $\mathcal{E} = \{5, 15, 25\}$, and $\delta = 0.5$, the globally optimal solution is $\sigma_a^* = \{1, 2, 3\}$, $\sigma_c^* = \{2, 3, 1\}$, and $\alpha^* = \{0.88, 0.08, 0.04\}$, while the solution found by SPORT is $\sigma_a = \{1, 2, 3\}$, $\sigma_c = \{3, 2, 1\}$, and $\alpha = \{0.88, 0.1, 0.02\}$.

Finding the conditions for the minimization of the error is a part of the future work. The performance of SPORT is rigorously tested by performing several sets of simulations with different ranges for E_k, C_k, and δ. The details of the simulations are given in the following sections.

6.5 SIMULATION SET A

The performance of SPORT was compared to four algorithms, viz. OPT, FIFOC, LIFOC, and ITERLP. Preliminary simulations for other heuristic algorithms, viz. FIFO, LIFO, FIFOE, LIFOE, and SUMCE, revealed large errors in favor of SPORT, and it was decided not to pursue them further. These algorithms have already been explained in Section 4.7.1.

To explore the effects of system parameter values on the performance of the algorithms, several sets of simulations were carried out:

Set A1 — Homogeneous Communication Speeds, Homogeneous Computation Speeds

Set A2 — Homogeneous Communication Speeds, Heterogeneous Computation Speeds

Set A3 — Heterogeneous Communication Speeds, Homogeneous Computation Speeds

Set A4 — Heterogeneous Communication Speeds, Heterogeneous Computation Speeds

To generate data values for the above sets, the simulation method has to be changed as compared to that used to test ITERLP. Two types of intervals are used to sample data from — main and sub-intervals:

- The intervals $I_c = [C_{min}, C_{max}]$ and $I_e = [E_{min}, E_{max}]$ given in Table 6.1 are used as the main intervals. This table is the same as Table 4.3 but is reproduced here for reference and completeness.
- The main intervals are divided into m equal-sized, contiguous, non-overlapping subintervals $I_{c1}, \dots, I_{cm}$ and $I_{e1}, \dots, I_{em}$, each of size $(C_{max} - C_{min})/m$ and $(E_{max} - E_{min})/m$ respectively as shown in Fig. 6.4.
- Depending on the simulation set, the main and sub-intervals are uniformly sampled to generate the communication and computation parameters. Sampling the subintervals in this manner not only generates a homogeneous system, but also it is possible to compare a "fast" and (comparatively) "slow" homogeneous system.

The sampling method for each set is described separately in its respective section below. All plots are to log-scale to magnify the values close to zero. Not all plots are shown on account of space considerations.

Table 6.1 Parameter values used for SPORT simulation sets A and B. There are 25 cases with different values of unit communication and computation times. Depending on the sub-set of simulation set A, the intervals are further divided into sub-intervals as explained in the main text. These are uniformly sampled to generate the sets $\mathcal{C}$ and $\mathcal{E}$.

Case	$C_k \in$	$E_k \in$	Case	$C_k \in$	$E_k \in$
1	[1,10]	[1,10]	14	[10,100]	[1,100]
2	[1,10]	[10,100]	15	[10,100]	[10,1000]
3	[1,10]	[100,1000]	16	[10,1000]	[1,10]
4	[1,10]	[1,100]	17	[10,1000]	[10,100]
5	[1,10]	[10,1000]	18	[10,1000]	[100,1000]
6	[1,100]	[1,10]	19	[10,1000]	[1,100]
7	[1,100]	[10,100]	20	[10,1000]	[10,1000]
8	[1,100]	[100,1000]	21	[100,1000]	[1,10]
9	[1,100]	[1,100]	22	[100,1000]	[10,100]
10	[1,100]	[10,1000]	23	[100,1000]	[100,1000]
11	[10,100]	[1,10]	24	[100,1000]	[1,100]
12	[10,100]	[10,100]	25	[100,1000]	[10,1000]
13	[10,100]	[100,1000]			

6.5.1 *Set A1 — $\mathcal{C}$ homogeneous, $\mathcal{E}$ homogeneous*

This set was carried out using the procedure below.

1. Let $i = 1, \ldots, 25$ represent the 25 cases in Table 6.1 above.

2. For each case i, m^2 sub-cases can be defined by taking the Cartesian product of the sets of sub-intervals:

$$\{I_{c1}, \ldots, I_{cm}\} \times \{I_{e1}, \ldots, I_{em}\} = \{[I_{c1}, I_{e1}], [I_{c1}, I_{e2}], \ldots, [I_{cm}, I_{em}]\}$$

   Let $j = 1, \ldots, m^2$ represent the sub-cases in each case i.

3. Let $v = 1, 2, 3, 4$ represent the variants SPORT, ITERLP, LIFOC, and FIFOC.

4. For each case i, sub-case j, $k = 1, \ldots, 100$ simulation runs are carried out. In each run, the corresponding sub-intervals are uniformly sampled to generate the values of $\mathcal{C}$ and $\mathcal{E}$.

5. In each run, the optimal processing time T_o^{ijk} and the variant processing time T_v^{ijk} are found.

6. Percentage error from optimal is calculated as:

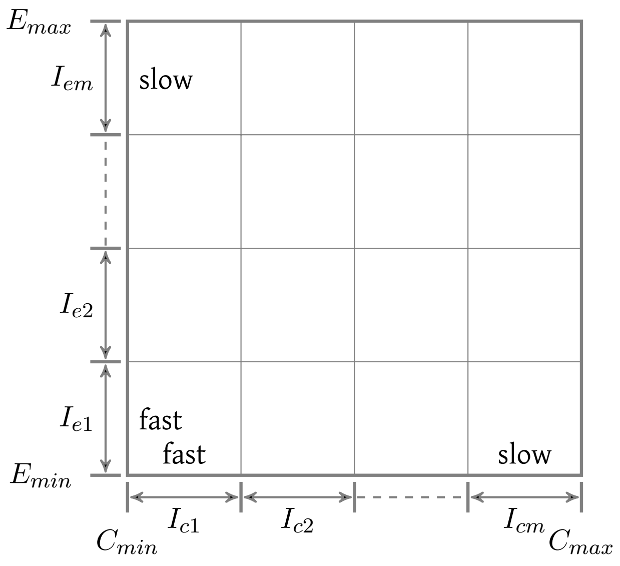


Figure 6.4 The main and sub-intervals used for Simulation Set A. The main intervals are $I_c = [C_{min}, C_{max}]$ and $I_e = [E_{min}, E_{max}]$ given in Table 6.1. The division into and sampling of sub-intervals enables the creation of fairly homogeneous sets $\mathcal{C}$ and $\mathcal{E}$. Sampling the sub-intervals close to the origin gives fast processors and network links, that get slower away from the origin. Sampling the main intervals generates heterogeneous sets $\mathcal{C}$ and $\mathcal{E}$.

$$\Delta T_v^{ijk} = \frac{T_v^{ijk} - T_o^{ijk}}{T_o^{ijk}} * 100\% \qquad i = 1, \ldots, 25, j = 1, \ldots, m^2, \; k = 1, \ldots, 100, v = 1, \ldots, 4 \tag{6.2}$$

7. Mean percent error from optimal is:

$$\overline{\Delta T}_v^{ij} = \frac{1}{100} \sum_{k=1}^{100} \Delta T_v^{ijk} \qquad i = 1, \ldots, 25, j = 1, \ldots, m^2, v = 1, \ldots, 4$$

8. The cases in Table 6.1 are defined with intervals differing in both interval width (ratio) as well as the absolute values of the communication and computation parameters. To aggregate the performance obtained over all the intervals, the error values of the individual sub-cases should be averaged over the 25 cases in Table 6.1 to give the final mean percent error for each sub-case. For each variant, the mean percent error is averaged over the 25 cases to give:

$$\langle \overline{\Delta T} \rangle_v^j = \frac{1}{25} \sum_{i=1}^{25} \overline{\Delta T}_v^{ij} \qquad j = 1, \ldots, m^2, v = 1, \ldots, 4$$

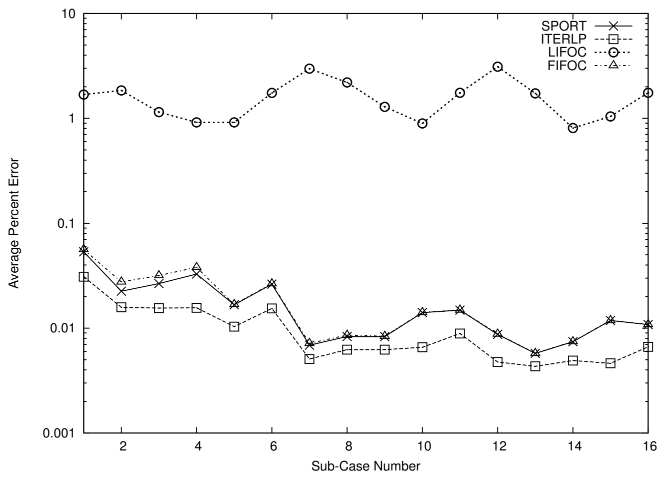


Figure 6.5 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A1 for $m = 4$, $\delta = 0.2$. SPORT performs almost exactly the same as FIFOC, with error between 0.1% and $0.01\% \approx 0$. ITERLP performance is even better, while LIFOC has comparatively large error. This figure appears in [47].

9. The values of $\langle\overline{\Delta T}\rangle_v^j$ are used for plotting and comparison of performance of the different variant algorithms.

10. This set is performed for $m = 4, 5$ and $\delta = 0.2, 0.5, 0.8$

11. Sub-case number 1 corresponds to fast communication and computation speeds while sub-case numbers 16 (for $m = 4$) and 25 (for $m = 25$) correspond to slow communication and computation speeds.

12. Intermediate cases represent combinations of slow and fast communication and computation speeds.

These values of $\langle\overline{\Delta T}\rangle_v^j$ are plotted in Figs. 6.5 and 6.6 for (m, δ) pairs $(4, 0.2)$ and (5,0.8) respectively.

Because the network links are homogeneous, FIFOC is expected to perform well. It is observed that SPORT performs almost exactly the same as FIFOC, with error between 0.1% and $0.01\% \approx 0$. ITERLP performance is even better, while LIFOC has comparatively large error, and the error increases with increase in value of δ.

