Dynamic Shortest Paths Algorithms: Parallel Implementations and Application to the Solution of Dynamic Traffic Assignment Models

by

SRIDEVI V. GANUGAPATI

B.Tech., Indian Institute of Technology, Madras (1996)

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Transportation Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

© Massachusetts Institute of Technology 1998. All rights reserved.

Author ....
Department of Civil and Environmental Engineering
May 22, 1998

Certified by...
Ismail Chabini
Assistant Professor
Thesis Supervisor

Accepted by ....
Joseph M. Sussman
Chairman, Department Committee on Graduate Students

JUN 02 1998

Eng.

Dynamic Shortest Paths Algorithms: Parallel Implementations and Application to the Solution of Dynamic Traffic Assignment Models

by

SRIDEVI V. GANUGAPATI

Submitted to the Department of Civil and Environmental Engineering
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Transportation Engineering

Abstract

Intelligent Transportation Systems (ITS) promise to improve the efficiency of the transportation networks by using advanced processing, control and communication technologies. The analysis and operation of these systems require a variety of models and algorithms. Dynamic shortest paths problems are fundamental problems in the solution of most of these models. ITS solution algorithms should run faster than real time in order for these systems to operate in real time. Optimal sequential dynamic shortest paths algorithms do not meet this requirement for real size networks. High performance computing offers an opportunity to speedup the computation of dynamic shortest path solution algorithms. The main goals of this thesis are (1) to develop parallel implementations of optimal sequential dynamic shortest paths algorithms and (2) to apply optimal sequential dynamic shortest paths algorithms to solve dynamic traffic assignment models that predict network conditions in support of ITS applications.

This thesis presents parallel implementations for two parallel computing platforms:(1) Distributed Memory and (2) Shared Memory. Two types of parallel codes are developed based on two libraries: PVM and Solaris Multithreading. Five decomposition dimensions are exploited: (1) destination, (2) origin, (3) departure time interval, (4) network topology and (5) data structure. Twenty two triples of (parallel computing platform, algorithm, decomposition dimension) computer implementations are analyzed and evaluated for large networks. Significant speedups of sequential algorithms are achieved, in particular, for shared memory platforms. Dynamic shortest path algorithms are integrated into the solution algorithms of dynamic traffic assignment models. An improved data structure to store paths is designed. This data structure promises to improve the efficiency of DTA algorithms implementations.

Thesis Supervisor: Ismail Chabini
Title: Assistant Professor

Acknowledgements

I am immensely grateful to Professor Ismail Chabini for his guidance and encouragement throughout the course of my study. I have benefitted a lot through the numerous research discussions we had in the past one and half years. The supervision of Professor Ismail Chabini not only helped me grow intellectually but also as a student, skilled professional and most importantly as an educated person.

I also want to recognise and thank many of my colleagues at MIT for their assistance: Brian Dean for very useful discussions on shortest path algorithms and also for the computer codes on one-to-all dynamic shortest path problems, Parry Husbands and Fred Donovan for their assistance with the xolas machines, Yiyi He for the DTA codes, Qi Yang for his assistance with PVM, Jon Bottom for the discussions we had about path data structures.

I would like to thank my friends in CTS, particularly people in ITS and 5-008 offices for making me feel at home while at MIT.

I do not find enough words, when I want to thank my family, Amma, Daddy, Suri and Anil, for their love, affection and support. I owe everything to them.

I dedicate my work to my parents Rama and Hymavathi,

my sister Suri and my brother Anil.

Contents

Acknowledgements **3**

Contents **5**

List of Figures **10**

List of Tables **15**

1 Introduction **16**

1.1 Research Objectives 17

1.2 Thesis Organization 18

2 Dynamic Shortest Paths - Formulations, Algorithms and Implementations **19**

2.1 Classification 20

2.2 Representation of a Dynamic Network Using Time Space Expansion . 22

2.3 Definitions and Notation 24

2.4 One to All Fastest Paths for One Departure Time in FIFO Networks when Waiting is Forbidden at All Nodes 25

2.4.1 Formulation 26

2.5 One to All Fastest Paths Problem for One Departure Time in non-FIFO Networks with Waiting Forbidden at All Nodes 27

2.5.1 Formulation 28

2.5.2 Algorithm `IOT` 31

2.5.3 Complexity of algorithm IOT 32
2.6 One to All Fastest Paths for One Departure Time when Waiting at Nodes is Allowed 33
2.7 All-to-One Fastest Paths for All Departure Times 35
2.7.1 Formulation 35
2.7.2 Algorithm DOT 36
2.7.3 Complexity of algorithm DOT 37
2.8 One-to-All Minimum Cost Paths for One Departure Time 38
2.8.1 Formulation 38
2.8.2 Algorithm IOT-MinCost 40
2.8.3 Complexity of algorithm IOT-MinCost 41
2.9 All-to-One Minimum Cost Paths Problem for all Departure Times Problem 42
2.9.1 Formulation 42
2.9.2 Algorithm DOT-MinCost 42
2.9.3 Complexity of algorithm DOT-MinCost 43
2.10 Experimental Evaluation 44
2.11 Summary 51

3 Parallel Computation 63
3.1 Classification of Parallel Systems 64
3.2 Parallel Computing Systems Used in This Thesis 68
3.3 How is Parallel Computing different from Serial Computing? 69
3.4 How to Develop a Parallel Implementation? 71
3.4.1 Decomposition Methods 71
3.4.2 Master/Slave Paradigm 72
3.4.3 Software Development Tools 73
3.5 Performance Measures 81

4 Parallel Implementations of Dynamic Shortest Paths Algorithms 84
4.1 Parallel Implementations: Overview 85

4.2 Notation 93
4.3 Distributed Memory Application Level Parallel Implementations . . . 94
4.3.1 Master process algorithm 95
4.3.2 Slave process algorithm 96
4.3.3 Run Time Analysis 97
4.4 Shared Memory Application Level Parallel Implementations 98
4.4.1 Master thread algorithm 98
4.4.2 Slave thread algorithm 99
4.4.3 Run Time Analysis 99
4.5 Experimental Evaluation of Application Level Parallel Implementations 100
4.5.1 Numerical tests and results 100
4.5.2 Conclusions 107
4.6 Distributed Memory Implementation of Algorithm DOT by Decomposition of Network Topology 109
4.6.1 Notation 109
4.6.2 Master process algorithm 109
4.6.3 Slave process algorithm 110
4.6.4 Run Time Analysis 112
4.7 Shared Memory Implementation of Algorithm DOT by Decomposition of Network Topology 113
4.7.1 Notation 113
4.7.2 Master thread algorithm 114
4.7.3 Slave thread algorithm 114
4.7.4 Run Time Analysis 115
4.8 Experimental Evaluation of Algorithm Level Parallel Implementations 116
4.8.1 Numerical tests 117
4.8.2 Conclusions 118
4.9 Idealized Parallel Implementation of Algorithm `DOT` 119
4.9.1 Notation 119
4.9.2 Algorithm `DOT-parallel` 120

4.9.3 Computing the minimum using a complete binary tree 120
4.9.4 Run time analysis of algorithm DOT-parallel 121
4.10 Shared Memory Implementation of Algorithm IOT by Decomposition of Data Structure 123
4.10.1 Notation 123
4.10.2 Master thread algorithm 124
4.10.3 Slave thread algorithm 125
4.11 Summary 127

5 Application of Dynamic Shortest Paths Algorithms to the Solution of Dynamic Traffic Assignment Models 150
5.1 Introduction 150
5.2 A Conceptual Framework for Dynamic Traffic Assignment Problem . 152
5.3 Subpath Data structure 153
5.4 Integration of Dynamic Shortest Path Algorithms into Analytical Dynamic Traffic Assignment Model Framework 156
5.4.1 Notation 156
5.4.2 Algorithm UPDATE 157
5.4.3 Algorithm INTEGRATE 158
5.5 Experimental Evaluation 158
5.5.1 Conclusions 165
5.6 Subpath Tree - A New Implementation of the Subpath Data Structure 168
5.6.1 Adding a new path to the subpath tree 168
5.7 Summary 172

6 Summary and Future Directions 173
6.1 Summary 173
6.2 Future Directions 175
6.2.1 Hybrid Parallel Implementations 175
6.2.2 Network Decomposition Techniques 175
6.2.3 More decomposition strategies 175

6.2.4 Implementation and evaluation of the `subpath tree` data structure . . . 176
6.2.5 Parallel implementation of the DTA software system 176

A Parallel Implementations 177
A.1 PVM Implementations 177
A.2 MT implementations 179

B Time-Dependent Path Generation Module in Dynamic Traffic Assignment 181

References 183

List of Figures

2.1 Representation of a dynamic network using time space expansion . . 23
2.2 Illustration of forward labeling for non-FIFO networks 30
2.3 Algorithms for FIFO networks 51
2.4 Algorithm IOT: varying maximum link travel time 52
2.5 Algorithm IOT: varying number of nodes 52
2.6 Algorithm IOT: varying number of links 53
2.7 Algorithm IOT: varying number of time intervals 53
2.8 Algorithm IOT-MinCost: varying number of nodes 54
2.9 Algorithm IOT-MinCost: varying number of links 54
2.10 Algorithm IOT-MinCost: varying number of time intervals 55
2.11 Algorithm IOT-MinCost: varying maximum link travel time 55
2.12 Algorithm IOT-MinCost: varying maximum link cost 56
2.13 Algorithm DOT: varying number of nodes) 56
2.14 Algorithm DOT: varying number of links 57
2.15 Algorithm DOT: varying number of time intervals 57
2.16 Algorithm DOT-MinCost: varying number of nodes 58
2.17 Algorithm DOT-MinCost: varying number of links 58
2.18 Algorithm DOT-MinCost: varying number of time intervals 59
2.19 Comparison of algorithm IOT and algorithm DOT 59
2.20 Comparison of algorithm IOT-MinCost and algorithm DOT-MinCost . 60

3.1 Shared Memory System 66
3.2 Distributed Memory System 67

3.3 The Solaris Multithreaded Architecture (Courtesy: Berg and Lewis [24]) 77

4.1 Application Level Parallel Implementations 89
4.2 Algorithm Level Parallel Implementations 90
4.3 Illustration of implementation on a distributed memory platform of an application level (destination-based) decomposition strategy 91
4.4 Illustration of implementation on a shared memory platform of an application level (destination-based) decomposition strategy 91
4.5 Illustration of Decomposition by Network Topology 92
4.6 Computing Minimum with a Complete Binary Tree 121
4.7 Decomposition by Destination(with collection of results by the master process in PVM implementations) 128
4.8 Performance of (PVM, Destination, `DOT` Implementation: varying number of nodes 129
4.9 Performance of (PVM, Destination, `DOT` Implementation: varying number of links 129
4.10 Performance of (PVM, Destination, `DOT` Implementation: varying number of time intervals 130
4.11 Decomposition by Destination(without collection of results by the master process in PVM implementations) 130
4.12 Decomposition by Origin (FIFO networks) 131
4.13 Decomposition by Origin (non-FIFO networks) 131
4.14 Decomposition by Departure Time (FIFO networks) 132
4.15 Decomposition by Departure Time (non-FIFO networks) 132
4.16 Performance of (PVM (without collection of results by the master process), Destination, `DOT`) Implementation: varying number of nodes . . 133
4.17 Performance of (PVM (without collection of results by the master process), Destination, `DOT`) Implementation: varying number of links . . 133

4.18 Performance of (PVM (without collection of results by the master process), Destination, DOT) Implementation: varying number of time intervals 134
4.19 Performance of (MT, Destination, DOT) Implementation: varying number of nodes 134
4.20 Performance of (MT, Destination, DOT) Implementation: varying number of links 135
4.21 Performance of (MT, Destination, DOT) Implementation: varying number of time intervals 135
4.22 Performance of (PVM, origin, lc-dequeue) Implementation: varying number of nodes 136
4.23 Performance of (PVM, origin, lc-dequeue) Implementation: varying number of links 136
4.24 Performance of (PVM, origin, lc-dequeue) Implementation: varying number of time intervals 137
4.25 Performance of (PVM, origin, IOT) Implementation: varying number of nodes 137
4.26 Performance of (PVM, origin, IOT) Implementation: varying number of links 138
4.27 Performance of (PVM, origin, IOT) Implementation: varying number of time intervals 138
4.28 Performance of (MT, origin, lc-dequeue) Implementation: varying number of nodes 139
4.29 Performance of (MT, origin, lc-dequeue) Implementation: varying number of links 139
4.30 Performance of (MT, origin, lc-dequeue) Implementation: varying number of time intervals 140
4.31 Performance of (MT, origin, IOT) Implementation: varying number of nodes 140

4.32 Performance of (MT, origin, IOT) Implementation: varying number of links 141
4.33 Performance of (MT, origin, IOT) Implementation: varying number of time intervals 141
4.34 Performance of (PVM, Departure Time, lc-dequeue) Implementation: varying number of nodes 142
4.35 Performance of (PVM, Departure Time, lc-dequeue) Implementation: varying number of links 142
4.36 Performance of (PVM, Departure Time, lc-dequeue) Implementation: varying number of time intervals 143
4.37 Performance of (PVM, Departure Time, IOT) Implementation: varying number of nodes 143
4.38 Performance of (PVM, Departure Time, IOT) Implementation: varying number of links 144
4.39 Performance of (PVM, Departure Time, IOT) Implementation: varying number of time intervals 144
4.40 Performance of (MT, Departure Time, lc-dequeue) Implementation: varying number of nodes 145
4.41 Performance of (MT, Departure Time, lc-dequeue) Implementation: varying number of links 145
4.42 Performance of (MT, Departure Time, lc-dequeue) Implementation: varying number of time intervals 146
4.43 Performance of (MT, Departure Time, IOT) Implementation: varying number of nodes 146
4.44 Performance of (MT, Departure Time, IOT) Implementation: varying number of links 147
4.45 Performance of (MT, Departure Time, IOT) Implementation: varying number of time intervals 147
4.46 Decomposition by Network Topology 148

4.47 MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of nodes) 148
4.48 MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of links) 149
4.49 MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of time intervals) 149

5.1 A Framework for Dynamic Traffic Assignment Models 154
5.2 A simple network to illustrate the subpath table data structure . . . 155
5.3 Algorithm `INTEGRATE` 159
5.4 Test Network 160
5.5 Amsterdam Beltway Network 167
5.6 Sub Path Tree for the network in Figure 5.4 169
5.7 Updated Sub Path Tree for the network in Figure 5.4 170
5.8 An extended sub path tree 171

A.1 Directory Structure 178

B.1 Flowchart of the Integration of Path Generation Module into Dynamic Traffic Assignment Software 182

List of Tables

5.1 The subpath table for the network shown in Figure 5.2 155

5.2 Initial subpath table for the network in Figure 5.4 162

5.3 Updated subpath table for the network in Figure 5.4 163

5.4 Path Travel Times (minutes) for OD pair (1,9) before the path generation 164

5.5 Path Travel Times (minutes) for OD pair (1,9) after the path generation 164

5.6 Path Flows for OD pair (1,9) before the path generation 165

5.7 Path Flows for OD pair (1,9) after the path generation 166

5.8 Performance measures before and after the path generation 166

5.9 Performance of the time dependent path generation component on the Amsterdam Network 166

Chapter 1

Introduction

Today's transportation infrastructure is often associated with congestion, inefficiency, danger and pollution. Traffic congestion costs society a lost in productivity. One way to solve the problems associated with traffic systems is to construct more highways. This solution is increasingly more expensive and is not always feasible because of spatial and environmental limitations, especially in urban areas.