6.5.2 *Set A2 — $\mathcal{C}$ homogeneous, $\mathcal{E}$ heterogeneous*

Similar to Set A1, m sub-intervals $I_{c1}, \ldots, I_{cm}$ are used for the communication parameters, but the computation parameters are generated by sampling the interval $I_e = [E_{min}, E_{max}]$. This creates m sub-cases with intervals $\{(I_{c1}, I_e), \ldots, (I_{cm}, I_e)\}$, such that

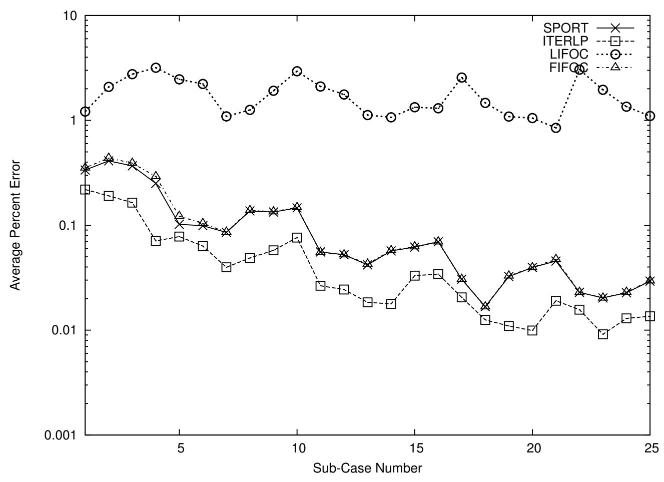


Figure 6.6 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A1 for $m = 5$, $\delta = 0.8$. SPORT performs almost exactly the same as FIFOC, with error between 0.1% and $0.01\% \approx 0$. ITERLP performance is even better, while LIFOC has comparatively large error, and the error increases with increase in value of δ. This figure appears in [47].

in each sub-case, the communication parameters are homogeneous, but the computation parameters are heterogeneous. The procedure used in this set is enumerated below

1. This set is performed for $m = 4, 5$ and $\delta = 0.2, 0.5, 0.8$.

2. There are $j = 1, \ldots, m$ sub-cases with intervals $\{(I_{c1}, I_e), \ldots, (I_{cm}, I_e)\}$, for each case i, where $I_e = [E_{min}, E_{max}]$.

3. The values of ΔT_v^{ijk}, $\overline{\Delta T}_v^{ij}$, and $\langle\overline{\Delta T}\rangle_v^j$ are calculated as in Set A1, with the only difference that the value of j ranges over $1, \ldots, m$ instead of $1, \ldots, m^2$ as in Set A1.

4. The values of $\langle\overline{\Delta T}\rangle_v^j$ are plotted for the different algorithms.

5. Sub-case 1 represents fast communication speed while sub-cases 4 (for $m = 4$) and 5 (for $m = 5$) represent slow communication speeds.

6. Computation speeds of the processors are heterogeneous.

Figs. 6.7 and 6.8 show the plots for (m, δ) pairs $(4, 0.5)$ and (5,0.2) respectively.

Again it is observed that as long as the network links are homogeneous, SPORT, ITERLP, and FIFOC are insensitive to the heterogeneity in the computation speed of the processors with average error between 0.1% to 0.001% ≈ 0. On the other hand, errors in LIFOC persist and increase with δ.

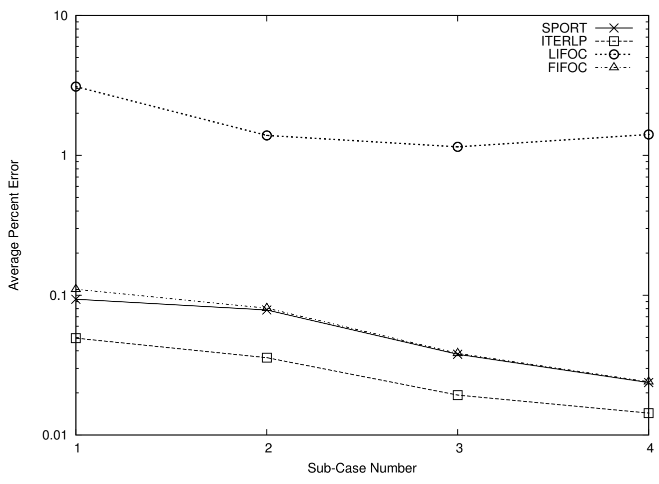


Figure 6.7 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A2 for $m = 4$, $\delta = 0.5$. SPORT, ITERLP, and FIFOC error is between 0.1% to 0.001% ≈ 0. On the other hand, errors in LIFOC persist. This figure appears in [47].

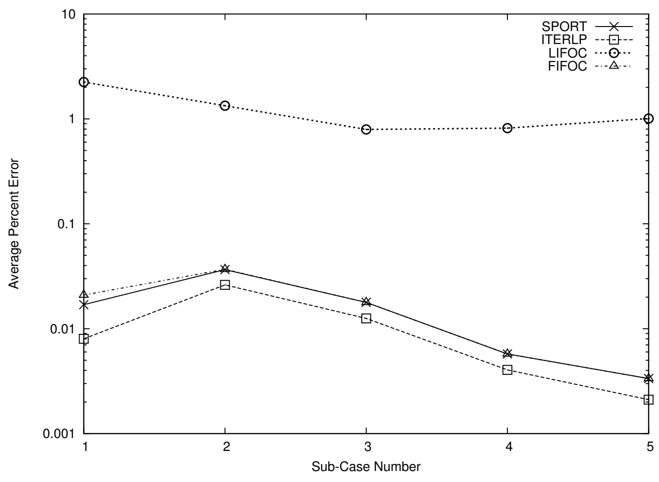


Figure 6.8 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A2 for $m = 5$, $\delta = 0.2$. SPORT, ITERLP, and FIFOC error is between 0.1% to 0.001% ≈ 0. On the other hand, errors in LIFOC persist and increase with δ. This figure appears in [47].

6.5.3 Set A3 — $\mathcal{C}$ heterogeneous, $\mathcal{E}$ homogeneous

This is the complement of Set A2, and the intervals $\{(I_c, I_{e1}), \ldots, (I_c, I_{em})\}$ form the m sub-cases, where $I_c = [C_{min}, C_{max}]$, such that the computation parameters are homogeneous and the communication parameters are heterogeneous.

1. This set is performed for $m = 4, 5$ and $\delta = 0.2, 0.5, 0.8$.

2. There are $j = 1, \ldots, m$ sub-cases $\{(I_c, I_{e1}), \ldots, (I_c, I_{em})\}$ for each case i, where

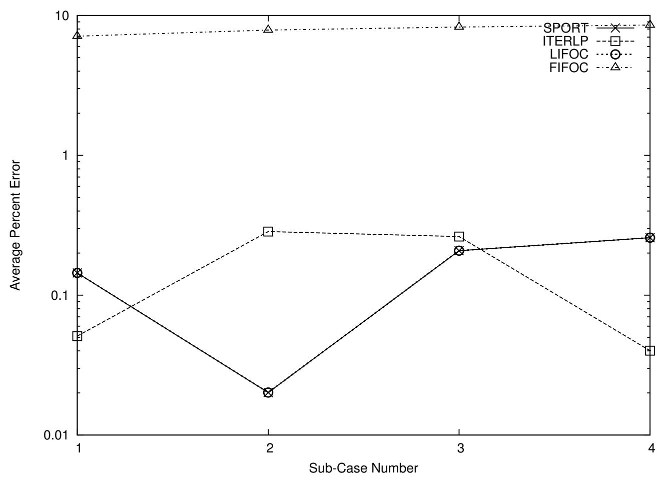


Figure 6.9 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A3 for $m = 4$, $\delta = 0.8$. FIFOC has large error while SPORT continues to have low error values around 0.1% along with LIFOC and ITERLP. This figure appears in [47].

$I_c = [C_{min}, C_{max}]$.

3. The values of ΔT_v^{ijk}, $\overline{\Delta T}_v^{ij}$, and $\langle\overline{\Delta T}\rangle_v^j$ are calculated as in Set A2.

4. The values of $\langle\overline{\Delta T}\rangle_v^j$ are plotted.

5. Sub-case 1 represents fast computation speed while sub-cases 4 (for $m = 4$) and 5 (for $m = 5$) represent slow computation speeds.

6. Communication speeds of the network links are heterogeneous.

The plots for (m, δ) pairs $(4, 0.8)$ and $(5, 0.5)$ are shown in Figs. 6.9 and 6.10 respectively.

This simulation set clearly shows the adaptiveness of SPORT. FIFOC, which had almost zero error in the previous two sets, now has large error as compared to the optimal schedule. SPORT however, continues to have low error values around 0.1% along with LIFOC and ITERLP.

6.5.4 *Set A4 — $\mathcal{C}$ heterogeneous, $\mathcal{E}$ heterogeneous*

Similar to the previous sets, the sub-intervals $I_{c1}, \ldots, I_{cm}$ and $I_{e1}, \ldots, I_{em}$ for each case are found. Each sub-interval is sampled once to generate a total of m values each for the communication and computation parameters. The values undergo a random permutation first before being assigned to the processors. Sampling the sub-intervals in this

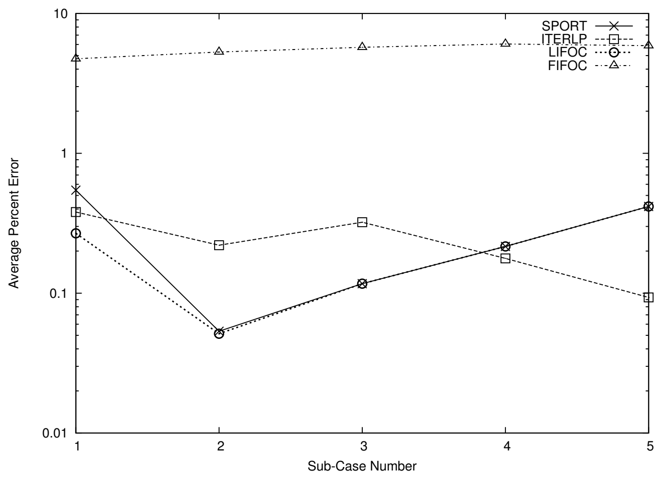


Figure 6.10 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A3 for $m = 5$, $\delta = 0.5$. FIFOC has large error while SPORT continues to have low error values around 0.1% along with LIFOC and ITERLP. This figure appears in [47].

manner minimizes the possibility of two processors being allocated similar communication or computation parameters and generates a truly heterogeneous system. There are no sub-cases here.

1. This set is performed for $m = 4, 5$ and $\delta = 0.2, 0.5, 0.8$.
2. There are no sub-cases. I_c and I_e are sampled directly.
3. The mean error from optimal is found as:

$$\overline{\Delta T}^i_v = \frac{1}{100}\sum_{k=1}^{100} \Delta T^{ik}_v \qquad i = 1, \ldots, 25, v = 1, \ldots, 4$$

4. The final error values for each variant $\langle\overline{\Delta T}\rangle_v$ are calculated by averaging over the 25 cases in Table 6.1 as

$$\langle\overline{\Delta T}\rangle_v = \frac{1}{25}\sum_{i=1}^{25} \overline{\Delta T}^i_v \qquad v = 1, \ldots, 4$$

5. The values of $\langle\overline{\Delta T}\rangle_v$ at different values of δ are plotted.
6. There is no differentiation between slow and fast speeds as the system is completely heterogeneous.

The values of $\langle\overline{\Delta T_v}\rangle$ for $\delta = 0.2, 0.5, 0.8$ at $m = 4$ and 5 are shown in Figs. 6.11 and 6.12 respectively.

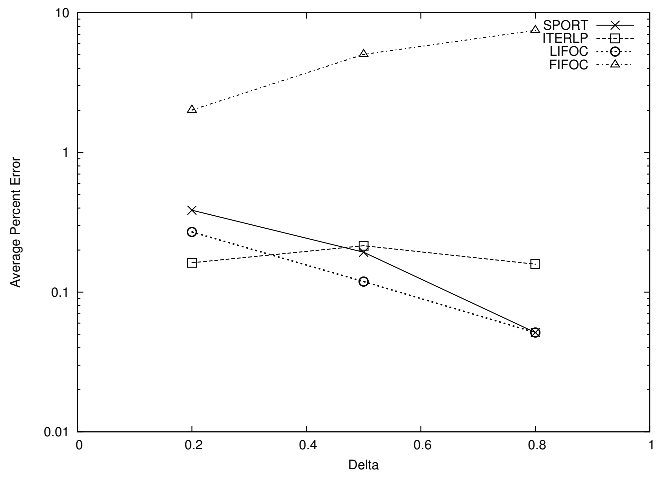


Figure 6.11 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A4 for $m = 4$. The percent error of FIFOC increases with the increase in δ, but there is a reduction in the error of the other three variants. This figure appears in [47].

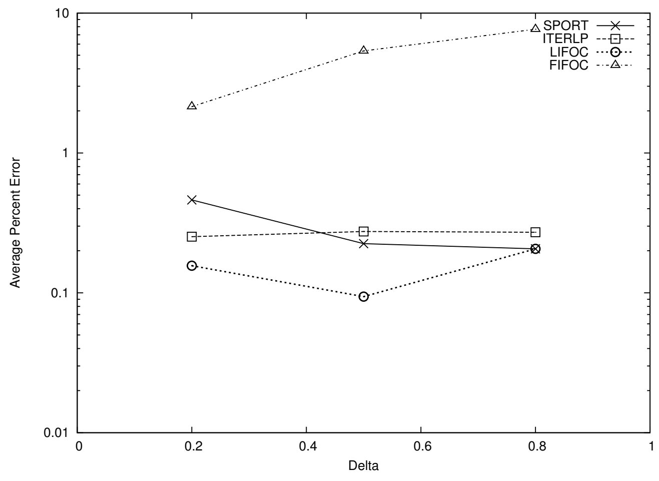


Figure 6.12 Average percent error with respect to optimal, $\langle\overline{\Delta T}\rangle$, in Set A4 for $m = 5$. The percent error of FIFOC increases with the increase in δ, but there is a reduction in the error of the other three variants. This figure appears in [47].

SPORT, LIFOC, and ITERLP are seen adapt to the heterogeneity in the processor computation and network link speeds. The percent error of FIFOC increases with the increase in δ, but there is a reduction in the error of the other three variants.

6.5.5 Simulation Result Analysis

Though not strictly applicable, some trends can be identified:

- In set A1, for the same value of δ, errors for "fast" homogeneous systems are higher

Table 6.2 Minimum statistics for SPORT simulation set A. In sets A1 and A2, the minimum errors in LIFOC are 2 orders of magnitude higher than SPORT, ITERLP, and FIFOC. In sets A3 and A4, FIFOC error is 2 to 3 orders of magnitude higher than the other three algorithms.

Set	m	$\delta = 0.2$				$\delta = 0.5$			
		SPORT	ITERLP	LIFOC	FIFOC	SPORT	ITERLP	LIFOC	FIFOC
A1	4	5.73e-03	4.32e-03	8.08e-01	5.76e-03	2.20e-02	1.06e-02	1.07e+00	2.21e-02
	5	7.89e-04	6.90e-04	7.21e-01	7.89e-04	5.40e-03	4.21e-03	9.63e-01	5.30e-03
A2	4	1.01e-02	5.78e-03	8.41e-01	1.01e-02	2.37e-02	1.43e-02	1.15e+00	2.40e-02
	5	3.34e-03	2.10e-03	7.93e-01	3.34e-03	1.06e-02	8.92e-03	1.10e+00	1.07e-02
A3	4	2.03e-01	1.80e-03	1.05e-01	1.61e+00	1.12e-01	5.13e-03	9.59e-02	4.43e+00
	5	3.96e-01	1.90e-01	8.90e-02	1.75e+00	5.34e-02	9.32e-02	5.13e-02	4.74e+00
A4	4	4.95e-06	1.97e-16	4.92e-06	1.05e+00	3.09e-02	2.77e-15	3.09e-02	3.23e+00
	5	1.08e-02	5.81e-04	2.75e-06	1.15e+00	5.84e-02	2.18e-03	5.84e-02	3.74e+00

Set	m	$\delta = 0.8$			
		SPORT	ITERLP	LIFOC	FIFOC
A1	4	3.58e-02	1.78e-02	1.16e+00	3.66e-02
	5	1.67e-02	9.13e-03	8.47e-01	1.67e-02
A2	4	3.59e-02	2.06e-02	1.22e+00	3.71e-02
	5	2.01e-02	1.69e-02	1.06e+00	2.02e-02
A3	4	2.01e-02	4.01e-02	2.01e-02	7.11e+00
	5	5.14e-02	4.98e-03	5.14e-02	7.42e+00
A4	4	6.15e-02	4.01e-03	6.15e-02	5.58e+00
	5	9.43e-11	0.00e+00	7.05e-11	6.38e-01

than "slow" homogeneous systems.

- In sets A2 and A3, for the same value of δ, errors for the "fast" and heterogeneous systems are higher than the "slow" and heterogeneous systems.
- In sets A1, A2, and A3, where either one or both of the variables are homogeneous, the average error increases with increase in δ. However, in set A4, error reduces with δ, for the better performing algorithms.

The minimum and maximum mean error values of each algorithm are tabulated in Tables 6.2 and 6.3. Scientific notation is used to enable a quick comparison of the algorithms in terms of *orders of magnitude*. It can be observed that overall in sets A1 and A2, the minimum and maximum errors in LIFOC are 2 orders of magnitude higher than SPORT, ITERLP, and FIFOC. On the other hand in sets A3 and A4, FIFOC error is 2 to 3 orders of magnitude higher than the other three algorithms.

The extensive simulations carried out in Set A clearly show that:

Table 6.3 Maximum statistics for SPORT simulation set A. In sets A1 and A2, the maximum errors in LIFOC are 2 orders of magnitude higher than SPORT, ITERLP, and FIFOC. In sets A3 and A4, FIFOC error is 2 to 3 orders of magnitude higher than the other three algorithms.

Set	m	$\delta = 0.2$				$\delta = 0.5$			
		SPORT	ITERLP	LIFOC	FIFOC	SPORT	ITERLP	LIFOC	FIFOC
A1	4	5.34e-02	3.09e-02	3.11e+00	5.61e-02	1.84e-01	7.57e-02	4.20e+00	2.02e-01
	5	8.24e-02	4.87e-02	3.00e+00	8.79e-02	2.26e-01	1.19e-01	3.91e+00	2.30e-01
A2	4	3.03e-02	1.69e-02	1.83e+00	3.06e-02	9.35e-02	4.93e-02	3.10e+00	1.10e-01
	5	3.66e-02	2.61e-02	2.24e+00	3.68e-02	1.15e-01	8.34e-02	2.75e+00	1.26e-01
A3	4	4.01e-01	3.42e-01	4.66e-01	2.02e+00	4.03e-01	2.22e-01	4.03e-01	5.44e+00
	5	5.31e-01	3.86e-01	4.84e-01	2.30e+00	5.45e-01	3.80e-01	4.16e-01	6.05e+00
A4	4	1.32e+00	6.50e-01	8.84e-01	4.47e+00	8.02e-01	7.11e-01	4.00e-01	1.12e+01
	5	1.56e+00	7.66e-01	4.34e-01	4.85e+00	9.35e-01	8.97e-01	4.24e-01	1.15e+01

Set	m	$\delta = 0.8$			
		SPORT	ITERLP	LIFOC	FIFOC
A1	4	2.57e-01	1.13e-01	3.39e+00	3.08e-01
	5	4.10e-01	2.19e-01	3.17e+00	4.37e-01
A2	4	2.44e-01	1.10e-01	2.91e+00	2.79e-01
	5	2.72e-01	1.27e-01	2.84e+00	2.89e-01
A3	4	2.57e-01	2.85e-01	2.57e-01	8.53e+00
	5	2.55e-01	4.37e-01	2.55e-01	9.22e+00
A4	4	1.56e-01	6.26e-01	1.56e-01	1.64e+01
	5	1.36e+00	2.04e+00	1.36e+00	1.63e+01

- If network links are homogeneous, then LIFOC performance is affected for both homogeneous and heterogeneous computation speeds.

- If network links are heterogeneous, then FIFOC performance is affected for both homogeneous and heterogeneous computation speeds.

- SPORT performance is also affected to a certain degree by the heterogeneity in network links and computation speeds, but since SPORT does not use a single predefined sequence of allocation and collection, it is able to better adapt to the changing system conditions.

- ITERLP performance is somewhat better than SPORT, but is expensive to compute. SPORT generates similar schedules at a fraction of the cost.

6.6 SIMULATION SET B

To evaluate the performance of the algorithms with the increase in number of nodes, the processing time of SPORT was compared with only FIFOC and LIFOC. This is because, OPT and ITERLP cannot be practically carried out beyond $m = 5$ and $m = 10$ respectively. In this simulation set, the algorithms are tested to compare their performance for large number of processors because P2P systems, and volunteer and grid computing platforms can potentially have hundreds of nodes.

6.6.1 *Simulation Method*

Using the procedure used in simulation Set A4, 100 simulation runs were carried out for SPORT, LIFOC, and FIFOC, at $m = 10, 50, 100, \ldots, 300, 350$, and $\delta = 0.2, 0.5, 0.8$ for each of the 25 cases listed in Table 6.1. ΔT_v, for each variant v (LIFOC $:= 1$ and FIFOC $:= 2$) was found as:

$$\Delta T_v = \frac{T_v - T_{\text{SPORT}}}{T_{\text{SPORT}}} * 100\% \qquad v = 1, 2$$

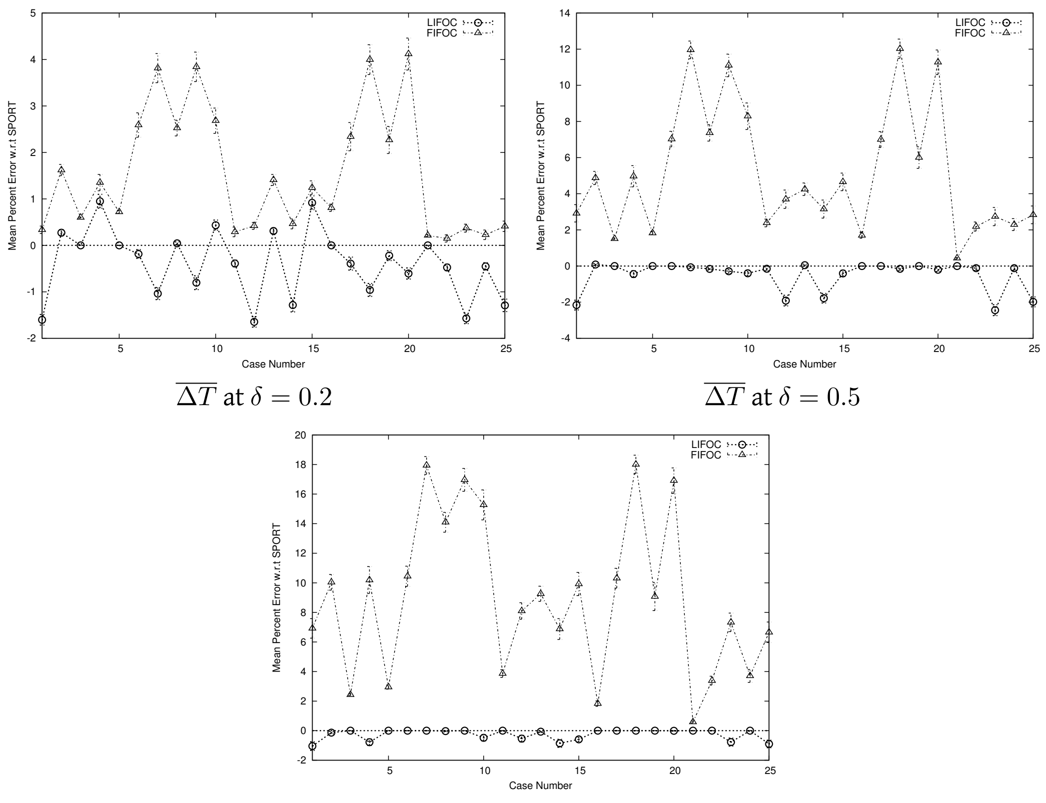
Mean error, $\overline{\Delta T_v^i}$, for each case $i = 1, \ldots, 25$ in Table 6.1 was calculated by averaging ΔT_v^{ik}, $k = 1, \ldots, 100$, over the 100 simulation runs and plotted.