A new way to reduce congestion problems is to use the existing infrastructure more efficiently through better traffic management by equipping transportation systems with information technologies. This is known as Intelligent Transportation Systems (ITS). These systems are based on integrating advances in sensing, processing, control and communication technologies.

The two building blocks of ITS are Advanced Traffic Management Systems (ATMS) and Advanced Traveler Information Systems (ATIS). ATMS is expected to integrate management of various roadway functions, predict traffic congestion and provide alternative routing instructions to users and transit operators. Real time data will be collected and disseminated. Dynamic traffic control systems will respond in real time to changing traffic conditions. Incident detection is seen as a critical function of ATMS. ATIS involves providing data to travelers in their vehicle, in their home or at their place of work. Users can make their travel choices based on the information provided by ATIS.

To achieve their goals, both ATIS and ATMS require certain decision support

models and algorithms. In order to meet the real-time operational requirement of ATIS/ATMS, such algorithms must run much faster than real time.

Dynamic shortest paths problems are fundamental problems in the solution of the network models that support ITS applications. These shortest path problems are different from the conventional static shortest paths problems since in ITS applications, networks are dynamic. Optimal sequential dynamic shortest paths algorithms do not compute dynamic shortest paths fast enough for ITS applications. High performance computing gives an opportunity to speedup the computation of these algorithms. In this thesis, we design, develop and evaluate various parallel implementations of dynamic shortest path algorithms.

Dynamic Traffic Assignment (DTA) models are used to predict network conditions to support ITS applications. Dynamic shortest paths algorithms are required to optimally solve DTA models. In this thesis, we apply dynamic shortest paths algorithms to the solution of analytical formulations of dynamic traffic assignment problems.

1.1 Research Objectives

The goals of this thesis are:

- to review optimal sequential dynamic shortest path algorithms and to report on the experimental evaluation of efficient computer implementations of these algorithms,
- to develop parallel implementations of optimal sequential dynamic shortest paths algorithms,
- to apply the dynamic shortest paths algorithms to the solution of analytical dynamic traffic assignment problems.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents formulations, algorithms and experimental evaluations of sequential computer implementations for dynamic shortest path problems. Chapter 3 introduces parallel computation concepts required to develop the parallel implementations. Chapter 4 presents twenty two parallel implementations of optimal sequential dynamic shortest path algorithms. Chapter 5 describes the application of algorithm `DOT` to the solution of analytical formulations of DTA models developed by Chabini and He [8]. Chapter 6 summarizes the main conclusions of this research and suggests some directions for future research.

Chapter 2

Dynamic Shortest Paths - Formulations, Algorithms and Implementations

Shortest path problems have been studied extensively in the literature for many years due to their importance in many engineering and scientific fields. With the advent of Intelligent Transportation Systems (ITS), a class of the shortest path problems has become important. These are known as dynamic shortest path problems. In these problems, the travel time of the link varies with starting time on the link. This varying travel time adds a new dimension to the shortest path problems, thus, increasing their complexity.

Dynamic shortest path problem has been first proposed by Cooke and Halsey [12] about 30 years ago. Recently, Chabini [10] solved one of the variants of this problem by proposing an optimal algorithm, that is, no other algorithm with a better running time complexity can be found.

In this chapter, we present the formulations, efficient algorithms and an extensive evaluation of dynamic shortest path problems, relevant to traffic networks with the following objectives:

- To understand the efficient algorithms available to solve dynamic shortest path

problems since these problems are critical in many transportation applications.

- To demonstrate that these sequential optimal algorithms do not solve realistic dynamic shortest path problems fast enough for Intelligent Transportation Systems applications. We will then conclude that an application of high performance computing is essential for solving realistic dynamic shortest path problems in real time.

This chapter is organized as follows: In Section 2.1, we develop a classification of dynamic shortest path problems. This classification is important to formulate these problems and to develop solution algorithms and computer implementations for them. In Section 2.2, we present a representation of the network using a time-space expansion. This representation is useful in presenting certain properties of the dynamic networks. These properties will be used to develop efficient solution algorithms. In Section 2.3, we define the notation used in formulations and algorithms of dynamic shortest path problems. Sections 2.4 through 2.9 are the crux of this chapter. In these sections, we present the formulations and algorithms for different variants of the dynamic shortest problems. In Section 2.10, we present an extensive experimental evaluation of the computer implementations of the dynamic shortest path algorithms discussed in this chapter. Finally, Section 2.11 summarizes the conclusions of this chapter and motivates the need for parallel implementations of dynamic shortest path problems.

2.1 Classification

Dynamic shortest path problems can be classified into many types. This classification is important for formulations and solution algorithms of these problems. Chabini [10] presents the following classification of the dynamic shortest path problems:

- **Minimum time vs. Minimum cost**: This classification is based on the minimization criterion. For minimum-time problems, one wishes to find the

least travel time between a pair of nodes in the network. For minimum cost problems, a path with the least cost is desired.

- **Discrete vs. Continuous time networks**: This classification is based on how time is represented. For discrete time networks, time is represented in discrete intervals. The link travel times and costs within this interval are assumed to be constant. A discrete dynamic network can alternatively be viewed as a static network, using a time-space expansion representation (see Figure 2.2). This idea is further discussed in Section 2.2.

- **Integer valued vs. real valued link travel times**: In integer time networks, travel times are assumed as positive integers. In real-valued time networks, travel times can take real values.

- **FIFO vs non-FIFO networks**: First-In-First-Out (FIFO) networks are those in which it is ensured that a vehicle entering a link earlier than another vehicle will leave that link earlier. It is also referred to as the no-overtaking condition. A mathematical definition of the FIFO condition is given in a later section. This condition may be used to develop efficient algorithms for certain dynamic shortest path problems.

- **Waiting is allowed vs waiting is not allowed at nodes**: Unlimited waiting or limited waiting may be permitted at all the nodes or at a subset of nodes in a network.

- **Type of shortest paths questions asked**: The models and algorithms depend on the kind of shortest paths questions asked. For instance, one may want to determine shortest paths from one origin node to all the nodes in the network or from all the nodes to one destination node, for one departure time interval or for many departure time intervals. These basic problems can be extended to many-to-all or all-to-many problems.

  The following two dynamic shortest path questions are most relevant for transportation applications:

- Question 1: What are the shortest paths from one origin node to all the other nodes in the network departing the origin node at a given instant, say 0?
- Question 2: What are the shortest paths from all nodes to a destination node in the network, for all departure times?

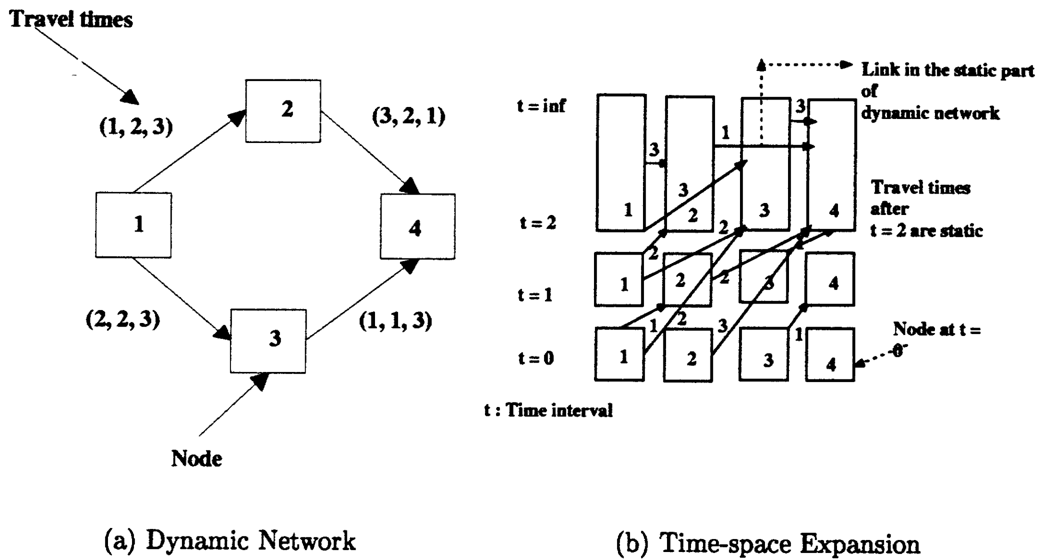
Some of the work published in the area of dynamic shortest paths has dealt with Question 1 ([15], [23], [25]) and some with Question 2 ([12], [30], [10]). Algorithms that answer Question 1 or Question 2 can be used for a specific problem. One such specific problem is a dynamic shortest path problem in a traffic network. For traffic problems, we will demonstrate that algorithms that answer Question 2 are more efficient.

Most transportation problems need shortest paths from all nodes to many destinations in the network. For instance, Let us assume that algorithm $alg1$ answers Question 1 and that $alg2$ answers Question 2. A typical traffic network can have approximately 3000 nodes, 9000 arcs and about 300 destinations. If the time frame is discretized into 100 time intervals, then 100 iterations of $alg1$ will be 10 times slower than $alg2$ for this network (as will be seen in Section 2.10).

In the following section, we describe a way of representing dynamic networks, which helps us present certain properties of the dynamic networks, which are useful in developing efficient algorithms. These properties can be used to develop efficient algorithms.

2.2 Representation of a Dynamic Network Using Time Space Expansion

A discrete dynamic network can be represented as a static network using a time space expansion. The time space expansion representation helps us understand certain properties of the dynamic networks for example, the acyclic property of the dynamic network along the time dimension.



(a) Dynamic Network

(b) Time-space Expansion

Figure 2.1: Representation of a dynamic network using time space expansion

Figure 2.1 illustrates this representation for a small dynamic network. Figure 2.1a, shows a small dynamic network with 4 nodes, 4 links and 3 time intervals. The numbers on the links denote the travel times of the link for each time interval. For instance, $(1, 1, 3)$ on link $(3, 4)$ denote that the travel time on link $(3, 4)$ 1 at time interval 0, 1 at time interval 1 and at time interval 2 is 3. The time-space expanded representation of this dynamic network is shown in Figure 2.1b. In this representation, each node in the dynamic network is represented as 4 copies, one at each time interval. Each row in the Figure 2.1b represents one time interval. In the dynamic network, the travel times are assumed to be static after 3 time intervals, hence the nodes at time interval $t = 2$ can be considered as extending to infinity. Each link in the dynamic network is also represented by four copies in the time-space expanded network, but in this representation, the links connect nodes at different time intervals. For example, one copy of the link between nodes 1 and 2 in the dynamic network with a travel time of 1 at time interval 0 is the link connecting node 1 at time interval 0 and node 2 at time interval 1. This shows that if we start at node 1 at time interval 0, we will reach node 2 at time interval 1.

A static shortest path algorithm can be applied on the network shown in Figure 2.1b to compute dynamic shortest paths in the network shown in Figure 2.1a.

This will be inefficient because the static shortest path algorithm will compute shortest paths labels for all nodes at all time intervals (Recall that these nodes are just copies of the node in the dynamic network). Hence, we need to find the minimum over all time intervals, for each node.

In the next section, we present the notation used in the formulations and algorithms for dynamic shortest path problems presented in this chapter.

2.3 Definitions and Notation

Let $G = (N, A, D, C)$ be a directed network where $N = \{1, ..., n\}$ is the set of nodes, $A = \{(i,j) | i \in N, j \in N\}$ is the set of links (or arcs). Let t be an index to an interval of time. We denote by $D = \{d_{ij}(t) | (i,j) \in A\}$, the set of time dependent link travel times and by $C = \{c_{ij}(t) | (i,j) \in A\}$, the set of time dependent link travel costs. Functions $d_{ij}(t)$ are assumed to have integer valued domain and range. The functions $d_{ij}(t)$ are assumed to take a static value after M time intervals. Hence, t can take values from 0 to $M-1$. Functions $c_{ij}(t)$ have integer valued domain and real valued range. $c_{ij}(t)$ are assumed to be static when departure time is greater than or equal to $M-1$. We denote by $B(i)$, the set of nodes having an outgoing arc into i, $B(i) = \{j \in N | (j,i) \in A\}$. Let $A(i)$ denote the set of nodes having an incoming arc from i.