6.6.2 *Simulation Results and Analysis*

Figure 6.13 shows the plots for $\overline{\Delta T_v^i}$ at $m = 10$, $\delta = 0.2, 0.5, 0.8$. First of all, FIFOC is seen to always have a positive error with respect to SPORT. This is to be expected since the system is heterogeneous. The value of error increases with increase in the value of δ.

Secondly, LIFOC has a negative error with respect to SPORT for several cases at $\delta = 0.2$, i.e. the processing time of LIFOC is smaller than SPORT. This is also to be expected since LIFOC uses all available processors and every processor added reduces the processing time by some amount. This ends up distributing very tiny load fractions (smaller than 1×10^{-6}) to a large number of tail-end processors. As the value of δ increases, the error between LIFOC and SPORT becomes insignificant.

This pattern of results is repeated even for higher values of m as can be seen in Figs. 6.14 and 6.15 for $m = 100$ and 300 respectively. It can be observed that as the number of processors increases, FIFOC performance in case numbers 11–15 and 21–25 becomes almost equal to that of SPORT. These ranges correspond to the intervals $I_c = [10, 100]$ and $I_c = [100, 1000]$ respectively, i.e., a ratio of $C_{min} : C_{max} = 1 : 10$. Because of the methodology used to perform the simulations, with a large number of processors, the values of C_k tend to become similar to each other. Consequently, $(C_2 - C_1)$ in (5.22) becomes small,



$\overline{\Delta T}$ at $\delta = 0.2$

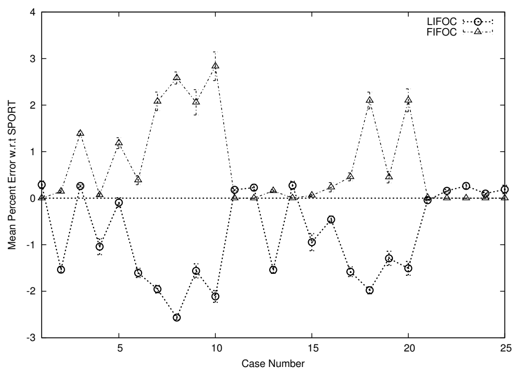
$\overline{\Delta T}$ at $\delta = 0.5$

$\overline{\Delta T}$ at $\delta = 0.8$

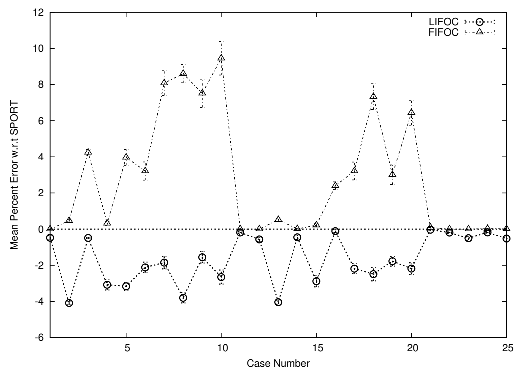
Figure 6.13 Mean percent error with respect to SPORT, $\overline{\Delta T}$, in simulation set B, at $m = 10$. FIFOC always have a positive error and value of error increases with increase in the value of δ. LIFOC has a negative error with respect to SPORT for several cases at $\delta = 0.2$. As the value of δ increases, the error between LIFOC and SPORT becomes insignificant. This figure appears in [47].

and Schedule f always tends to be optimal for each pair of processors being compared. If Schedule f is optimal for all processors in SPORT, the resulting σ_a and σ_c are the same as FIFOC. However, surprisingly, cases 1–5, with $I_c = [1, 10]$ do not show this trend. This leads us to hypothesize, that the performance of the algorithms not only depends on the range (ratio) of parameters but also on the absolute values of the parameters. This belief is reinforced by the fact that in case numbers 21–25, LIFOC also has comparable performance to SPORT, especially at higher values of δ and m.

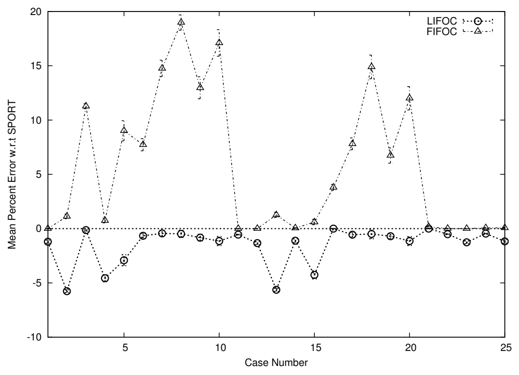
Consider Table 6.4 that gives the minimum error of LIFOC with respect to SPORT, the case number when it occurs, along with the mean error of LIFOC averaged over all 25 cases (i.e., $\langle\overline{\Delta T_v}\rangle$ in set A4) for different values of δ and m. The minimum error for LIFOC is -5.76% for $m = 100$, $\delta = 0.8$, case number 2, but the minimum average error is -2.12% for $m = 300$, $\delta = 0.5$. It can be observed that the average error values at $\delta = 0.5$ are all



$\overline{\Delta T}$ at $\delta = 0.2$



$\overline{\Delta T}$ at $\delta = 0.5$



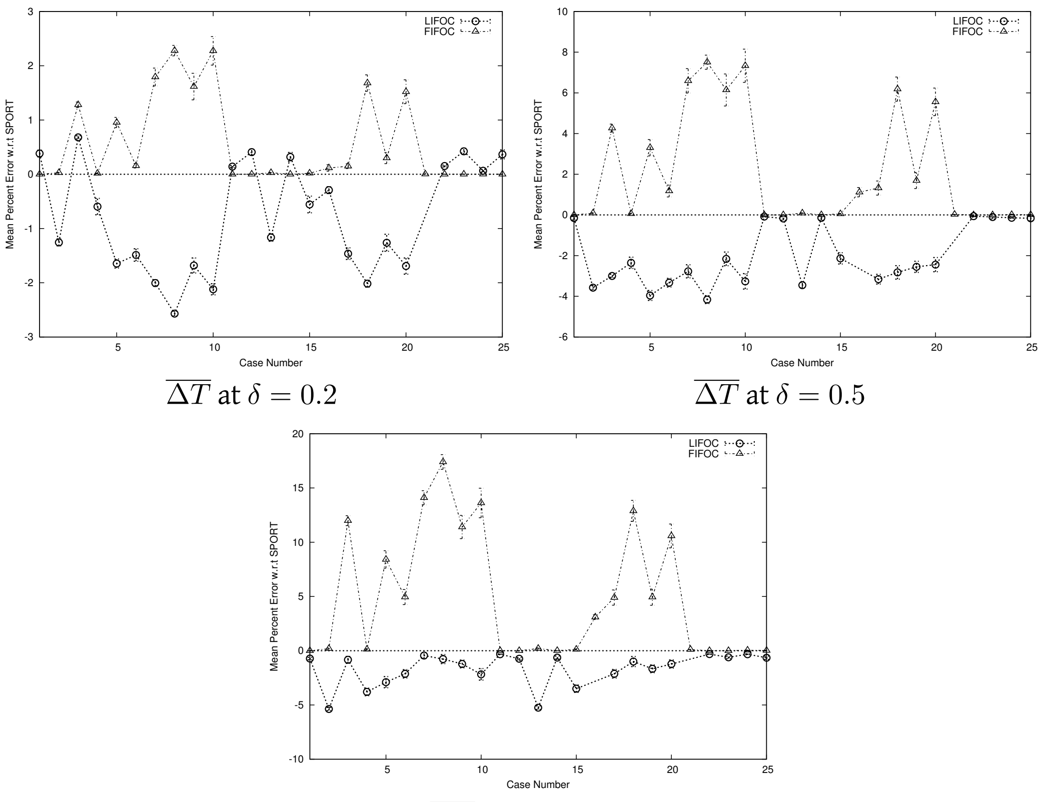
$\overline{\Delta T}$ at $\delta = 0.8$

Figure 6.14 Mean percent error with respect to SPORT, $\overline{\Delta T}$, in simulation set B, at $m = 100$. FIFOC always have a positive error, but the error is reduced in some cases. The value of error increases with increase δ. LIFOC has a larger negative error with respect to SPORT for several cases at $\delta = 0.2$. As the value of δ increases, the error between LIFOC and SPORT becomes insignificant. This figure appears in [47].

smaller than those at $\delta = 0.2$, while the average error values at $\delta = 0.8$ are again greater than those at $\delta = 0.5$ (except for $m = 250$). We hypothesize that initially as δ increases, the error increases, but as $\delta \to 1$, i.e., size of result data approaches the size of allocated load, performance of SPORT and LIFOC becomes similar. This is supported by the results of set A4, where the $\langle \overline{\Delta T_v} \rangle$ of SPORT and LIFOC is almost equal for $\delta = 0.8$ (see Figs. 6.11 and 6.12).

6.6.3 Discussion on Performance of LIFOC and SPORT

There is a significant downside to LIFOC because of its property to use all available processors — the time required to compute the optimal solution (wall-clock time) is almost two orders of magnitude greater than that of SPORT as seen in Table 6.5 and Fig. 6.16. These values were obtained separately from the simulations above by averaging the wall-



$\overline{\Delta T}$ at $\delta = 0.2$ $\overline{\Delta T}$ at $\delta = 0.5$

$\overline{\Delta T}$ at $\delta = 0.8$

Figure 6.15 Mean percent error with respect to SPORT, $\overline{\Delta T}$, in simulation set B, at $m = 300$. FIFOC error is reduced in several cases, but the value of error increases with increase δ. LIFOC has a larger negative error with respect to SPORT for most cases at $\delta = 0.2$. As the value of δ increases, the error between LIFOC and SPORT becomes insignificant. This figure appears in [47].

clock time over 100 runs for $I_c = [10, 100]$, $I_e = [50, 500]$, and $\delta = 0.5$. The results show that though both SPORT and LIFOC are $O(m)$ algorithms given a set of processors sorted by decreasing communication bandwidth, clearly SPORT is the better performing algorithm, with the best cost-performance ratio for large values of m. The values for FIFOC are almost four orders of magnitude larger than SPORT — too large to even warrant consideration.

The other disadvantage of LIFOC is that the chain of multiplications involved in the calculation of load fractions quickly leads to underflow because the numbers involved are tiny fractions and multiplying them results in smaller and smaller numbers until the floating point system cannot handle them anymore. Because of this, for $m > 150$, it is difficult to get valid results for LIFOC in a large number of cases. For example, for $m = 250, 300, 350$, LIFOC returned underflow errors in 24, 32 and 32 runs respectively out of

Table 6.4 Statistics for LIFOC in simulation set B. The minimum error for LIFOC is -5.76% for $m = 100$, $\delta = 0.8$, case number 2, but the minimum average error is -2.12% for $m = 300$, $\delta = 0.5$.

m	$\delta = 0.2$			$\delta = 0.5$			$\delta = 0.8$		
	case	min	avg	case	min	avg	case	min	avg
10	12	-1.64	-0.39	23	-2.44	-0.50	1	-1.04	-0.24
50	8	-2.41	-0.88	2	-4.39	-1.47	4	-3.72	-1.33
100	8	-2.56	-0.79	2	-4.08	-1.66	2	-5.76	-1.49
150	8	-2.56	-0.78	8	-4.16	-2.01	2	-5.37	-1.68
200	8	-2.57	-0.82	5	-4.25	-2.06	13	-4.55	-1.77
250	8	-2.55	-0.77	17	-4.28	-1.36	17	-4.01	-1.85
300	8	-2.54	-0.88	3	-4.57	-2.12	5	-4.47	-1.70
350	8	-2.52	-0.83	3	-4.63	-2.04	5	-4.53	-1.67

Table 6.5 Comparison of wall-clock time for SPORT, LIFOC, and FIFOC. SPORT is two orders of magnitude faster than LIFOC and almost four orders of magnitude faster than FIFOC.

m	SPORT (s)	LIFOC (s)	FIFOC (s)
50	0.00025	0.00427	0.1190
100	0.00056	0.00687	0.4690
150	0.00063	0.01038	1.3190
200	0.00071	0.01409	2.8740
250	0.00077	0.01743	5.3990
300	0.00084	0.02112	9.1100
350	0.00092	0.02509	14.046
400	0.00099	0.02951	20.419
450	0.00107	0.03458	28.482
500	0.00114	0.04018	37.497

the 100 simulation runs carried out. In MATLAB™, this causes a NaN (Not a Number) to be returned, and the load fractions cannot be calculated. Of course this is not a limitation of the algorithm itself, nevertheless it is an important practical consideration during implementation.

6.7 SUMMARY

This chapter introduced the SPORT algorithm as a good solution to the DLSRCHETS problem. The basic idea behind SPORT is very simple — to use two processors at a time and build a piecewise locally optimal schedule. However it is not very straightforward to

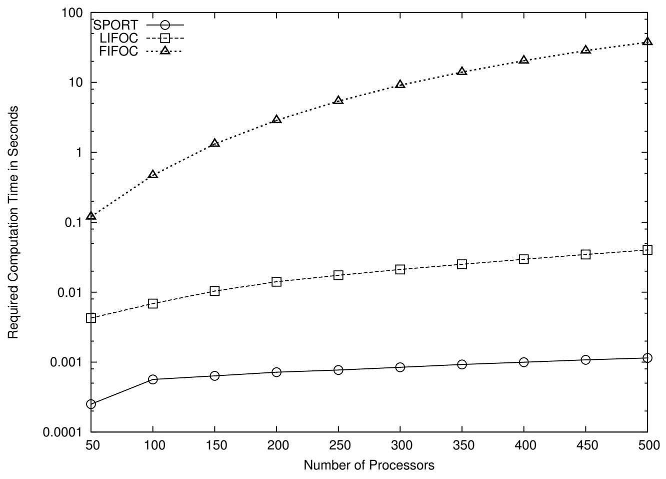


Figure 6.16 Comparison of wall-clock time for SPORT, LIFOC, and FIFOC. SPORT is two orders of magnitude faster than LIFOC and almost four orders of magnitude faster than FIFOC. This figure appears in [47].

be able to do this, and several tools are necessary that were designed over the preceding chapters in this thesis.

The comprehensive simulation testing of the performance of the algorithms is undoubtedly the highlight of this chapter. SPORT performance was proved to be robust to heterogeneity, number of participants, and value of δ. Moreover, this superior performance is obtained at a fraction of the computation time of other algorithms.

CHAPTER 7

CONCLUSION

In this thesis, the DLSRCHETS problem for the scheduling of divisible loads on heterogeneous master-slave systems and considering the result collection phase was formulated and analyzed. Two new polynomial-time algorithms were proposed and tested. Several new intermediate results were obtained during the work on these algorithms. This final chapter summarizes the various points covered in the thesis and presents several ideas for future work.

7.1 SUMMARY OF THE THESIS

This work considers the most general form of DLSRCHETS. No assumptions are made regarding the number of slaves that are allocated load, both the network and computation speeds of the slaves are considered to be heterogeneous, and idle time can be present in the schedule if it reduces the makespan.

The theoretical basis of DLSRCHETS was first established, and it was defined in terms of a linear program for analysis. The optimal schedule for a system with two slaves was extensively explored because the proposed algorithms are built on it. Two new polynomial-time algorithms, namely ITERLP (ITERative Linear Programming) and SPORT (System Parameters based Optimized Result Transfer) were proposed as solutions to DLSRCHETS. The performance of traditional and new algorithms was compared using a large number of simulations and the proposed algorithms were shown to have superior performance.

The features and performance of the algorithms are summarized in Table 7.1. The brute force approach finds the optimal solution, but is impractical as it is computationally much to expensive. ITERLP generates near-optimal schedules, and allows comparison of other heuristic algorithms when it is impossible to find the optimal solution. SPORT is extremely fast, with an error that is slightly more than ITERLP. Practically, SPORT offers the best cost-performance ratio.

Table 7.1 A summary of algorithm features and performance. Brute force finds the optimal solution, but is impractical. ITERLP generates near-optimal schedules, but is still computationally expensive. SPORT is extremely fast, with an error that is slightly more than ITERLP.

Algorithm	Features	Complexity	Performance
Brute force	Impractical, optimal	$O((m!)^2)$ LP	6 proc. $\Rightarrow$ 80 min 7 proc. $\Rightarrow$ 70 hrs
ITERLP	Expensive, near-optimal, adapts to heterogeneity	$O(m^3)$ LP	65 proc. $\Rightarrow$ 80 min 100 proc. $\Rightarrow$ 15 hrs **max error = 0.8%**
SPORT	Fast, near-optimal, adapts to heterogeneity	$O(m \log m)$	500 proc. $\Rightarrow$ 1 ms **max error = 1.5%**

Several new and unique contributions resulted from the work on this thesis:

The *Allocation Precedence Lemma* The allocation precedence condition states that, *the master distributes load to all participating slaves first, before receiving any results.* The allocation precedence lemma proves that in the general case considered in this thesis, there always exists an optimal schedule that satisfies the allocation precedence condition. This is necessary to limit the range of optimal solutions to DLSRCHETS to a finite number.

The *Idle Time Theorem* A proof is given for the Idle Time Theorem, which states that, *there exists an optimal solution for DLSRCHETS in which, irrespective of whether load is allocated to all available slaves, at most one of the slaves allocated load has idle time, and that the idle time exists only when the result collection begins immediately after the completion of load distribution.* This is one of the principal contributions of this thesis. First, because it shows that in some cases insertion of idle time can be beneficial, and second, because it enables the definition of a constraint on the number of processors to be used in the SPORT algorithm.

The ITERLP Algorithm The new ITERLP (ITERative Linear Programming) algorithm is proposed and found to be near-optimal after rigorous testing. The ITERLP algorithm does not necessarily use all processors (slaves) and determines the number of processors to be used by repeatedly solving a number of linear programs. The complexity of ITERLP is polynomial in the number of slaves (m) and requires solving $O(m^3)$ linear programs in the worst case. Though the algorithm is computationally too expensive to be used for a large number of slaves, nevertheless it can be used as a benchmark to compare other heuristic algorithms when obtaining the optimal solution is impractical.

Condition for Idle Time The idle time theorem proves that under some conditions, idle time may be present in a single processor, but does not specify when the idle time will be present, i.e., under what conditions of the processor communication and computation speeds does it occur. For the first time in DLT, the condition to identify the presence of idle time in a FIFO schedule for two slaves is derived. It has already been proved that there can never be idle time in a LIFO schedule. What is surprising is the simplicity of the condition, and how it is related not only to the communication and computation speeds, but also to the particular divisible load under consideration, specifically to the ratio of size of the result data to the size of input data.

Condition for Optimality The identification of the limiting condition for the optimality of the FIFO and LIFO schedules for two processors is a significant addition to DLT. This condition shows that even though the presence of idle time depends on the divisible load under consideration, whether LIFO or FIFO is optimal in a two-slave system depends *only on* the communication speeds of the two processors, and the computation speeds do not matter. This condition supports the conclusions drawn by Rosenberg [77] regarding the performance of LIFO and FIFO.

The concept of *equivalent processor* The equivalent processor concept was used by Bharadwaj et al. [24] to prove a number of results in traditional DLT. It is introduced here for the first time in divisible load scheduling for heterogeneous systems with result collection. The equivalent processor is used to summarize the total processing capacity of a pair of slaves. It enables derivation of a piecewise locally optimal solution to DLSRCHETS by combining two processors into one (virtual) processor at a time.

The SPORT Algorithm The polynomial-time heuristic algorithm SPORT (System Parameters based Optimized Result Transfer) is another principal contribution of this thesis. The algorithm gives near-optimal solutions to DLSRCHETS and is robust to system heterogeneity. The SPORT algorithm does not necessarily use all processors and determines the number of processors to be used based on the system parameters (computation and communication capacities). SPORT simultaneously finds the sequence of load allocation and result collection, and the load fractions to be allocated to the processors. Given m processors sorted in the order of decreasing network link bandwidth, the complexity of SPORT is of the order $O(m)$, which is a huge improvement over ITERLP. It is rigorously tested using simulations and its performance is found to be only slightly worse than that of ITERLP.

This work has added to the body of knowledge on divisible load scheduling, and the concepts and results herein have been published in two international journal papers, one major international refereed conference paper, and several domestic conference papers.

7.2 IDEAS FOR FUTURE WORK

Future work can proceed in the following main directions:

1. Theoretical analysis of complexity and other optimality results
2. Extensions to the current system model
3. Modifying the nature of DLSRCHETS itself
4. Development of applications and physical testing

Each of these is considered separately below.

7.2.1 *Theoretical Analysis*

The complexity of DLSRCHETS is still an open issue. It makes for an interesting research topic. Is it at all possible that DLSRCHETS can be solved in polynomial time? Does imposition of some additional constraints make it tractable? What are those conditions?

7.2.2 *Extending the System Model*

This area has a large number of possibilities for future work. Scheduling purists may consider the system model used in this thesis to be quite simplistic.

As future work, the conditions (constraints on values of E_k and C_k), that minimize the error need to be found. An interesting area would be the investigation of the effect of affine cost models, processor deadlines and release times. Another important area would be to extend the results to multi-installment delivery and multi-level processor trees.

7.2.3 *Modification of DLSRCHETS*

Just some of the ways in which DLSRCHETS may be modified are listed below. Primarily the nature of DLSRCHETS is changed by consideration of:

- Stochasticity (dynamism) and uncertainty in the system parameters
- Non-clairvoyance, non-omniscience of the master
- Node (slave) turnover (failure)
- Slave sharing
- Multiple jobs on one master
- Multiple masters

- Multiple jobs on several masters
- Decentralization of scheduling decision (P2P model)
- QoS requirements
- Buffer, bandwidth, and computation constraints on slaves

Each of these modifications is a research topic in itself and as can be seen, there are plenty of research opportunities.