Arc travel times can possess a property called FIFO (or *no-overtaking*) condition ([23]). This condition may be useful in developing algorithms for certain dynamic shortest path problems. The FIFO condition can be defined mathematically in various forms. For instance, FIFO condition is valid if and only if the following system of inequalities hold:

$$t + d_{ij}(t) \leqslant (t+1) + d_{ij}(t+1), \forall (i,j,t) \tag{2.1}$$

Intuitively, it can be seen that this condition holds if no overtaking takes place. If the link travel times of a link in the network satisfy the above condition, that link

is called a FIFO link. If all the links in the network satisfy the FIFO condition, the network is called a FIFO network, else, it is called a non-FIFO network. When the FIFO condition is not satisfied, it may be preferable to wait at the beginning of the link before traveling on the link.

In non-FIFO networks, if waiting is not allowed, we can have shortest paths with loops (which simulate waiting at the nodes). Therefore, two waiting policies need to be considered: *waiting is allowed at nodes* and *waiting is not allowed at nodes.*

In the rest of the chapter, we present formulations, algorithms and evaluation of computer implementations for two main dynamic shortest path problems: dynamic fastest path problems and dynamic minimum cost path problems. The two main questions discussed in each of these classes of problems are:

- Determine the dynamic fastest paths or minimum cost paths from one origin node in the network to all the nodes in the network for a given departure time at the origin node?
- Determine the dynamic fastest paths or minimum cost paths from all the nodes in the network to one destination node in the network for all departure time intervals?

For both these questions, we consider two kinds of dynamic networks: FIFO and non-FIFO. For each of these types of networks, we consider two kinds of waiting policies: unlimited waiting is allowed at all nodes and waiting is forbidden at all nodes.

2.4 One to All Fastest Paths for One Departure Time in FIFO Networks when Waiting is Forbidden at All Nodes

This is the most studied version of the dynamic shortest paths problem ([12], [15], [23]). A celebrated result is: *When FIFO condition is satisfied, any static shortest path*

algorithm can be generalized to solve the time dependent fastest path problem with the same time complexity as the static shortest paths problem. Dreyfus [15] was the first to present this generalization heuristically. He concluded that Dijkstra's algorithm can be adapted to solve the dynamic shortest path problem. Later, Ahn and Shin [1] and Kaufman and Smith [23] and a few others proved that this generalization is valid only if the FIFO condition is valid.

Chabini [10] gives an intuitive and simple proof of the generalization of the results, as opposed to a lengthy proof given by earlier authors. These are obtained from a formulation of the problem. These are presented in the next subsection.

2.4.1 Formulation

Let f_j denote the minimum travel time from origin node s to node j leaving the origin node at a given time interval t_o. To write the Bellman's optimality conditions for node j, we need to consider only the paths arriving at node $i \in B(j)$ at a time greater or equal to f_i. Minimum travel times can then be defined as solution of the following set of equations:

$$f_j = \begin{cases} \min\limits_{i \in B(j)} \min\limits_{t \geqslant f_i} \quad (t + d_{ij}(t)) & , j \neq o \\ 0 & , j = o \end{cases} \tag{2.2}$$

Proposition 1 *If the FIFO condition is satisfied, the above formulation of the fastest paths problem is equivalent to the following equations :*

$$f_j = \begin{cases} \min\limits_{i \in B(j)} \quad (f_i + d_{ij}(f_i)) & , j \neq o \\ 0 & , j = o \end{cases} \tag{2.3}$$

Proof: The equivalence holds because, $\min\limits_{t \geqslant f_i}(t + d_{ij}(t)) = f_i + d_{ij}(f_i)$, if the FIFO condition holds (see equation 2.1). □

The equivalent formulation in equation 2.3 is similar to static shortest paths optimality conditions with t replaced by f_i. Hence, it shows that all static shortest paths algorithms can be extended, without any extra execution time, to solve the fastest paths problem if the FIFO condition is satisfied. Chabini ([10]) also notes that only a static forward labeling process can be used. We summarize this result in the following proposition.

Proposition 2 *If the FIFO condition is satisfied, the formulation of the fastest paths problem in dynamic networks is equivalent to a static shortest paths problem. Hence, any forward labeling static shortest path algorithm can be used to solve the dynamic fastest paths problem. The dynamic fastest problem is solved in the same time complexity as the static shortest paths problem.*

□

Based on the above results, three different forward labeling algorithms/ implementations are designed: label setting algorithm with heaps (Dijkstra's algorithm [13], label setting with buckets (Dial's implementation [14]) and a label correcting algorithm with a dequeue ([26], [30]). As these algorithms have been discussed in detail in the literature, we do not discuss them in this thesis. However, an evaluation of these algorithms for dynamic networks similar to traffic networks is presented in Section 2.10. This evaluation is done to determine which of these algorithms is an efficient algorithm for transportation applications.

2.5 One to All Fastest Paths Problem for One Departure Time in non-FIFO Networks with Waiting Forbidden at All Nodes

This variant of the problem is the least studied in the literature. Orda and Rom [25] prove that in continuous dynamic non-FIFO networks, computation of simple or loop-less fastest paths is NP-Hard. This is mainly due to the fact that in non-FIFO

networks, fastest paths are not "concatenated", i.e., subpath from node s to node q of the fastest path from node s to node p via node q, may not be the fastest path from node s to node q.

The following approach to solve this problem in discrete networks was proposed by Chabini and Dean [6]. The fastest paths returned using this approach may contain loops. The main idea is to apply an increasing order of time labeling algorithm on the part of time-space expanded network, after departure time interval t.

2.5.1 Formulation

Let the origin node be s and the departure time interval be t_o. We define variables $w_i(t)$ for every node (i, t) in the time-space expansion representation as:

$$w_i(t) = \begin{cases} t - t_o & \texttt{if there is at least one path} \\ & \texttt{reaching node } i \texttt{ at time } t \\ \infty & \texttt{otherwise} \end{cases} \tag{2.4}$$

Let f_i denote the minimum travel time to reach node i from origin s departing at time interval t_o. These are given by :

$$f_i = \min_{t \geqslant t_o} w_i(t), \forall i \in N \tag{2.5}$$

For optimality, $w_i(t)$ should satisfy the following system of equations:

$$w_s(t_o) = 0, \tag{2.6}$$

$$w_j(t + d_{ij}(t)) = w_i(t) + d_{ij}(t), \qquad \forall\, j \in N, \forall\, i \in B(j), \forall\, t < M - 1 \tag{2.7}$$

$$w_j(M-1) = \min_{\substack{i \in B(j), \\ t + d_{ij}(t) \geqslant M-1, \\ t < M-1}} \begin{cases} w_i(t) + d_{ij}(t), \\ w_i(M-1) + d_{ij}(M-1) \end{cases} \qquad \forall j \in N \tag{2.8}$$

Proposition 3 *Labels $w_i(t)$, $\forall t < M-1$, $\forall i$ can be set in an increasing order of time intervals.*

Proof: Since all arc travel times are positive integers, labels corresponding to time intervals t are updated only by labels at time intervals earlier than t (see equation 2.6). This result implicitly reflects the acyclic property, along the time dimension, of the time-space expanded network. □

When the labels are set in the increasing order of time, at a particular time interval, some of the nodes may not have been reached ($w_i(t)$ is still infinity). These nodes will not be able to improve the label of any other node. Hence, this leads us to our next proposition.

Proposition 4 *At each time interval t, only those nodes i for which $w_i(t) = t - t_o$ need to be processed.*

□

Some nodes may not have feasible paths in the dynamic network (i.e., the node is never reached in the dynamic part of the network). Also, some other nodes can have paths consisting of both a dynamic part and a static part. When an increasing order of time algorithm is applied till the end of dynamic part, the fastest paths to these nodes are not computed. Hence, once we reach the time interval $M-1$, we use the following proposition to determine the labels of these nodes.

Proposition 5 *At time horizon $M-1$, any one-to-all static shortest paths algorithm can be used to compute fastest paths to those node whose fastest paths do not entirely belong to the dynamic part of the dynamic network. In the static shortest path computation, label of every node $i \in N$ should be initialized to $w_i(M-1)$.*

Proof: This can be easily proved using the equation 2.6. In the static part, label of a node j is minimum of length of paths coming from dynamic part and those in the static part. After processing the dynamic part of the dynamic network in the increasing order, the first term $(w_i(t) + d_{ij}(t), \quad t + d_{ij}(t) \geqslant M-1,\ t < M-1)$ in the minimum is determined. Hence, this can be viewed as a constant. The second part of

the equation is exactly similar to a static shortest path formulation. Hence, we can use any static shortest path algorithm to compute the labels for $t > M-1$. But, the labels computed for each node i should be compared against $w_i(M-1)$. Hence, we initialize the label of node i to $w_i(M-1)$, before we proceed with the static shortest path computation.

This result can also be illustrated by viewing the static shortest path computation as a reoptimization problem with certain initial estimates of the labels given by the dynamic network. Figure 2.2 illustrates this idea. In Figure 2.2, s denotes the source node and the dashed line from (s, t_0) to $(s, M-1)$ indicates the minimum path going from (s, t_o) to $(s, M-1)$ in the dynamic part of the network. The solid lines in the graph are the links in the static part of the network. The dashed lines are paths in the dynamic network with length of path to node i computed as $w_i(M-1)$ using an increasing order of time algorithm till time interval $M-1$. These can be considered as the artificial arcs added to the static part of the network. Hence, it is clear that the shortest path from node s can be computed by initializing the labels of all nodes to $w_i(M-1)$ and then, applying any one-to-all static shortest path algorithm. □

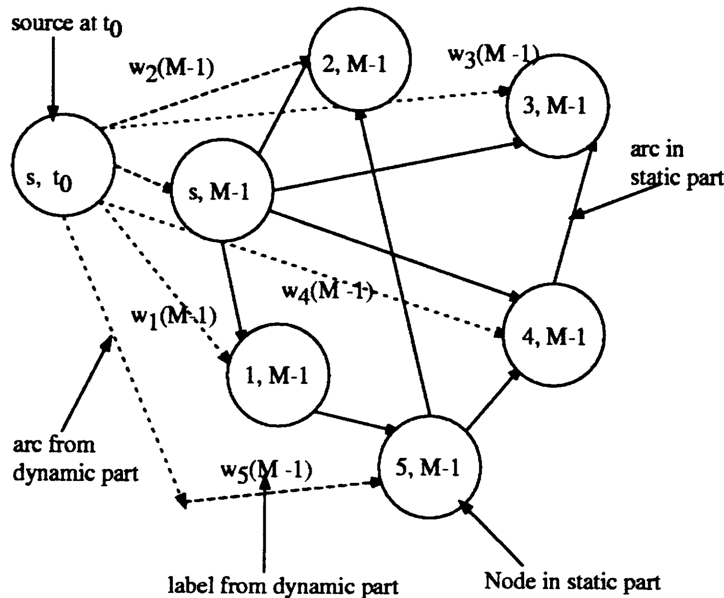


Figure 2.2: Illustration of forward labeling for non-FIFO networks

An algorithm, called IOT using the above approach is developed by Chabini and

Dean [6]. This algorithm is discussed in the next section.

2.5.2 Algorithm IOT

Chabini and Dean [6] design the following algorithm using propositions 3, 4,and 5 to compute dynamic fastest paths from node s to all nodes in the network for a given departure time interval t_o. Let f_i denote the minimum time in which i can be reached from s starting at time t_o and $w_i(t)$ be the label of the node (i, t) in the time space expanded network. Let $Q(t)$ contain the nodes (i, t) which have been reached and hence for these nodes $w_i(t) = t - t_o$.

Once n node labels have been set, the algorithm can stop, even if all time intervals are not processed. Hence, we denote by S, the set of nodes, whose labels are set. The algorithm stops when $|S| = n$.

If G is the network, C is the set of link costs, π is the set of labels of the nodes and s be the source node then, $SSP(G, C, \pi, s)$ is a static shortest path procedure, which returns a set of shortest path labels σ_i for every node $i \in N$.

Algorithm IOT

Step 0 (Initialization):

$$\begin{aligned} & f_i = \infty, \quad \forall i \in N - \{s\} \\ & w_i(t) = \infty, \quad \forall i \in N \text{ and } \quad t_o \leqslant t \leqslant M \\ & w_s(t_o) = 0 \\ & f_s = 0 \\ & Q(t_o) = \{s\} \\ & S = \{s\} \end{aligned} \tag{2.9}$$

Step 1 (Main Loop):

```
For  t = t_o ... M - 1
  For  all  i ∈ Q(t)
    if  (i ∉ S)  then  S = S ∪ {i}
    if  (|S| = n)  then RETURN.
    For  all  j ∈ A(i)
      τ = min(M - 1, t + d_ij(t))
      if  (w_j(τ) = ∞)  then
        Q(τ) = Q(τ) ∪ {j}
        w_j(τ) = τ - t_o
        f_j = min(f_j, τ)
```

(2.10)

Step 2 (Static Shortest Path Computation for $t \geqslant M-1$):

```
if(|S| < n)
  σ = SSP(G, d(M - 1), w_i(M - 1), s)
  for all i ∉ S
    f_i = σ_i
```

(2.11)

2.5.3 Complexity of algorithm IOT

Let L_o denote the longest fastest path from the origin node s to any node in the network. Then, the running time complexity of the algorithm IOT can be given by the following proposition.

Proposition 6 *Algorithm IOT has a running time complexity of :*

$$\begin{cases} O(nM + m * L_o) & if\ L_o < M \\ O(nM + mM + SSP) & if\ L_o \geqslant M, \end{cases} \tag{2.12}$$

where SSP is the time taken to compute static shortest paths.

Proof: The running time complexity can be easily obtained by counting the number of operations in the algorithm IOT. In the worst case (i.e., for departure time 0), initialization takes $O(nM)$ operations. Recall that the main loop of the algorithm is exited as soon as the fastest paths to all the nodes in the network are computed. Hence, if $(L_o < M)$, that is, all the nodes have fastest paths within the dynamic part itself, then, the running time complexity of the main loop is $O(m * L_o)$ and Step 2 (Static shortest path computation) need not be executed.