7.2.4 Application Development

All the testing in this thesis has been carried out using simulations. It will be interesting to see how the algorithms perform in practice. New and different applications apart from the number of possible scientific applications mentioned in the introduction, need to be developed that use the results in this thesis. This may require development of new libraries and middleware to support the computation models considered.

REFERENCES

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10, New York, NY, USA, 2003. ACM. ISBN 1-58113-661-7. doi: http://doi.acm.org/10.1145/777412.777414.

[2] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, Nov. 2004. URL http://boinc.berkeley.edu/grid_paper_04.pdf.

[3] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Singapore, May 2006. URL http://boinc.berkeley.edu/boinc_papers/internet/paper.pdf.

[4] D. P. Anderson and J. McLeod VII. Local scheduling for volunteer computing. In *Workshop on Large-Scale, Volatile Desktop Grids (PCGrid 2007) held in conjunction with the IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, Mar. 2007. URL http://boinc.berkeley.edu/boinc_papers/sched/paper.pdf.

[5] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *First IEEE Intl. Conf. on e-Science and Grid Technologies*, Melbourne, Dec. 2005. URL http://boinc.berkeley.edu/boinc_papers/server_perf/server_perf.pdf.

[6] K. R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, New York, USA, 1974.

[7] E. Bampis, J.-C. Konig, and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. *Theoretical Computer Science*, 147(1-2):1–18, Aug. 1995.

[8] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):1–12, Apr. 2004.

[9] G. D. Barlas. Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees. *IEEE Trans. Parallel Distrib. Syst.*, 9(5):429–441, May 1998.

[10] J. Basney, M. Livny, and T. Tannenbaum. High throuhgput Monte Carlo. In *9th SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 1999. URL http://citeseer.ist.psu.edu/basney99high.html.

[11] S. Bataineh and B. Al-Asir. An efficient scheduling algorithm for divisible and indivisible tasks in loosely coupled multiprocessor systems. *Software Engineering Journal*, 9(1):13–18, Jan. 1994.

[12] S. Bataineh and T. G. Robertazzi. Performance limits for processors with divisible jobs. *IEEE Trans. Aerosp. Electron. Syst.*, 33(4):1189–1198, Oct. 1997.

[13] S. Bataineh, T.-Y. Hsiung, and T. G. Robertazzi. Closed form solutions for bus and tree networks of processors load sharing a divisible job. *IEEE Trans. Comput.*, 43(10): 1184–1196, Oct. 1994.

[14] O. Beaumont, A. Legrand, and Y. Robert. Optimal algorithms for scheduling divisible loads on heterogeneous systems. In *IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2003*, Nice Acropolis Convention Center, Nice, France, Apr. 2003.

[15] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Trans. Parallel Distrib. Syst.*, 16(3):207–218, Mar. 2005.

[16] O. Beaumont, L. Marchal, V. Rehn, and Y. Robert. FIFO scheduling of divisible loads with return messages under the one-port model. Research Report 2005-52, LIP, ENS Lyon, France, Oct. 2005.

[17] O. Beaumont, L. Marchal, and Y. Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. Research Report 2005-21, LIP, ENS Lyon, France, May 2005.

[18] O. Beaumont, L. Marchal, V. Rehn, and Y. Robert. FIFO scheduling of divisible loads with return messages under the one port model. In *Proc. Heterogeneous Computing Workshop HCW'06*, Apr. 2006.

[19] G. Berti, S. Benkner, J. W. Fenner, J. Fingberg, G. Lonsdale, S. E. Middleton, and M. Surridge. Medical simulation services via the grid. In *Proc. HealthGrid Workshop 2003*, Lyon, France, Jan. 2003.

[20] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997. ISBN 1-886529-19-1.

[21] V. Bharadwaj and N. Viswanadham. Suboptimal solutions using integer approximation techniques for scheduling divisible loads on distributed bus networks. *IEEE Trans. Syst., Man, Cybern. A*, 30(6):680–691, Nov. 2000.

[22] V. Bharadwaj, D. Ghose, and V. Mani. Optimal sequencing and arrangement in distributed single-level tree networks with communication delays. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):968–976, Sept. 1994.

[23] V. Bharadwaj, D. Ghose, and V. Mani. Multi-installment load distribution in tree networks with delays. *IEEE Trans. Aerosp. Electron. Syst.*, 31(2):555–567, Apr. 1995.

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

[25] V. Bharadwaj, 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*, 18(1):919–938, Jan. 2000.

[26] V. Bharadwaj, X. Li, and C. C. Ko. On the influence of start-up costs in scheduling divisible loads on bus networks. *IEEE Trans. Parallel Distrib. Syst.*, 11(12):1288–1305, Dec. 2000.

[27] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible Load Theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, Jan. 2003.

[28] J. Blazewicz and M. Drozdowski. Distributed processing of divisible jobs with communication startup costs. *Discrete Applied Mathematics*, 76(1-3):21–41, June 1997.

[29] J. Blazewicz, M. Drozdowski, B. Soniewicki, and R. Walkowiak. Two-dimensional cutting problem: basic complexity results and algorithms for irregular shapes. *Foundations of Control Engineering*, 14(4):137–160, 1989.

[30] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag Telos, New York, USA, second edition, 2001.

[31] A. Bonhomme and L. Prylli. Performance evaluation of a distributed video storage system. In *Proc. IEEE IPDPS'02*, pages 126–135, Fort Lauderdale, FL, USA, Apr. 2002.

[32] R. H. Bush, G. D. Power, and C. E. Towne. WIND: The production flow solver of the NPARC alliance. In *36th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, Jan. 1998. AIAA Paper 98-0935. URL http://www.grc.nasa.gov/WWW/winddocs/aiaa98/aiaa-98-0935.html.

[33] H. Casanova and L. Marchal. A network model for simulation of grid application. Technical Report 4596, LIP, ENS Lyon, France, Oct. 2002.

[34] Y.-C. Cheng and T. G. Robertazzi. Distributed computation for a tree network with communication delays. *IEEE Trans. Aerosp. Electron. Syst.*, 26(3):511–516, May 1990.

[35] V. Chvátal. *Linear Programming*. W. H. Freeman, 1983. ISBN 0716715872.

[36] N. Comino and V. L. Narasimhan. A novel data distribution technique for host-client type parallel applications. *IEEE Trans. Parallel Distrib. Syst.*, 13(2):97–110, Feb. 2002.

[37] G. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1963.

[38] T. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. ReMoS: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, School of Computer Science, Carnegie Mellon University, 1997. URL http://citeseer.ist.psu.edu/dewitt97remos.html.

[39] M. Drozdowski. *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems.* Politechnika Poznanska, Book No. 321, Poznan, Poland, 1997.

[40] J. Edward Grady Coffman, editor. *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, New York, USA, 1976.

[41] J. Edward Grady Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, N.J., USA, 1973.

[42] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS, pages 2–13. Springer-Verlag, 2006.

[43] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2): 115–128, Aug. 1997.

[44] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, Aug. 2001.

[45] A. Galstyan, K. Czajkowski, and K. Lerman. Resource Allocation in the Grid Using Reinforcement Learning. In *Intl. Jt. Conf. on Autonomous Agents and Multiagent Systems*, volume 3, pages 1314–1315, 2004.

[46] A. Ghatpande, O. Beaumont, H. Nakazato, and H. Watanabe. Divisible load scheduling with result collection on heterogeneous systems. In *Proc. Heterogeneous Computing Workshop (HCW 2008) held in the IEEE Intl. Parallel and Distributed Processing Sysmposium (IPDPS 2008)*, Miami, FL., Apr. 2008.

[47] A. Ghatpande, H. Nakazato, O. Beaumont, and H. Watanabe. SPORT: An algorithm for divisible load scheduling with result collection on heterogeneous systems. *IEICE Transactions on Communications*, E91-B(8), Aug. 2008 (forthcoming).

[48] D. Ghose 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*, 55(1):32–59, Nov. 1998.

[49] D. Ghose and V. Mani. Distributed computation with communication delays: Asymptotic performance analysis. *Journal of Parallel and Distributed Computing*, 23 (3):293–305, Dec. 1994.

[50] GRIDSTART Initiative. GRIDSTART homepage. http://www.gridstart.org/index.shtml, June 2004.

[51] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide area computing: Resource sharing on a large scale. *IEEE Trans. Comput.*, 32(5):29–37, May 1999.

[52] E. Haddad. Optimal load sharing in dynamically heterogeneous systems. In *Proc. IEEE Symposium on Parallel and Distributed Processing 1995*, pages 346–353, San Antonio, TX, USA, Oct. 1995.

[53] W. A. Halang and K. Ramamritham. Real-time programming. In W. A. Halang and K. Ramamritham, editors, *Proc. ILFAC/IFIP Workshop on Real-Time Programming*, Atlanta, Georgia,USA, May 1991.

[54] J. T. Hung and T. G. Robertazzi. Scalable scheduling for clusters and grids using cut through switching. *Intl. J. of Computers and their Applications*, 26(3):147–156, 2004.

[55] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[56] H. J. Kim, G. Jee, and J. G. Lee. Optimal load distribution for tree network processors. *IEEE Trans. Aerosp. Electron. Syst.*, 32(2):607–612, Apr. 1996.

[57] K. Konstantinides, R. T. Kaneshiro, and J. R. Tani. Task allocation and scheduling models for multiprocessor digital signal processing. *IEEE Trans. Acoust., Speech, Signal Processing*, 38(12):2151–2161, Dec. 1990.

[58] D. Kranzlmuller, G. Kurka, P. Heinzlreiter, and J. Volkert. Optimizations in the grid visualization kernel. In *Proc. IEEE IPDPS'02*, pages 129–135, Fort Lauderdale, FL, USA, Apr. 2002.

[59] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. *IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2003*, Apr. 2003.

[60] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *National Conference on Artificial Intelligence*, pages 122–127, 1988. URL http://citeseer.ist.psu.edu/kumar88parallel.html.

[61] J. Landin. *An Introduction to Algebraic Structures*. Dover Publications, 1989. ISBN 0486659402.

[62] C.-H. Lee and K. G. Shin. Optimal task assignment in homogeneous networks. *IEEE Trans. Parallel Distrib. Syst.*, 8(2):119–129, Feb. 1997.

[63] X. Li, V. Bharadwaj, and C. C. Ko. Divisible load scheduling on single-level tree networks with buffer constraints. *IEEE Trans. Aerosp. Electron. Syst.*, 36(4):1298– 1308, Oct. 2000.

[64] B. Lowekamp, N. Miller, T. Gross, P. Steenkiste, J. Subhlok, and D. Sutherland. A resource query interface for network-aware applications. *Cluster Computing*, 2(2): 139–151, 1999. URL http://citeseer.ist.psu.edu/lowekamp99resource.html.

[65] V. Mani and D. Ghose. Distributed computation in linear networks: Closed-form solutions. *IEEE Trans. Aerosp. Electron. Syst.*, 30(2):471–483, Apr. 1994.

[66] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. Software Eng.*, 18(4):319–328, Apr. 1992.