If the $(L_o \geqslant M)$, the running time of the main loop would be in the order of $O(mM)$ and as the fastest paths to some of the nodes have not been found yet, we need to compute one-to-all static shortest paths(O(SSP)). Hence, the order of algorithm IOT in this case is $O(nM + mM + SSP)$. □

The value of L_o depends on the maximum link travel time in the network, and also on the diameter of the network. Hence, the performance of algorithm IOT is better understood by an extensive experimental evaluation (see Section 2.10).

In the next section, we present certain results relating to dynamic fastest path problems when waiting at nodes is permitted.

2.6 One to All Fastest Paths for One Departure Time when Waiting at Nodes is Allowed

One of the earliest papers dealing with waiting at nodes has been published by Orda and Rom [25]. They study the following waiting policies for continuous networks:

- Unrestricted waiting (UW) in which unlimited waiting is allowed everywhere in the network.
- Forbidden waiting (FW) in which waiting is disallowed everywhere in the network (same as described in Sections 2.4 and 2.5).
- Source Waiting (SW) in which waiting is disallowed everywhere in the network except at the source node where unlimited waiting is permitted.

They prove that unrestricted waiting problem can be solved in the same time complexity as the static shortest path problem. They also prove that if the departure time from the source node is unrestricted, a shortest path with the same path travel time as that of unrestricted waiting time can be found.

The most celebrated result of the waiting at nodes is allowed variant is: *When unlimited waiting is permitted at all nodes, the fastest paths problem in a FIFO or a non-FIFO network has the same time complexity as that of static shortest paths computation.*

To prove this result, let us denote by $D_{ij}(t)$, the minimum travel time on arc (i,j) when one arrives at node i at time t. Hence, $D_{ij}(t) = \min_{s \geqslant t}(s - t + d_{ij}(s))$. That is, if we leave the node i at time interval s, the waiting time is $s - t$ and travel time on the link is $d_{ij}(s)$, at time interval s.

Proposition 7 *Functions $D_{ij}(t)$ satisfy FIFO condition.*

Proof : It can be seen from the definition of $D_{ij}(t)$ that, for a given link (i,j) and time interval t, $D_{ij}(t) \leqslant D_{ij}(t+1)+1$. If we add t to both the sides of this inequality, we obtain an inequality that satisfies the mathematical condition given for FIFO links (see equation 2.1). Hence, it is proved that functions $D_{ij}(t)$ satisfy the FIFO condition.

□

When waiting is allowed at nodes, the minimum travel times are given by the following functional form :

$$f_j = \begin{cases} \min\limits_{i \in B(j)} \min\limits_{t \geqslant f_i} \quad (t + D_{ij}(t)) & j \neq o \\ 0 & j = o \end{cases} \tag{2.13}$$

Proposition 8 *The waiting is allowed variant is a particular case of waiting is not allowed variant of the dynamic fastest paths problem. If unlimited waiting at nodes is permitted, results of propositions 1, 2 hold even for non-FIFO networks.*

Proof: Using the optimality conditions given in equation 2.13, we can deduce that waiting is allowed at all nodes variant of the fastest paths problem is equivalent to

waiting is not allowed variant with the $d_{ij}(t)$ in the latter replaced by $D_{ij}(t)$. As $D_{ij}(t)$ satisfy FIFO condition (see proposition 7), the waiting is allowed at all nodes variant is exactly similar to the waiting is not allowed at nodes policy in FIFO networks, hence, propositions 1, 2 hold even for non-FIFO networks. □

Any work discussing limited waiting at nodes has not been published yet. But, certain optimal algorithms for this problem have been proposed by Chabini and Dean [6].

As it is established that waiting is allowed policy is a special case of waiting is not allowed variant, we study only the latter variant from now on.

We conclude the above discussion of one-to-all fastest paths problems noting that the most difficult variant of this problem, in terms of worst time complexity, is finding loopless fastest paths when waiting is not allowed in non-FIFO networks.

2.7 All-to-One Fastest Paths for All Departure Times

This variant of the dynamic shortest paths problem is most relevant in context of traffic networks. The problem can be modeled using a backward star formulation.

2.7.1 Formulation

When waiting is not allowed , the minimum travel times can be defined by following functional form:

$$\pi_i(t) = \begin{cases} \min\limits_{j \in A(i)} \quad (d_{ij}(t) + \pi_j(t + d_{ij}(t)) & i \neq q \\ 0 & i = q \end{cases} \qquad (2.14)$$

where, $\pi_i(t)$ denotes the fastest travel time to destination q departing node i at time interval t. This optimality condition was first established by Cooke and Halsey [12]. They developed an algorithm using this optimality condition, with worst case running time complexity of $O(n^3M^2)$. Later, Ziliaskopoulos and Mahmassani

[30] extended the static label correcting algorithm to design a solution to this problem. The worst case running time complexity of their algorithm is $O(nmM^2)$.

Chabini [10] used the acyclic nature in the time dimension of the discrete dynamic network and designed an optimal algorithm for this problem. The algorithm computes labels in the decreasing order of time, and is called DOT. Algorithm DOT is proved to be optimal and offers many parallelization avenues. Hence, we use DOT to develop parallel implementations of all-to-one dynamic fastest paths problems. We present the ideas behind the design of this algorithm and algorithm in the rest of the section.

Proposition 9 *Labels $\pi_i(t)$ can be set in a decreasing order of departure time intervals.*

Proof: Since all arc travel times are positive integers, labels corresponding to time steps t never update labels corresponding to time steps greater than t (see equation 2.14). This result implicitly reflects the acyclic property, along the time dimension, of the time-space expansion of a discrete dynamic network. □

2.7.2 Algorithm DOT

Algorithm DOT was developed by Chabini [10] using the Proposition 9.

Algorithm DOT

Step 0 (Initialization):

$$\begin{aligned} &\pi_i(t) = \infty, \ \forall(i \neq q), \\ &\pi_q(t) = 0, \ \forall(t < M-1) \\ &\pi_i(M-1) = \mathtt{StaticShortest}(d_{ij}(M-1), q) \ \forall i \end{aligned} \tag{2.15}$$

Step 1 (Main Loop):

$$\begin{aligned} &\text{For } t = M-2 \text{ down to } 0 \text{ do} \\ &\quad \text{For } (i,j) \in A \text{ do} \\ &\quad\quad \pi_i(t) = \min(\pi_i(t), d_{ij}(t) + \pi_j(t + d_{ij}(t))) \end{aligned} \tag{2.16}$$

2.7.3 Complexity of algorithm DOT

Proposition 10 *Algorithm DOT solves for the all-to-one fastest paths problem, with running time in $\theta(SSP + nM + mM)$, where $O(SSP)$ is the worst time complexity of static shortest paths computation.*

Proof: The correctness of the algorithm follows directly from Proposition 9. The running time complexity can be calculated in a straight forward manner by counting the number of operations in the algorithm. The initialization step needs $\theta(nM)$ operations, the main loop requires $\theta(mM)$ operations and the worst time complexity of the static shortest path computation is $O(SSP)$. Hence, the total running time complexity is $\theta(SSP + nM + mM)$. □

Proposition 11 *The complexity of the all-to-one fastest paths problem for all departure times is $O(nM + mM + SSP)$. Hence, algorithm DOT is optimal (no algorithm with better running time complexity can be found).*

Proof : The problem has the complexity of $O(nM+mM+SSP)$ since every solution has to access all arc data (mM), initialize nM labels because fastest paths for all departure times are sought (nM), and compute all to one static shortest paths for departure time interval greater than or equal to $M-1$ (SSP). In proposition 10 we proved that worst time complexity of algorithm DOT is $\theta(nM + mM + SSP)$. Hence, algorithm DOT is optimal. □

In Section 2.10, we discuss the performance of algorithm DOT for different test networks.

In the next two sections, we discuss the dynamic minimum cost path problems. We will see that some of the results obtained for dynamic fastest path problems can be extended to these problems as well.

2.8 One-to-All Minimum Cost Paths for One Departure Time

The one-to-all minimum-cost path problems are more complex than the fastest path problems. No obvious condition resembling FIFO condition in the fastest path problems can be identified for these problems, because, time and cost are two different dimensions. Hence, a variety of combinations of conditions on link costs and link travel times is possible.

Moreover, The waiting at nodes is allowed variant of this problem is more complicated than that of the fastest path problems because a new function to measure the waiting cost as a function of amount of time waited needs to be defined. This function can be any general function. Note that in the fastest paths problems, the cost incurred due to waiting is the amount of time we wait at a node.

The optimality conditions for the one-to-all minimum cost path problems when waiting is not allowed are given in the next section. Algorithms developed using these conditions may be used to solve the fastest path problems, but they will be inefficient compared to the algorithms developed for fastest path problems. Because, algorithms developed for fastest path problems are more specialized and take into account certain properties of those problems.

2.8.1 Formulation

Let the origin node be s and departure time be t_o. Let us denote the minimum cost to reach a node i in the network by C_i and let $w_i(t)$ denote the minimum cost of reaching the node (i,t) in the time space expanded network. If there is no path reaching node (i,t), the value of $w_i(t) = \infty$. Therefore, $C_i(t) = \min\limits_{t \geqslant t_o} w_i(t)$. For optimal solutions, $w_i(t)$ should satisfy the following system of equations:

$$w_s(t_o) = 0, \tag{2.17}$$

$$w_j(\tau) = \min_{i \in B(j)} (w_i(t) + c_{ij}(t)) \qquad \forall\, j \in N,\ \forall\, t < M-1, \quad \tau = t + d_{ij}(t) \tag{2.18}$$

$$w_j(M-1) = \min_{\substack{i \in B(j),\\ t + d_{ij}(t) \geqslant M-1,\\ t < M-1}} \begin{cases} w_i(t) + c_{ij}(t), \\ w_i(M-1) + c_{ij}(M-1) \end{cases} \qquad \forall j \in N \tag{2.19}$$

It can be seen that the above conditions are similar to those in equation 2.6. Hence, the propositions 3, 4, 5 can be extended to minimum cost path problems. We present below the extended propositions.

Proposition 12 *Labels $w_i(t)$, $\forall t < M-1$, $\forall i$ can be set in an increasing order of time intervals.*

□

Proposition 13 *At each time interval t, only those nodes i for which $w_i(t) \neq \infty$ need to be processed.*

□

Proposition 14 *At time horizon $M-1$, a one-to-all static shortest paths algorithm should be used to compute shortest paths in the static part of the network. The minimum cost path of each node is the minimum of the minimum cost path in the dynamic network and that in the static network.*

□

An algorithm `IOT-MinCost` using propositions 12, 13 and 14 was developed by Chabini and Dean [6]. We discuss this algorithm in the next section.

2.8.2 Algorithm IOT-MinCost

We extend the notation used for algorithm IOT to algorithm IOT-MinCost. Again, let us denote by $Q(t)$ the bucket at time interval t. It contains the nodes that have been reached at time interval t, and hence for these nodes, $w_i(t) \neq \infty$. We also assume that there exists a static shortest paths procedure $SSP(G, C, \pi, s)$ same as the one assumed for fastest paths problem.

Note that in the fastest paths problem, if $t < \tau$, then, $w_i(t) \leqslant w_i(\tau)$. Hence, we used this condition to quit the IOT algorithm once n node labels are set. In the minimum cost paths problem, $w_i(t)$ are not functions of t. Hence, in these problems, one has to process all the time intervals and also compute the static shortest paths for any network.

Algorithm IOT-MinCost

Step 0 (Initialization):

$$
\begin{aligned}
&\mathtt{C_i} = \infty, \quad \forall \mathtt{i} \in \mathtt{N} - \{\mathtt{s}\} \\
&\mathtt{w_i(t)} = \infty, \quad \forall \mathtt{i} \in \mathtt{N} \text{ and } \quad \mathtt{t_o} \leqslant \mathtt{t} \leqslant \mathtt{M} \\
&\mathtt{w_s(t_o)} = 0 \\
&\mathtt{C_s} = 0 \\
&\mathtt{Q(t_o)} = \{\mathtt{s}\}
\end{aligned}
\tag{2.20}
$$

Step 1 (Main Loop):

$$
\begin{aligned}
&\mathtt{For} \quad \mathtt{t} = \mathtt{t_o} \ldots \mathtt{M} - 1 \\
&\quad \mathtt{For} \quad \mathtt{all} \quad \mathtt{i} \in \mathtt{Q(t)} \\
&\quad\quad \mathtt{For} \quad \mathtt{all} \quad \mathtt{j} \in \mathtt{A(i)} \\
&\quad\quad\quad \tau = \mathtt{min(M-1,\ t + d_{ij}(t))} \\
&\quad\quad\quad \mathtt{if} \quad (\mathtt{w_j}(\tau) = \infty) \quad \mathtt{then} \quad \mathtt{Q}(\tau) = \mathtt{Q}(\tau) \cup \{\mathtt{j}\} \\
&\quad\quad\quad \mathtt{w_j}(\tau) = \mathtt{min(w_j(\tau),\ t + c_{ij}(t)} \\
&\quad\quad\quad \mathtt{C_j} = \mathtt{min(C_j,\ w_j(\tau))}
\end{aligned}
\tag{2.21}
$$

Step 2 (Static Shortest Path Computation for $t \geqslant M-1$):

$$\begin{aligned} &\sigma = \mathtt{SSP}(\mathtt{G}, \mathtt{c}(\mathtt{M}-1), \mathtt{w_i}(\mathtt{M}-1), \mathtt{s}) \\ &\mathtt{for\ \ all\ \ i} \in \mathtt{N} \\ &\quad \mathtt{C_i} = \mathtt{min}(\mathtt{C_i},\ \sigma_\mathtt{i}) \end{aligned} \tag{2.22}$$

2.8.3 Complexity of algorithm IOT-MinCost

We have seen that algorithm IOT-MinCost is similar to algorithm IOT except that we do not have any dependence on the factor L_o, where L_o is the longest minimum-cost path to a node in the network. The complexity of this algorithm is given by the following proposition.