[67] T. D. Nguyen, C. Peery, and J. Zahorjan. DDDDRRaW: A prototype toolkit for distributed real-time rendering on commodity clusters. In *IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2001*, San Francisco, CA, USA, Apr. 2001.

[68] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Trans. Comput.*, 26(2):62–76, Feb. 1993.

[69] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 2nd edition, 1998. ISBN 978-0486402581.

[70] D. A. L. Piriyakumar and C. S. R. Murthy. Distributed computation for a hypercube network of sensor-driven processors with communication delays including setup time. *IEEE Trans. Syst., Man, Cybern. A*, 28(3):245–251, Mar. 1998.

[71] G. N. S. Prasanna and B. R. Musicus. Generalized multiprocessor scheduling for directed acyclic graphs. In *Proc. IEEE Supercomputing 1994*, pages 237–246, Washington, DC, USA, Nov. 1994.

[72] T. V. Raman. Cloud computing and equal access for all. In *W4A '08: Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A)*, pages 1–4, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-153-8. doi: http://doi.acm.org/10.1145/1368044.1368046.

[73] T. Robertazzi. Divisible (partitionable) load scheduling research, Feb. 2008. URL http://www.ece.sunysb.edu/~tom/dlt.html#THEORY.

[74] T. G. Robertazzi. Processor equivalence for daisy chain load sharing processors. *IEEE Trans. Aerosp. Electron. Syst.*, 29(4):1216–1221, Oct. 1993.

[75] T. G. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, May 2003.

[76] M. Romberg. The UNICORE architecture: Seamless access to distributed resources. In *Proc. IEEE International Symposium on High Performance Distributed Computing HPDC 1999*, page 44, Redondo Beach, CA , USA, Aug. 1999.

[77] A. Rosenberg. Sharing partitionable workload in heterogeneous NOWs: Greedier is not better. In *IEEE International Conference on Cluster Computing*, pages 124–131, Newport Beach, CA, Oct. 2001.

[78] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[79] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. Software Eng.*, 18(8):736–748, Aug. 1992.

[80] F. J. Seinstra, D. Koelma, and J.-M. Geusebroek. A software architecture for user transparent parallel image processing. *Parallel Computing*, 28(7-8):967–993, Aug. 2002.

[81] T. Shepard and J. A. M. Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Software Eng.*, 17(7):669–677, July 1991.

[82] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A parallel computing approach to genetic sequence comparison: the master-worker paradigm with interworker communication. In *Computers and Biomedical Research*, volume 24, pages 152–169, 1991.

[83] V. J. R. Smith. UET scheduling with unit interprocessor communication selays. *Discrete Applied Mathematics*, 18(1):55–71, Sept. 1987.

[84] J. Sohn and T. G. Robertazzi. Optimal divisible job load sharing for bus networks. *IEEE Trans. Aerosp. Electron. Syst.*, 32(1):34–40, Jan. 1996.

[85] J. Sohn, T. G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):225–234, Mar. 1998.

[86] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[87] L. A. Steen, editor. *Mathematics Today: Twelve Informal Essays*. Springer-Verlag, New York, USA, 1978.

[88] A. S. Tanenbaum. *Operating Systems : Design and Implementation*. Prentice-Hall, Englewood Cliffs, N.J., USA, 1987.

[89] R. Taniguchi, Y. Makiyama, N. Tsuruta, S. Yonemoto, and D. Arita. Software platform for parallel image processing and computer vision. In *Proc. SPIE Parallel and Distributed Methods for Image Processing*, volume 3166, pages 2–10, July 1997.

[90] I. Taylor, M. Shields, I. Wang, and R. Philip. Distributed P2P computing within Triana: A galaxy visualization test case. In *IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2003*, Nice Acropolis Convention Center, Nice, France, Apr. 2003.

[91] Y. M. Teo, S. C. Low, S. C. Tay, and J. P. Gozali. Distributed geo-rectification of satellite images using grid computing. In *IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2003*, Nice Acropolis Convention Center, Nice, France, Apr. 2003.

[92] R. Thakur and W. Gropp. *Open Issues in MPI Implementation*, volume 4697 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007. URL http://www.springerlink.com/content/vx57711552l1262t/.

[93] The BOINC Project. BOINC project list. http://boinc.berkeley.edu/wiki/Project_list, June 2008.

[94] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*, volume 37 of *International Series in Operations Research & Management*. Kluwer Academic Publishers, 2nd edition, 2001. URL http://www.princeton.edu/~rvdb/LPbook/online.html.

[95] D. Weber, M. Spezialetti, and H. Barada. VidNet: Distributed processing environment for computer generated animation. *Software — Practice & Experience*, 26(2):237–250, 1996. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<237::AID-SPE13>3.3.CO;2-7.

[96] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007. ISSN 1091-3556. doi: http://doi.acm.org/10.1145/1327512.1327513.

[97] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley, 1999. ISBN 978-0-471-35943-2.

[98] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. IEEE International Symposium on High Performance Distributed Computing HPDC 1997*, pages 316–325, Portland, OR, USA, May 1997.

[99] R. Wolski. The network weather service: A distributed resource performance forecasting service for metacomputing. Technical Report CS98-599, University of California, San Diego, CA, USA, Oct. 1998.

[100] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared UNIX systems on the computational grid. In *Proc. IEEE International Symposium on High Performance Distributed Computing HPDC 1999*, pages 105–112, Redondo Beach, CA, USA, Aug. 1999.

[101] J. Xu. Multiprocessor scheduling of processors with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Software Eng.*, 19(2):139–154, Feb. 1993.

[102] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Trans. Software Eng.*, 19(1):70–84, Jan. 1993.

[103] D. Yu and T. G. Robertazzi. Scalable scheduling in parallel processors. In *Proc. Conference on Information Sciences and Systems*, Princeton, NJ, Mar. 2002.

[104] D. Yu and T. G. Robertazzi. Divisible load scheduling for grid computing. In *Proc. International Conference on Parallel and Distributed Computing Systems (PDCS 2003)*, volume 1, Los Angeles, CA, USA, Nov. 2003.

LIST OF PUBLICATIONS

REFEREED JOURNALS AND TRANSACTIONS

◯ A. Ghatpande, H. Nakazato, O. Beaumont, and H. Watanabe. Analysis of divisible load scheduling with result collection on heterogeneous systems. *IEICE Transactions on Communications*, E91-B(7):2234–2243, July 2008.

INTERNATIONAL CONFERENCES

◯ A. Ghatpande, O. Beaumont, H. Nakazato, and H. Watanabe. Divisible load scheduling with result collection on heterogeneous systems. In *Proc. Heterogeneous Computing Workshop (HCW 2008) held in the IEEE Intl. Parallel and Distributed Processing Sysmposium (IPDPS 2008)*, Miami, FL., Apr. 2008.

H. Watanabe, A. Ghatpande, and H. Nakazato. Distributed computing for real-time video processing. In *Proc. 1st International Conference on Ubiquitous Computing (ICUC 2003)*, pages 207–213, Seoul, Korea, Oct. 2003.

DOMESTIC ACADEMIC MEETINGS

S. Iwasaki, A. Ghatpande, H. Nakazato, H. Kanemitsu, T. Hoshiai, and H. Tominaga. A study of resource information exchange method on P2P-Grid. Technical Report NS2007-53, IEICE, Sept. 2007.

K. Kondou, A. Ghatpande, H. Nakazato, H. Kanemitsu, T. Hoshiai, and H. Tominaga. A study of data transferring for job execution time optimization on P2P-Grid. Technical Report NS2007-54, IEICE, Sept. 2007.

DOMESTIC CONFERENCES

B. Volodya, A. Ghatpande, and H. Nakazato. Regression based execution time estimation for scheduling in distributed computing systems. In *Proc. 2007 Forum on Information Technology (FIT 2007)*, L-025, Sept. 2007.

A. Ghatpande, H. Nakazato, and H. Watanabe. SPORT: Extended simulation results for divisible load scheduling on heterogeneous systems. In *Proc. 2006 IEICE Society Conference*, BS-15-3, Sept. 2006.

A. Ghatpande, H. Nakazato, and H. Watanabe. SPORT: A near-optimal solution to divisible load scheduling on heterogeneous systems. In *Proc. 2005 IEICE Society Conference*, BS-9-4, Sept. 2005.

A. Ghatpande, H. Nakazato, and H. Watanabe. Distributed video encoding on heterogeneous processor trees. In *Proc. 19th Picture Coding Symposium of Japan (PCSJ 2004)*, P-5.11, Nov. 2004.

A. Ghatpande, H. Nakazato, and H. Watanabe. An architecture for distributed video encoding on the Internet. In *Proc. 18th Picture Coding Symposium of Japan (PCSJ 2003)*, P-2.02, Nov. 2003.

POSTER PRESENTATIONS

A. Ghatpande and H. Nakazato. Bandwidth measurement in broadband networks for QoS guarantees. In *Wabot-House Symposium*, Gifu, Japan, Nov. 2007.

A. Ghatpande and H. Nakazato. Server architecture and HNML for networked home appliances. In *Wabot-House Symposium*, Gifu, Japan, Nov. 2007.

A. Ghatpande, H. Nakazato, and H. Watanabe. Grid over P2P systems: Issues and concepts. In *2006 Global Information and Telecommunication Research Workshop*, Saitama, Japan, Oct. 2006.

A. Ghatpande, H. Nakazato, and H. Watanabe. Distributed computing on P2P systems. In *Spring Symposium of the 21st Century COE Productive ICT Academia Program*, Tokyo, Japan, Mar. 2006.

A. Ghatpande, H. Nakazato, and H. Watanabe. Adaptive load scheduling using autonomous learning agents. In *2005 Global Information and Telecommunication Research Workshop*, Saitama, Japan, Nov. 2005.

A. Ghatpande, H. Nakazato, and H. Watanabe. Wide area distributed computing. In *2004 Global Information and Telecommunication Research Workshop*, Saitama, Japan, Nov. 2004.

ACRONYMS AND ABBREVIATIONS

AFS	All slaves Finish Simultaneously
CFD	Computational Fluid Dynamics
COTS	Cheap, Off-The-Shelf (hardware)
CPU	Central Processing Unit
DLS	Divisible Load Scheduling
DLSRCHETS	DLS with Result Collection on HETerogeneous Systems
DLT	Divisible Load Theory
FIFO	First In, First Out
FIFOC	First In, First Out, processors sorted by Communication time
FIFOE	First In, First Out, processors sorted by Execution time
GIS	Grid Information Service
ITERLP	ITERative Linear Programming
LHS	Left Hand Side (of an equation)
LIFO	Last In, First Out
LIFOC	Last In, First Out, processors sorted by Communication time
LIFOE	Last In, First Out, processors sorted by Execution time
LP	Linear Program or Linear programming Problem
MPI	Message Passing Interface
NaN	Not a Number (MATLAB™artifact)
NWS	Network Weather Service
OPT	OPTimal solution using brute force
OS	Operating System
P2P	Peer-to-Peer (computing, systems)
QoS	Quality of Service
RAM	Random Access Memory
ReMoS	Resource Monitoring System
RHS	Right Hand Side (of an equation)
SPORT	System Parameters based Optimized Result Transfer
SUMCE	Processor sorting by SUM of Communication and Execution time

LIST OF SYMBOLS

σ_a^{-1}	The inverse permutation of σ_a. It indicates the position of processor p_i in the allocation sequence.
σ_c^{-1}	The inverse permutation of σ_c. It indicates the position of processor p_i in the collection sequence.
α	The set of load fractions allocated to the slave processors; also known as a *load distribution*.
α^*	The optimal load distribution.
$\alpha_1, \ldots, \alpha_m$	The m elements of α that represent the load fractions allocated to $p_1, \ldots, p_m$.
$\alpha_1^*, \ldots, \alpha_m^*$	The m elements of α^* that represent the optimal load fractions to the processors $p_1, \ldots, p_m$.
α_1^f	The load fraction allocated to p_1 in a two processor schedule of type f.
α_1^g	The load fraction allocated to p_1 in a two processor schedule of type g.
α_1^l	The load fraction allocated to p_1 in a two processor schedule of type l.
α_2^f	The load fraction allocated to p_2 in a two processor schedule of type f.
α_2^g	The load fraction allocated to p_2 in a two processor schedule of type g.
α_2^l	The load fraction allocated to p_2 in a two processor schedule of type l.
$\alpha_{k:r}$	The load fraction allocated to the equivalent processor $p_{k:r}$.
ΔT_{VAR}	The percentage deviation from the optimal processing time for a variant algorithm with processing time of T_{VAR} for the same instance of the DLSRCHETS problem.
δ	The (constant) ratio of the size of output result data to the size of input allocated data for a job $\mathcal{J}$.