Proposition 15 *Algorithm IOT-MinCost has a running time complexity of* $O(nM + mM + SSP)$*, where* SSP *is the time taken to compute the static shortest paths.*

Proof: The running time complexity can be easily computed by counting the number of operations in the algorithm. In the worst case, we need to initialize $O(nM)$ labels. The main loop needs to process $O(mM)$ links in the worst case. And, finally for time interval $t = M - 1$, we need to compute the static shortest paths. Hence, the order of the algorithm IOT-MinCost is $O(nM + mM + SSP)$. □

The performance of this algorithm will be demonstrated in the experimental evaluation section (Section 2.10).

In the next section, we discuss the all-to-one minimum cost paths problem, for all departure time intervals. The extension of algorithm DOT to solve this problem has been proposed by Chabini [10]. We call this extension algorithm DOT-MinCost to distinguish it from algorithm DOT.

2.9 All-to-One Minimum Cost Paths Problem for all Departure Times Problem

In this section, we present a formulation of the all-to-one minimum cost paths problem for all departure time intervals. This formulation will be used to extend the results obtained in Section 2.7 for all-to-one fastest paths problem to this problem.

2.9.1 Formulation

Let $C_i(t)$ denote the minimum cost to reach the destination q from node i departing at time interval t. Minimum costs are then defined by the following functional form:

$$C_i(t) = \begin{cases} \min\limits_{j \in A(i)} \quad (c_{ij}(t) + C_j(t + d_{ij}(t)) & i \neq q \\ \qquad 0 & i = q \end{cases} \tag{2.23}$$

Again, as $d_{ij}(t)$ are positive integers, we can see that proposition 9 can be extended to the minimum cost paths problem. Note that to extend this result, $c_{ij}(t)$ can be any real value. Hence, we formalize this result using the following proposition.

Proposition 16 *Labels $C_i(t)$ can be set in a decreasing order of departure time intervals.*

□

2.9.2 Algorithm DOT-MinCost

As the network is static after time interval $M - 1$, we use a static shortest path algorithm to set the labels $C_i(M-1)$. The choice of the static shortest paths algorithm should be made depending on the costs $c_{ij}(t)$. Some static shortest paths procedures require that there is no negative cycle in the network. Otherwise, this negative cycle can be circled infinite number of times leading to an infinite decrease in the labels. Hence, depending on the values of $c_{ij}(t)$, we have to choose an appropriate static shortest path algorithms.

Algorithm `DOT-MinCost`

Step 0 (Initialization)

$$\begin{aligned} &\mathtt{C_i(t)} = \infty,\ \forall(\mathtt{i} \neq \mathtt{q}), \\ &\mathtt{C_q(t)} = 0,\ \forall(\mathtt{t} < \mathtt{M}-1) \\ &\mathtt{C_i(M-1)} = \mathtt{StaticShortest}(\mathtt{c_{ij}(M-1), q})\ \forall \mathtt{i} \end{aligned} \tag{2.24}$$

Step 1 (Main Loop)

$$\begin{aligned} &\textbf{For } \mathtt{t} = \mathtt{M}-2 \textbf{ down to } 0 \textbf{ do} \\ &\quad \textbf{For } (\mathtt{i,j}) \in \mathtt{A} \textbf{ do} \\ &\qquad \mathtt{C_i(t)} = \min(\mathtt{C_i(t)}, \mathtt{c_{ij}(t)} + \mathtt{C_j(t + d_{ij}(t))}) \end{aligned} \tag{2.25}$$

We can extend the propositions 10 and 11 to the minimum cost paths problem as well.

2.9.3 Complexity of algorithm `DOT-MinCost`

Proposition 17 *Algorithm* `DOT-MinCost` *solves for all-to-one minimum cost paths, for all departure times in* $\theta(nM + mM + SSP)$.

Proof: This result can be proved by counting the number of operations in the algorithm `DOT-MinCost`. Initializing the labels of all nodes for all time intervals requires $\theta(nM)$ operations. Then the main loop requires $\theta(mM)$ operations and one needs to compute static shortest paths for the time interval $M-1$. Hence, the order of the algorithm `DOT-MinCost` is $\theta(nM + mM + SSP)$. □

Proposition 18 *The complexity of the all-to-one minimum cost path problem, for all departure times is* $O(nM + mM + SSP)$*. Hence, algorithm* `DOT-MinCost` *is optimal.*

Proof: The problem has the complexity $O(nM + mM + SSP)$ because to compute minimum cost paths, we need to access all the arc data (mM) and initialize labels for all nodes for all departure time intervals (nM), as the minimum cost paths from all nodes for all departure time intervals are desired. We have to compute static

shortest paths for the time interval $M - 1$ (O(SSP)). In proposition 17, we have proved that the order of algorithm `DOT-MinCost` is $\theta(nM + mM + SSP)$. Hence, algorithm `DOT-MinCost` is optimal. □

All the above mentioned algorithms were coded in C++. An extensive evaluation of these implementations was done to gauge the performance of different algorithms. We discuss this evaluation in the next section.

2.10 Experimental Evaluation

In this section, we report on the experimental evaluation of the following solution algorithms:

- `ls-heap` : Label setting algorithm using heaps to compute one to all fastest paths problem for FIFO networks.
- `lc-dequeue`: Label correcting algorithm using dequeue for one to all fastest paths for FIFO networks.
- `dial-buckets`: Dial's implementation of label setting algorithm using buckets to compute one to all fastest paths for FIFO networks.
- `IOT` : Increasing order of time algorithm to compute one to all fastest paths for one departure time in non FIFO networks.
- `DOT` : Decreasing order of time algorithm to compute all to one fastest paths for all departure times.
- `IOT-MinCost` : Increasing order of time algorithm to compute one to all dynamic minimum cost paths for a given departure time.
- `DOT-MinCost` : Decreasing order of time algorithm to compute all to one minimum cost paths for all departure times.

The network instances used for evaluation in this section are generated using a discrete dynamic network generator developed to test computer codes on networks

with different topologies, number of cycles, densities and link travel times. Running times are wall clock times obtained on a SUN SPARC workstation.

Let the maximum link cost be denoted by W and the maximum link travel time be C. Unless otherwise mentioned, in the following tests, $W = 10$ and $C = 3$. The evaluations done are presented below.

- **Comparison of the performance of the three algorithms for FIFO networks :** In Figure 2.3, we show the performance of algorithms `ls-heap`, `lc-dequeue` and `dial-buckets` for networks similar to transportation networks (i.e., with degree of 3). The x-axis in the Figure 2.3 shows the number of nodes. The network has n nodes, $3 * n$ links and 100 time intervals. We can see that for these networks, the algorithm `lc-dequeue` performs better than the other two algorithms. Considerable evaluation of these algorithms has appeared in the literature. Hence, we do not do a detailed evaluation of these algorithms.

- **Performance evaluation of algorithm `IOT` with respect to the different network parameters :** Algorithm `IOT` was designed using certain specific properties of the dynamic networks. We have established in Proposition 6 that the worst time complexity of algorithm `IOT` is a function of the longest fastest path in the network, the number of time intervals and the number of links. The longest minimum path is the network is, in turn, a function of the diameter of the network and the maximum link travel time.

  In this evaluation, we establish the performance of algorithm `IOT` with respect to the following network parameters: maximum link travel time, number of nodes, number of arcs and number of time intervals in the network. In all these evaluation, we show the variation of running time with respect to each network parameter and also the variation of $Min(L_o, M)$ with respect to each network parameter.

  - *Maximum link travel time:* In Figure 2.4, we show the performance of algorithm `IOT` with respect to the maximum link travel time (C) for a

network with 3000 nodes, 9000 links and 100 time intervals. We also show the variation of $Min(L_o, M)$ with respect to the network parameter C.

In Figure 2.4, we notice the following: for smaller values of C, the running time increases and for very high values of C, the running time decreases. With increasing C, L_o increases. For smaller values of C, L_o is less than M. Hence, the running time increases as L_o increases. This explains the performance of algorithm IOT for smaller values of C. When C is very high, there are lesser number of nodes in each bucket. Hence, the time taken by the main loop (See set of equations 2.10) is less. The time taken by the static shortest path algorithm very less when compared to the time taken by the main loop in the algorithm IOT. Hence, the running time of algorithm IOT decreases for high values of C.

- *Number of nodes:* Figure 2.5 shows the performance of algorithm IOT with respect to the number of nodes in the network. In the same figure, we also show the variation of $Min(L_o, M)$ with number of nodes.

  In Figure 2.5, we observe that the running time of algorithm IOT initially increases with the number of nodes and then decreases as the network tends to be a tree. The running time increases initially because the value of L_o increases (see Figure 2.5). For a network with 9000 nodes and 9000 links, the number of nodes to be processed within the main loop of the algorithm IOT is less and as L_o is greater than M, the static shortest path computation is necessary. The static shortest path computation requires very less amount of time when a network is a tree as there are lesser number of loops in the network. Hence, we observe a decrease in the running time of algorithm IOT when the number of nodes is increased to 9000.

- *Number of arcs:* Figure 2.6 shows the performace of algorithm IOT with respect to the number of links in the network. In the same figure, we show the variation of $Min(L_o, M)$ with number of links.

  In Figure 2.6, we notice that the running time of algorithm IOT increases

initially with the number of links and then decreases. We also notice that L_o decreases with the number of links in the network. We have established in Proposition 6 that the running time of algorithm IOT is proportional to $m * L_o$ when $L_o < M$. We have observed in Figure 2.6 that as m increases, L_o decreases. For lesser m, the decrease of L_o is not high. The running time increases (the increase in m outweighs the decrease in L_o). But, for higher values of m, we see that L_o decreases a lot. Thus, the running time of algorithm *IOT* decreases (the decrease in L_o outweighs the increase in m).

– *Number of time intervals:* Figure 2.7 shows the performance of algorithm IOT with respect to the number of time intervals. In the same figure, we show the variation of $Min(L_o, M)$ with number of time intervals.

  The longest minimum path in the network should not depend on the number of time intervals. That is why, we see only a slight increase in the value L_o with the increase in the number of time intervals. This may be due to the random nature of these networks. This increase in L_o leads to an increase in the running time of the algorithm IOT.

- **Performance evaluation of algorithm `IOT-MinCost` with respect to the different network parameters :** Algorithm `IOT-MinCost` was developed to compute one-to-all minimum cost paths for a given departure time. For any network, all the three part of the algorithm `IOT-MinCost` should be run (see Section 2.8.2. The factors which would affect the run time highly are: total number of nodes in the buckets, the number of links per each node to be processed and the time required for the static shortest path computation. These factors would depend on the following network parameters: number of nodes, number of links, number of time intervals, maximum link travel time and maximum link cost. Hence, we evaluate the performance of algorithm `IOT-MinCost` with respect to these parameters.

  – *Number of nodes:* Figure 2.8 shows the performance of algorithm IOT-

MinCost with respect to the number of nodes. In this figure, we observe that the run time of algorithm IOT-MinCost decreases with the number of nodes. As the number of nodes increases, number of links to be processed per each node decreases. This leads to lesser number of nodes in the buckets. Moreover, as the network tends to be a tree, the time required for static shortest path computation decreases. Hence, the running time of algorithm IOT-MinCost decreases with increase in number of nodes.

- *Number of arcs:* Figure 2.9 shows the performance of algorithm IOT-MinCost with respect to the number of links. In this figure, we that the running time of algorithm IOT-MinCost increases with the number of links. This case exactly the reverse of the earlier evaluation (with respect to the number of nodes). As the number of links increase, the number of links per each node increases. This leads to an increase in the number of nodes in the buckets. Also, the static shortest path computation time increases with increase in the number of links. Hence, the running time of algorithm IOT-MinCost increases with increasing number of links.

- *Number of time intervals:* Figure 2.10 shows the performance of algorithm IOT-MinCost with the respect to the number of time intervals. In this figure, we see that the running time of algorithm IOT-MinCost increases with the increase in the number of time intervals. The running time of the main loop in algorithm IOT-MinCost is in the order of mM. Thus, as the number of time intervals increases, the running time of algorithm IOT-MinCost increases.

- *Maximum link travel time:* Figure 2.11 shows the performance of algorithm IOT-MinCost with respect to the maximum link travel time (C). In this figure, we see that the running time of algorithm IOT-MinCost decreases as C increases. As C increases, the number of nodes in the buckets decreases (since, lesser number of nodes are reached). Thus, the computation time of the main loop in algorithm IOT-MinCost decreases leading to a decrease

in the running time of algorithm `IOT-MinCost`.

- *Maximum link cost:* Figure 2.12 shows the performance of algorithm `IOT-MinCost` with respect to the maximum link cost (W). W does not directly impact any of the major factors which affect the running time of the algorithm. Hence, although W effects the running time of the algorithm `IOT-MinCost`, the variation of the running time with respect to W can not be determined. Hence, in figure 2.12, we do not observe any specific pattern of variation of the performance with W.

- **Performance evaluation of algorithm `DOT` with respect to the different network parameters :** We have noted that the order of algorithm `DOT` is $\theta(nM + mM + SSP)$. To confirm this theoretical analysis, performance of algorithm `DOT` was evaluated with respect to the following network parameters: number of nodes, number of links and number of time intervals. These results confirm the theoretical analysis.

  - *Number of nodes:* Figure 2.13 shows the performace of algorithm `DOT` with the respect to number of nodes. In this figure , we observe that the running time of algorithm `DOT` is indeed linearly proportional to the number of nodes.
  - *Number of arcs:* Figure 2.14 shows the performance of algorithm `DOT` with respect to number of links. In this figure, we notice that the running time is linearly proportional to the number of links.
  - *Number of time intervals:* Figure 2.15 shows the performance of algorithm `DOT` with respect to number of time intervals. In this figure, we see that the running time of algorithm `DOT` is linearly proportional to the number of time intervals. Note that all the other algorithms proposed to calculate the all-to-one dynamic shortest paths are proportional to M^2 in the worst case.