$\mathcal{J}$	The divisible load (job) to be processed on $\mathcal{H}$.
Λ_k	The set of all k-subsets of a set.
$\langle \Delta T_{\text{VAR}} \rangle$	The mean deviation from optimal for a variant algorithm calculated by averaging ΔT_{VAR} over all simulation runs.
$\mathcal{H}$	Heterogeneous master-slave system with $m+1$ processors and m network links.
$\prec_a$	A total order on $\mathcal{T}$ that represents the allocation sequence of the tasks.
$\prec_a^*$	The optimal order on the set of tasks $\mathcal{T}$.
$\prec_a^+$	The minimal element of $\prec_a$, i.e., the first task in the allocation sequence.
$\prec_a^-$	The maximal element of $\prec_a$, i.e., the last task in the allocation sequence.
$\prec_c$	A total order on $\mathcal{R}$ that represents the collection sequence of the results.
$\prec_c^*$	The optimal order on the set of results $\mathcal{R}$.
$\prec_c^+$	The minimal element of $\prec_c$, i.e., the first task in the collection sequence.
$\prec_c^-$	The maximal element of $\prec_c$, i.e., the last task in the collection sequence.
$\preccurlyeq_a$	The immediate predecessor or successor relation on $\mathcal{T}$. If $i \preccurlyeq_a j$, then task i is the immediate predecessor of task j in $\prec_a$. Equivalently, task j is the immediate successor of task i in $\prec_a$.
$\preccurlyeq_c$	The immediate predecessor or successor relation on $\mathcal{R}$. If $i \preccurlyeq_c j$, then task i is the immediate predecessor of task j in $\prec_c$. Equivalently, task j is the immediate successor of task i in $\prec_c$.
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^+$	The set of positive real numbers.
$\mathbb{R}_0^+$	The set of nonnegative real numbers.
$\mathbb{R}^d$	d dimensional Euclidean space.
$\rho_{k:r}$	The network parameter of the equivalent processor $p_{k:r}$.
$\rho_{k:r}^f$	The network parameter of the equivalent processor $p_{k:r}$ in a general FIFO schedule.

ρ_k	The network parameter for a processor p_k. It is equal to the ratio of the processor's unit computation speed to its unit communication speed.
$\mathcal{S}$	A schedule for the DLSRCHETS problem.
$\mathcal{C}$	The set of unit communication times of the slave processors.
$\mathcal{E}$	The set of unit computation times of the slave processors.
$\mathcal{L}$	The set of m network links in $\mathcal{H}$.
$\mathcal{P}$	The set of $m + 1$ processors in $\mathcal{H}$.
$\mathcal{R}$	The set of results that are collected from the processors $p_1, \ldots, p_m$.
σ_a	A permutation of order m that represents the allocation sequence.
σ_a^*	The optimal load allocation sequence.
Σ_a^k	The set of candidate allocation sequences for k processors in the ITERLP algorithm.
σ_c	A permutation of order m that represents the collection sequence.
σ_c^*	The optimal result collection sequence.
Σ_c^k	The set of candidate collection sequences for k processors in the ITERLP algorithm.
$\binom{n}{k}$	The binomial coefficient representing the number of ways of obtaining an unordered subset of k elements from a set of n elements.
$\mathbf{0}$	The null vector. The dimensions are context dependent.
$\boldsymbol{a}$	The d dimensional column vector whose elements are row coefficients of the constraint matrix $\boldsymbol{A}$, i.e., $\boldsymbol{A} = (\boldsymbol{a}_1^\top, \ldots, \boldsymbol{a}_n^\top)^\top$.
$\boldsymbol{b}$	The d dimensional right hand side column vector of constants in a linear programming problem.
$\boldsymbol{c}$	The d dimensional column vector of coefficients of the objective function in a linear programming problem.
$\boldsymbol{x}$	The d dimensional column vector of decision variables in a linear programming problem.
$\boldsymbol{A}$	The $n \times d$ dimensional coefficient or constraint matrix of the linear programming problem..

$\boldsymbol{u}^\top$	The transpose of any vector $\boldsymbol{u}$.
ζ	The objective function to be optimized (either maximized or minimized) in a linear programming problem.
$B^i_{\prec_a}$	The set of task i and the tasks before i (predecessors of i) in $\prec_a$.
$B^i_{\prec_c}$	The set of task i and the tasks before i (predecessors of i) in $\prec_c$.
C	Message transmission rate between two nodes.
$C_1, \ldots, C_m$	The m elements of $\mathcal{C}$ that represent unit communication times of $l_1, \ldots, l_m$.
$C^f_{1:2}$	The unit communication time for the equivalent processor $p_{1:2}$ in a two processor schedule f.
$C^g_{1:2}$	The unit communication time for the equivalent processor $p_{1:2}$ in a two processor schedule g.
$C^l_{1:2}$	The unit communication time for the equivalent processor $p_{1:2}$ in a two processor schedule l.
$C_{k:r}$	The unit communication time for the equivalent processor $p_{k:r}$.
$C^f_{k:r}$	The unit communication time for the equivalent processor $p_{k:r}$ in a general FIFO schedule.
$C^l_{k:r}$	The unit communication time for the equivalent processor $p_{k:r}$ in a general LIFO schedule.
D	The distance between two nodes in number of hops.
d	Dimension of a linear programming problem; the number of decision variables in the problem.
$E_1, \ldots, E_m$	The m elements of $\mathcal{E}$ that represent unit computation times of $p_1, \ldots, p_m$.
$E^f_{1:2}$	The unit computation time for the equivalent processor $p_{1:2}$ in a two processor schedule f.
$E^g_{1:2}$	The unit computation time for the equivalent processor $p_{1:2}$ in a two processor schedule g.
$E^l_{1:2}$	The unit computation time for the equivalent processor $p_{1:2}$ in a two processor schedule l.

$E_{k:r}$	The unit computation time for the equivalent processor $p_{k:r}$.
$E^f_{k:r}$	The unit computation time for the equivalent processor $p_{k:r}$ in a general FIFO schedule.
$E^l_{k:r}$	The unit computation time for the equivalent processor $p_{k:r}$ in a general LIFO schedule.
f	A permutation. It is represented by listing its values as $\{f_1, \ldots, f_n\}$.
f^{-1}	The inverse permutation of permutation f. It is represented as $\{f^{-1}(1), \ldots, f^{-1}(n)\}$.
$F^i_{\prec_a}$	The set of task i and the followers (successors) of task i in $\prec_a$.
$F^i_{\prec_c}$	The set of task i and the followers (successors) of task i in $\prec_c$.
I_c	The span of the set $\mathcal{C}$ for simulations.
I_e	The span of the set $\mathcal{E}$ for simulations.
K	The index set of slave processors in $\mathcal{H}$.
L	Message length being transmitted between two nodes.
$l_1, \ldots, l_m$	The m elements of $\mathcal{L}$ that represent the network links in $\mathcal{H}$.
m	The number of available slaves in $\mathcal{H}$.
n	The number of constraints in a linear programming problem. Also the number of slave processors allocated non-zero load (clarified by context).
p	The number of processors allocated non-zero load in the optimal solution, i.e., the number of participating processors or participants.
p_0	The master processor in $\mathcal{H}$.
$p_0, \ldots, p_m$	The $m+1$ elements of $\mathcal{P}$ that represent the processors in $\mathcal{H}$.
$p_1, \ldots, p_m$	The m slave processors in $\mathcal{H}$.
$p_{k:r}$	The equivalent processor for processors $p_k, \ldots, p_r$.
r	Result collection start time function. Transmission of result data from p_i to p_0 begins at time r_i in a schedule $\mathcal{S}$.
S	Message start-up time before data transmission between two nodes.

S_n	The symmetric group of degree n, i.e., the group of all permutations on n symbols.
T	The total processing time for a job $\mathcal{J}$ from the point when the master first initiates the allocation of tasks, to the point when the master completes reception of all the results.
t	Task allocation start time function. Transmission of load fraction to p_i from p_0 begins at time t_i in a schedule $\mathcal{S}$.
T^1	The total processing time if the entire load is allocated to processor p_1 only.
T^2	The total processing time if the entire load is allocated to processor p_2 only.
T^f	The total processing time of a two processor schedule of type f.
T^g	The total processing time of a two processor schedule of type g.
T^l	The total processing time of a two processor schedule of type l.
t_i^{send}	The time at which processor p_i starts sending its result data to the master in a feasible schedule.
t_i^{stop}	The time at which processor p_i stops execution of its load fraction in a feasible schedule.
$T_{1:2}^f$	The total processing time for the equivalent processor $p_{1:2}$ in schedule f.
$T_{1:2}^g$	The total processing time for the equivalent processor $p_{1:2}$ in schedule g.
$T_{1:2}^g$	The total processing time for the equivalent processor $p_{1:2}$ in schedule g.
$T_{1:2}^l$	The total processing time for the equivalent processor $p_{1:2}$ in schedule l.
t_{comm}	The total communication time of a data message between two nodes.
T_{OPT}	The optimal processing time for an instance of the DLSRCHETS problem. In simulations, this value is obtained by solving linear programs.
T_{VAR}	The processing time of a variant algorithm used for comparison in simulations.
t_i^{exec}	The time at which processor p_i completes reception of its data and starts execution in a feasible schedule.
t_i^{recv}	The time at which processor p_i starts receiving is data in a feasible schedule (equivalent to t_i in $\mathcal{S}$).

$T_{k:r}$	The total processing time of the equivalent processor $p_{k:r}$.
$T^f_{k:r}$	The total processing time of the equivalent processor $p_{k:r}$ in a general FIFO schedule.
$T^l_{k:r}$	The total processing time of the equivalent processor $p_{k:r}$ in a general LIFO schedule.
U	The commutation time per switch in a network.
x_i	The idle time in a processor p_i.
y	The intervening time interval between the end of allocation phase of the last processor in the allocation sequence and the start of result collection from the first processor in the collection sequence.
${}_nP_k$	The number of ways of obtaining an ordered subset of k elements from a set of n elements.
$\mathcal{T}$	The set of tasks corresponding to the m load fractions.