- **Performance evaluation of algorithm `DOT-MinCost` with respect to the**

different network parameters : Algorithm `DOT-MinCost` is not very different from the algorithm `DOT`. We evaluate the performance of this algorithm with respect to the following parameters: number of nodes, number of links and number of time intervals. We have noted in Proposition 17 that the worst time complexity of algorithm `DOT-MinCost` is $\theta(nM + mM + SSP)$. We confirm this theoretical analysis from the following evaluations:

- *Number of nodes:* Figure 2.16 shows the running time of algorithm `DOT-MinCost` with respect to number of nodes. In this figure, we observe that running time of this algorithm is linearly proportional to number of nodes.
- *Number of arcs:* Figure 2.17 shows the running time of algorithm `DOT-MinCost` with respect to number of links. In this figure, we notice that the running time of algorithm `DOT-MinCost` is linearly proportional to the number of links.
- *Number of time intervals:* Figure 2.18 shows the performance of algorithm `DOT-MinCost` with respect to number of time intervals. In this figure, we see that the running time of the algorithm `DOT-MinCost` is linearly proportional to the number of time intervals.

- **Comparison of algorithm `IOT` and algorithm `DOT`:** We know that algorithms `IOT` and `DOT` solve two different basic dynamic fastest paths problems. But, given a specific dynamic fastest problems (For example, to calculate the many-to-many dynamic fastest paths for all departure time intervals in a network), which is a better algorithm to use? The following evaluation partly answers this question. We compare the running times of both algorithms to compute same amount of information. That is, we calculate the one-to-all dynamic fastest paths for all departure time intervals using `IOT` and compute the all-to-one dynamic fastest paths for all departure time intervals using algorithm `DOT`.

Figure 2.19 shows these results for a network with 3000 nodes, 9000 arcs and 100 time intervals. This figure shows running times for this network for different

maximum travel times (C), as we know that algorithm IOT is very sensitive to the parameter C. Of course, algorithm DOT does not depend on C. It can be seen from Figure 2.19 that algorithm DOT is approximately 10 times faster than algorithm IOT.

- **Comparison of algorithm IOT-MinCost and algorithm DOT-MinCost:** (Table 2.20) We do a similar comparison as was done for the minimum cost paths problem. Algorithm IOT-MinCost would be more time-consuming than algorithm IOT. Algorithm DOT-MinCost requires approximately same time as taken by algorithm DOT. Hence, in Figure 2.20 we see that algorithm DOT-MinCost is approximately 30 times faster than algorithm IOT-MinCost.

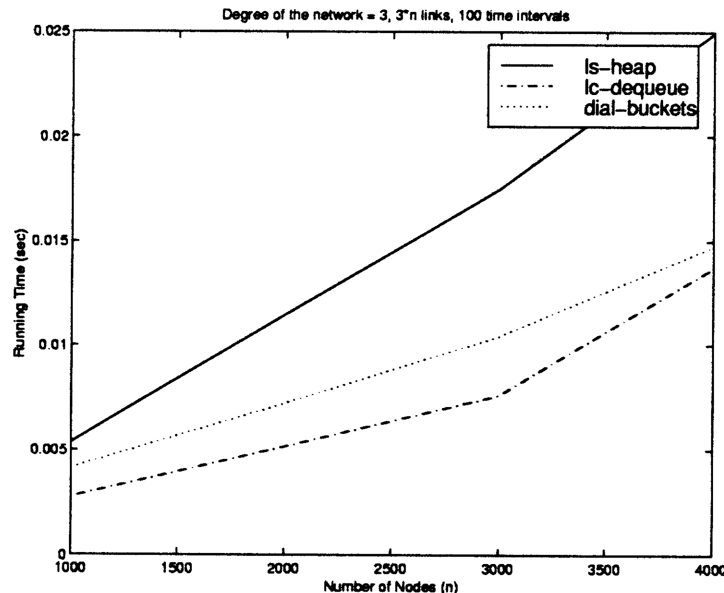


Figure 2.3: Algorithms for FIFO networks

2.11 Summary

In this chapter, we first, presented a classification of dynamic shortest path problems. We then, described a way of representing dynamic network using the time space expansion. This representation was useful in designing some efficient algorithms for

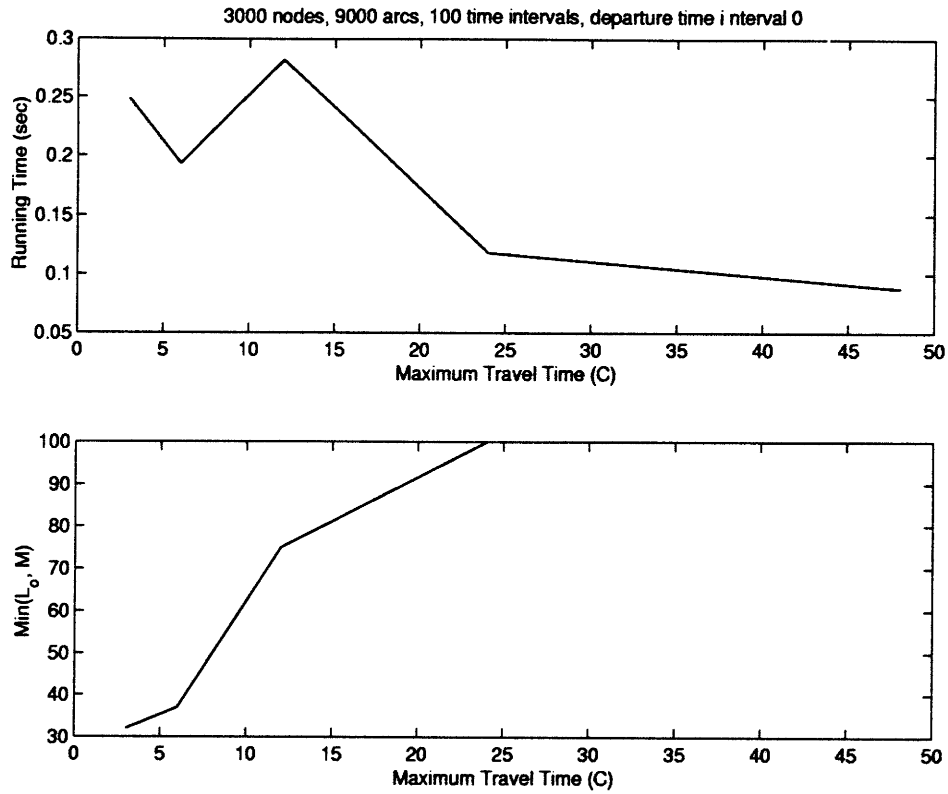


Figure 2.4: Algorithm IOT: varying maximum link travel time

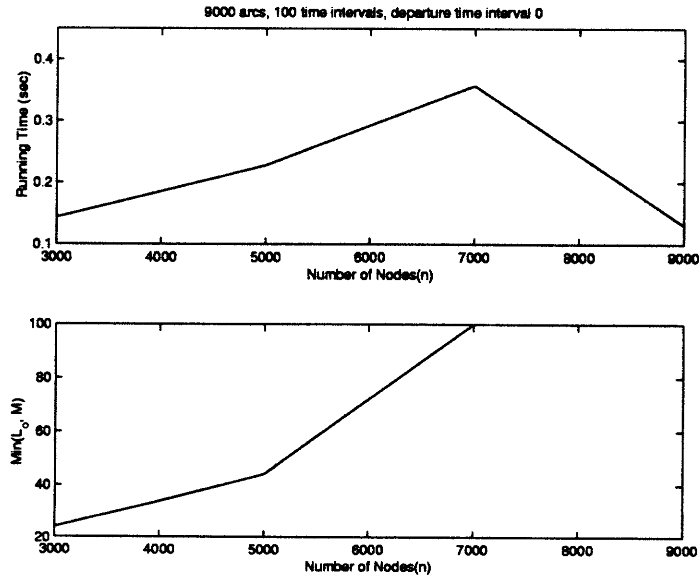


Figure 2.5: Algorithm IOT: varying number of nodes

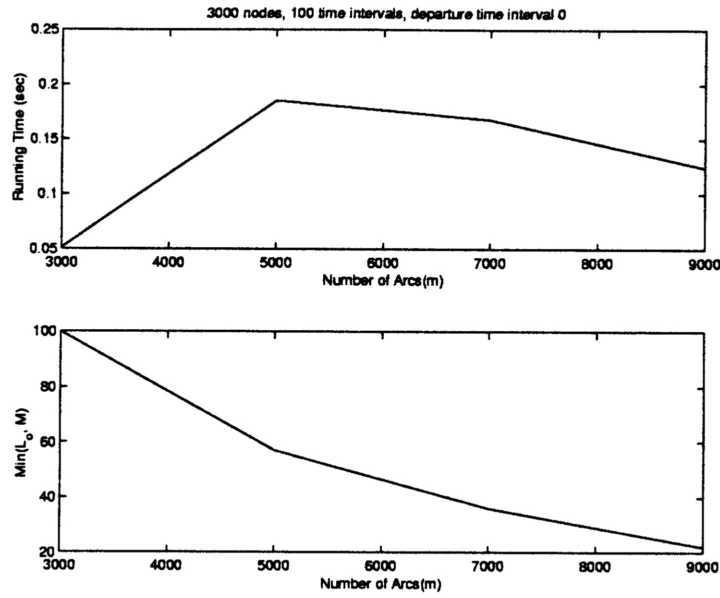


Figure 2.6: Algorithm IOT: varying number of links

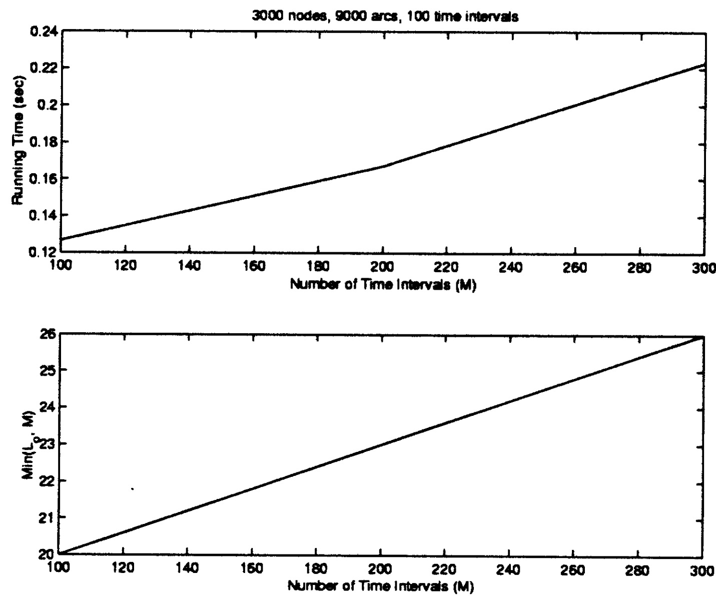


Figure 2.7: Algorithm IOT: varying number of time intervals

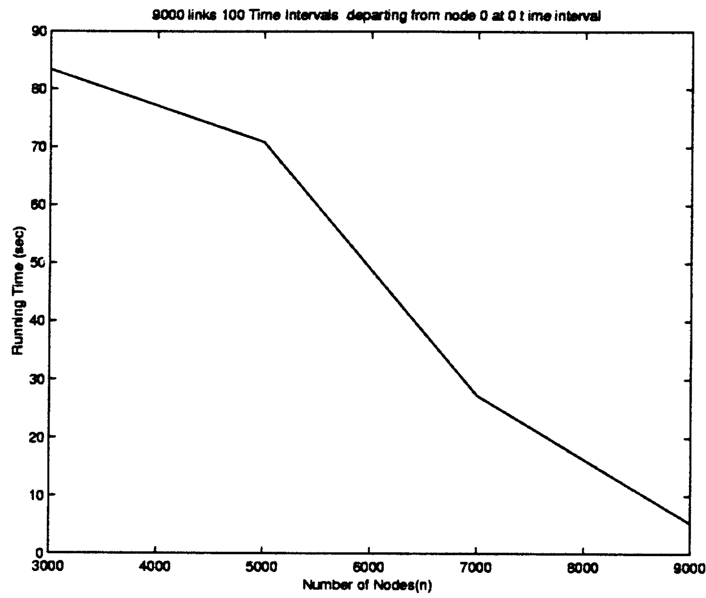


Figure 2.8: Algorithm IOT-MinCost: varying number of nodes

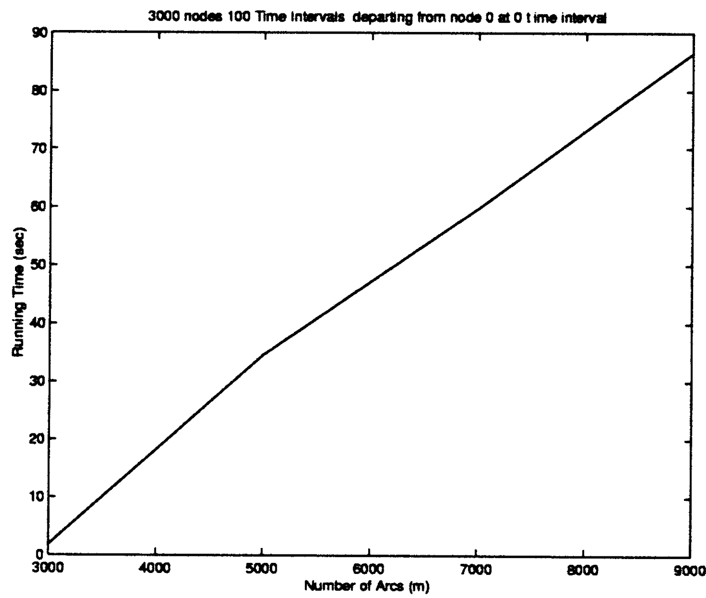


Figure 2.9: Algorithm IOT-MinCost: varying number of links

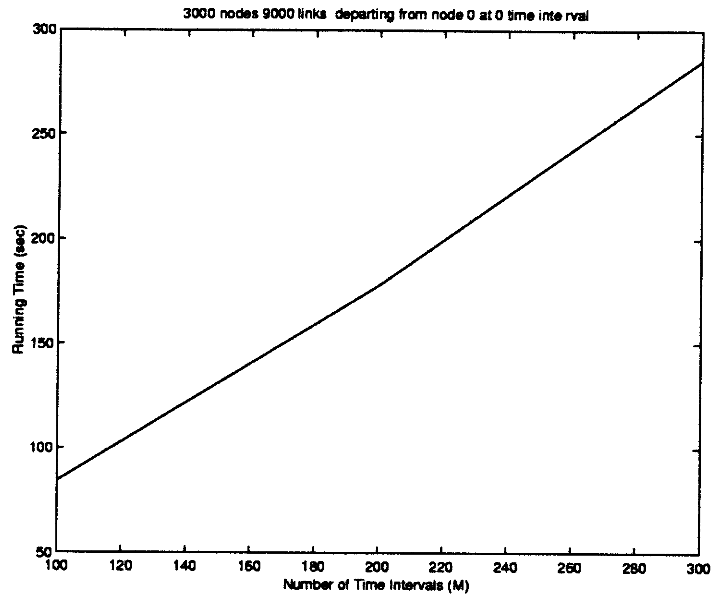


Figure 2.10: Algorithm IOT-MinCost: varying number of time intervals

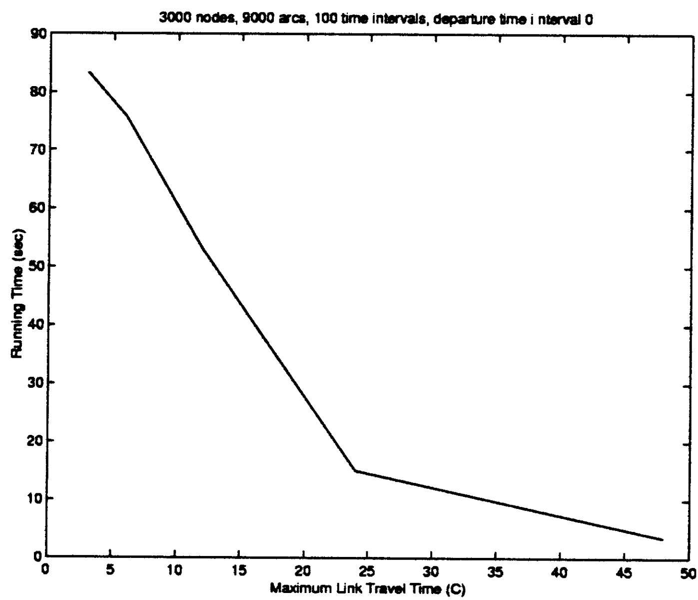


Figure 2.11: Algorithm IOT-MinCost: varying maximum link travel time

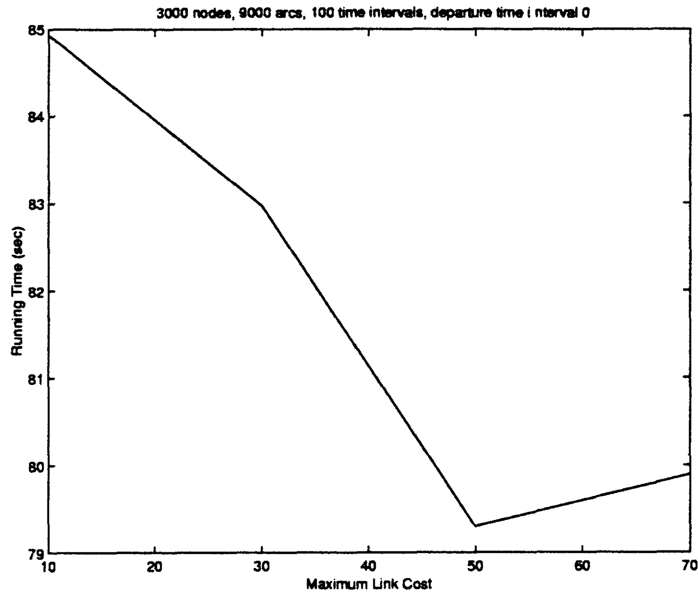


Figure 2.12: Algorithm `IOT-MinCost`: varying maximum link cost

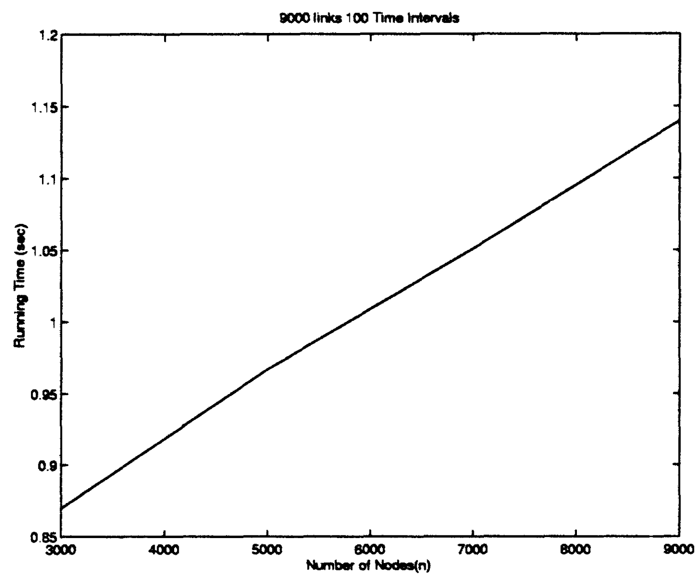


Figure 2.13: Algorithm `DOT`: varying number of nodes)

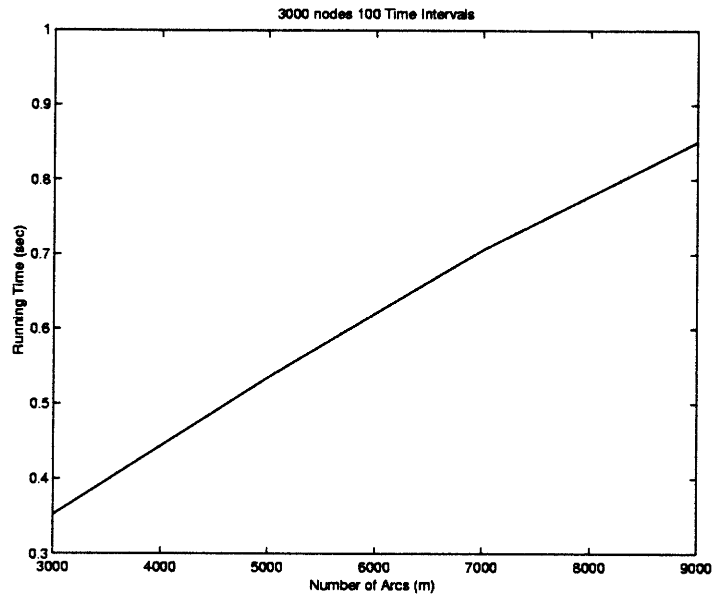


Figure 2.14: Algorithm DOT: varying number of links

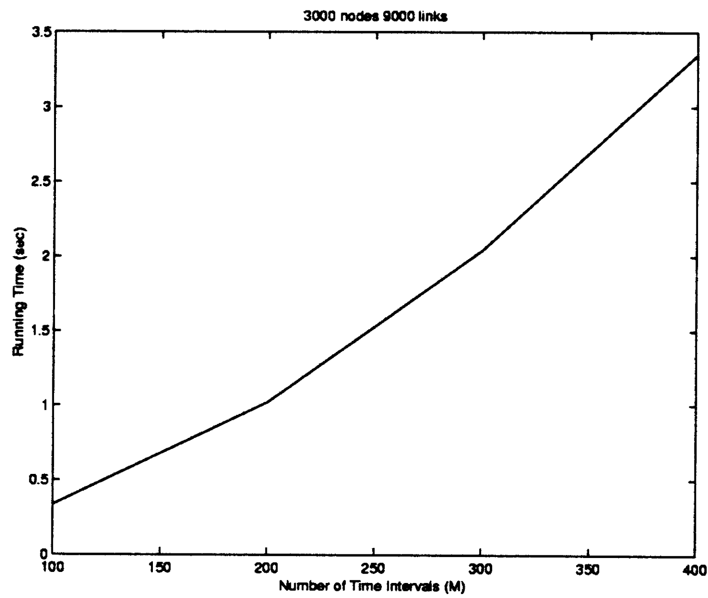


Figure 2.15: Algorithm DOT: varying number of time intervals

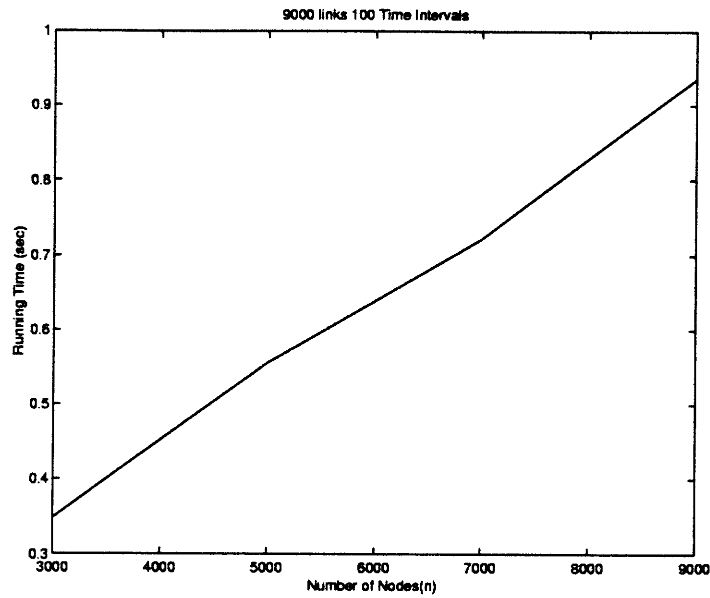


Figure 2.16: Algorithm DOT-MinCost: varying number of nodes

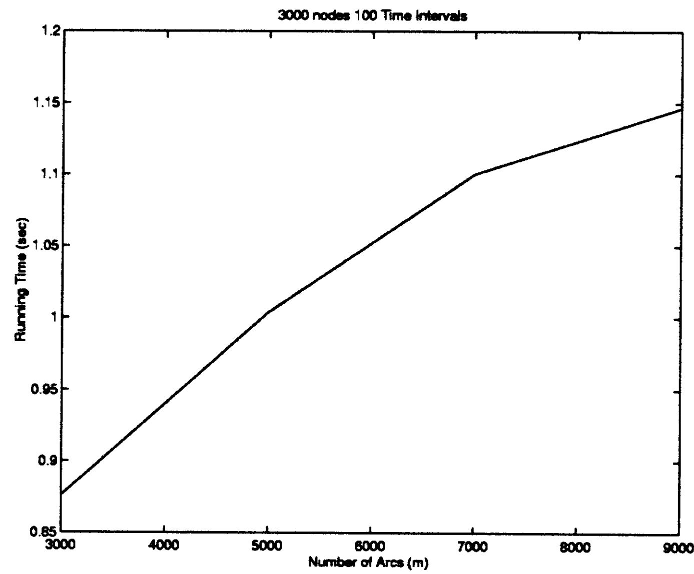


Figure 2.17: Algorithm DOT-MinCost: varying number of links

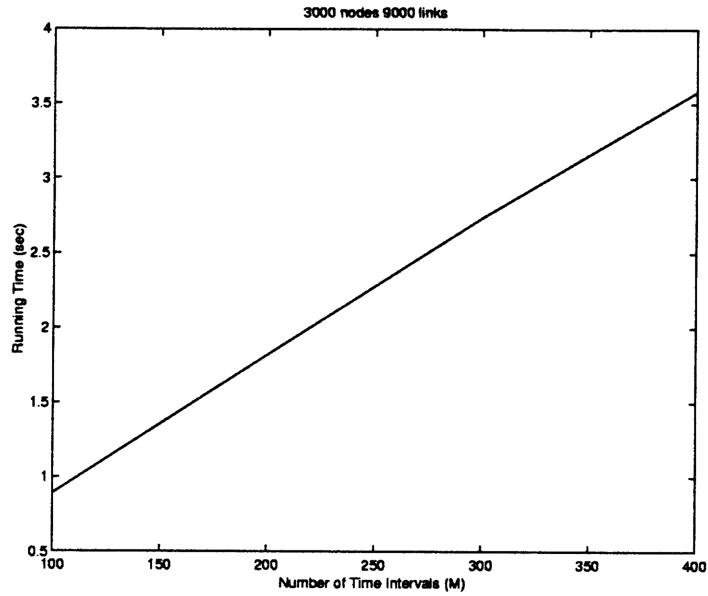


Figure 2.18: Algorithm DOT-MinCost: varying number of time intervals

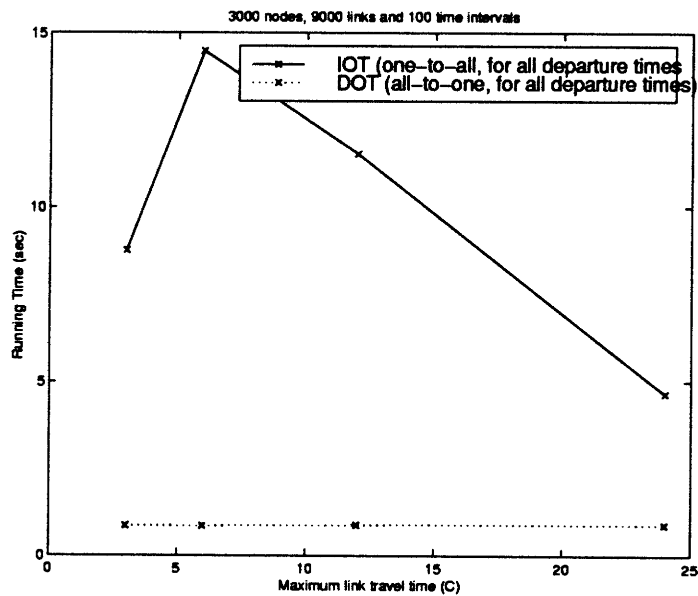


Figure 2.19: Comparison of algorithm IOT and algorithm DOT

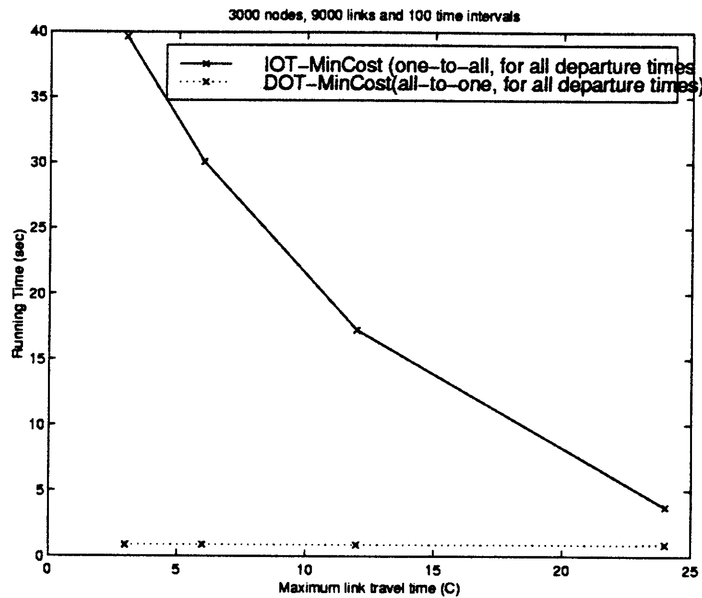


Figure 2.20: Comparison of algorithm `IOT-MinCost` and algorithm `DOT-MinCost`

certain variants of dynamic shortest path problems. Then, we presented the formulations, algorithms and computer implementations for the following types of dynamic shortest paths problems:

- One-to-all dynamic fastest paths for one departure time for FIFO networks when waiting is forbidden at all nodes.
- One-to-all dynamic fastest paths for one departure time for non-FIFO networks when waiting is forbidden at all nodes.
- One-to-all dynamic fastest paths when waiting is allowed at all nodes.
- All-to-one dynamic fastest paths for all departure times
- One-to-all dynamic minimum cost paths for one departure time
- All-to-one dynamic minimum cost paths for all departure times.

We have shown that static shortest path algorithms can be extended to solve the one-to-all dynamic fastest problems in FIFO networks.

We have shown that an increasing order of time algorithm can be used to solve one-to-all dynamic fastest paths in non-FIFO networks. Algorithm IOT was developed to solve the one-to-all fastest paths problem in non-FIFO networks using this result. An extension of algorithm IOT called IOT-MinCost was developed to solve the one-to-all dynamic minimum cost path problems.

We have established that decreasing order of time algorithms can be used to solve all-to-one dynamic fastest paths and all-to-one minimum cost paths problems. Using these results, algorithms DOT and DOT-MinCost were developed to solve the all-to-one dynamic fastest paths and all-to-one minimum cost paths problems respectively. We have proved that algorithms DOT and DOT-MinCost are optimal, that is, no other algorithm with a better running time complexity can be found.

An extensive experimental evaluation of all these algorithms was done. The conclusions of this evaluation can be summarized as:

- Among the algorithms tested, label correcting algorithm with dequeue data structure to hold the candidate list is the best algorithm to use for FIFO networks. Tests were carried out on traffic-like networks.

- For traffic networks, that require computation of fastest paths from almost all nodes in the network to few destinations for all departure times, DOT leads to best results.

- For a random network with 3000 nodes, 9000 links and 100 time intervals and maximum link travel time as 3, 100 iterations of IOT are 10 times slower than DOT. (see Figure 2.19).

- Although, algorithm DOT is proved to be an optimal algorithm, it still does not solve realistic dynamic fastest path problems faster than real time. Consider, for instance, the traffic network model of Boston with approximately 7000 nodes, 20000 arcs and 100 time intervals and 700 destinations. 100 time intervals usually denote 100 seconds of real time. DOT requires approximately 1000 seconds to compute fastest paths from all nodes to 700 destinations for all departure

time intervals. Hence, we can conclude that even an optimal algorithm like DOT does not provide faster than real time solutions that are needed for dynamic management of traffic systems. One way of solving these problems faster, is to use high performance computing platforms.

Chapters 3 and 4 describe the concepts of high performance computation and the different parallel implementations developed.

Chapter 3

Parallel Computation

In Chapter 2, we presented efficient algorithms to solve dynamic shortest path problems. While developing those algorithms, we have implicitly assumed that they will be executed on a machine that can do only a single calculation at a given time. In the same chapter, we had concluded that these implementations do not solve dynamic shortest path problems fast enough for Intelligent Transportation Systems (ITS) applications. One of the ways of improving the speed of these algorithms is to use parallel computers.

Parallel computers have multiple processors. These processors simultaneously solve a given problem and this may involve collaboration between them. Chapter 4 presents parallel algorithms for dynamic shortest path problems exploiting parallel computing technology. To develop parallel algorithms, a thorough understanding of parallel computing concepts is necessary.

This Chapter is organized as follows: In Section 3.1, we present a classification of the parallel computers. In Section 3.2, we describe the two kinds of parallel systems used for developing the parallel implementations developed in this thesis. Then, in Section 3.3, we present the issues that are common to parallel computing which do not arise in the case of sequential computing. In Section 3.4, we describe the sequence of steps involved in developing parallel implementations. Finally, in Section 3.5, we present the measures used to assess the performance of a parallel program.

3.1 Classification of Parallel Systems

The growing need for parallel computation in different scientific fields has led to the development of a variety of parallel computers. These parallel computers differ from each other in many ways. Hence, to develop efficient parallel implementations, we need to choose a suitable parallel computing architecture for a given problem. For this, a systematic classification of parallel computers is necessary to understand the different parallel computing paradigms and to choose the most suitable parallel computer for a given problem.

Chabini et al [9] describe the following two modes of classification of parallel computers:

- **Number and Type of Streams of Instructions and Data:** This is one of the first and most popular classification due to Flynn [16]. The classification distinguishes computers by the number and types of streams they permit. This leads to four classes of computers:
  - Single Instruction Single Data (SISD): In this model, the processors execute a single instruction stream on a single data item. Hence, it corresponds to the classical sequential computers.
  - Multiple Instruction Single Data (MISD): This model is not useful.
  - Single Instruction Multiple Data (SIMD): In this model, all the processors execute a single set of instructions on different data items. For example, the connection machine CM-2 is based on such a kind of model. Parallel implementations of shortest path algorithms using such machines have been reported in the literature (see Habbal et al [18], Ziliaskopoulos et al [30]).
  - Multiple Instruction Multiple Data (MIMD): In this model, processors can execute multiple instructions on multiple data. Most parallel machines are based on this model. For example, a network of workstations can be considered as an MIMD parallel system.

- **Hardware Characteristics:** Flynn's classification helps to categorize parallel computers, but, it does not present the parameters required for parallel implementation of algorithms. These implementations should take advantage of some hardware characteristics. Parallel computers differ a lot in their hardware characteristics. The most important of these hardware characteristics are:

  - **Type and Number of Processors:**

    * *Massively Parallel Computers / fine grained parallel systems:* These parallel computing systems have thousands of processors. The *grain* refers to the amount of RAM memory available to each processor. In these machines, each processor has only 16kB of RAM, hence these machine as also called is small *fine grained parallel systems.* They can be used for solving very huge computational problems. CM-2 machine is an example of massively parallel systems.

    * *Coarse grain parallel systems:* These parallel systems contain a small number of processors, usually in the order of 10. Each processor can handle a large amount of data, of the order of megabytes. Most parallel systems are of this type. They are particularly attractive because these computers are affordable by most organizations. All the parallel implementations developed in this thesis use these parallel systems.

  - **Synchronous vs. Asynchronous Operation:** The distinction in this case refers to the presence or absence of a common global clock used to synchronize the operations of the different processors. Machines with a single global clock are called synchronous, while, those without the global clock are called asynchronous.

    For example, SIMD machines are synchronous because all the processors in a SIMD machine process a single instruction stream.

  - **Processor Interconnection:** An important requirement of parallel computation is the communication of the required information between the processors. Processors may communicate in different ways. The main

categories into which the parallel computers are classified based on this parameter are:

* *Shared Memory:* In this design, there exists a global shared memory which can be accessed by all the processors. A processor can communicate with another by writing into a memory location in the global memory and the other processor requiring this information, reads from the same memory location in the global memory. Though this solves the inter processor communication problem, it introduces the problem of resolving conflict of simultaneous modification of the same memory location by different processors. This problem is usually solved using *mutual exclusion locks.* These are discussed in greater detail in Section 3.4.3. The processors may not try to modify the value in the memory location, but just try to access/read it simultaneously. This operation calls for complex switching circuits, resulting in longer memory access times. Figure 3.1 illustrates a shared memory system. This figure shows that processors P_1, P_2, P_3 and P_4 have access to a global memory through a switching circuit.

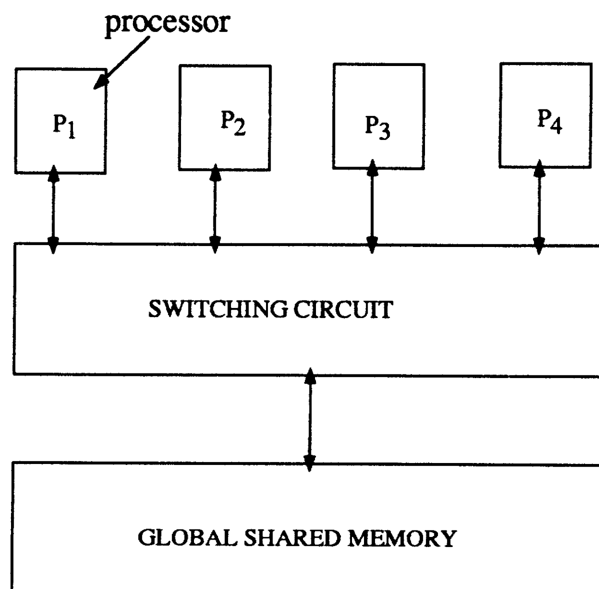


Figure 3.1: Shared Memory System

* *Message Passing / Distributed Memory systems:* Each processor has its own local memory and communicates through an interconnection network. It has a topology which describes how processors are connected. The most common topologies are the ring, the tree, the mesh and the hypercube. An appropriate topology can be has to be chosen depending on the communication requirements of the parallel algorithm. A major factor effecting the speed of parallel algorithms developed for these systems is the amount of time taken to communicate between the processors. We will notice in the next chapter that for certain decomposition strategies, the communication requirements of these systems can be high. In such a case, shared memory systems are proven to be better than distributed memory platforms. Figure 3.2 shows an example of a distributed memory system. In this figure, P_1, P_2, P_3 and P_4 are four processors connected by an interconnection network and M_1, M_2, M_3 and M_4 denote their local memories.

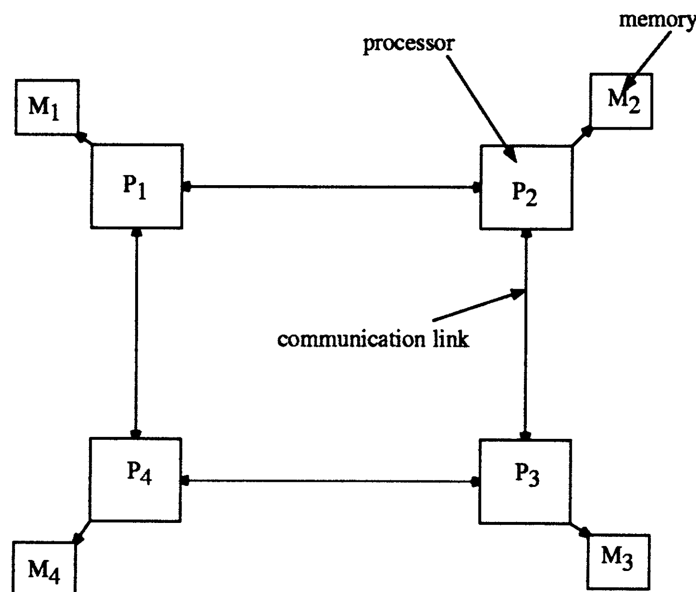


Figure 3.2: Distributed Memory System

Several hybrid choices are also available where some or all the processors have a local memory and communication links and share a global memory.

These designs are not discussed in this thesis. Interested reader is referred to Bertsekas and Tsitsikilis [4].

In the next section, we discuss the parallel computing systems used to develop the parallel programs described in Chapter 4.

3.2 Parallel Computing Systems Used in This Thesis

One of the objectives of this thesis is to develop parallel implementations of dynamic shortest paths algorithms for both distributed and shared memory platforms. Parallel systems used to develop these parallel implementations are the following:

- **A Sun Ultra HPC 5000 symmetric Multiprocessors' Cluster (*Xolas*):** These machines are used to develop the shared memory implementations of dynamic shortest path algorithm. A Symmetric Multiprocessing (SMP) system can be defined as a MultiProcessing system, where all CPUs can share all or a part of the main memory with the other CPUs, and have equal access to all peripherals. All CPUs are equal, and hence, any of the CPUs can do any work, including scheduling work inside the kernel.

  MIT is equipped with a cluster of SMPs called the Xolas cluster, providing access to high performance scientific computing. The Xolas cluster has nine HPC 5000 machines. All the workstations are Sun Ultra HPC 5000 Machines. Their configuration is:

  - 512MB RAM
  - 8 CPU
  - 1GB of swap memory
  - Solaris 2.5.1/2.6 operating system

  More information on this cluster can be found at URL: *http://xolas.lcs.mit.edu.* We will demonstrate in Chapter 4 that shared memory implementations on

these systems show a significant improvement in the computation time of the dynamic shortest path algorithms.

- **A Network of SGI Workstations:** This network is used for the distributed memory implementations of dynamic shortest path algorithms. In this network, we have a set of 6 SGI workstations connected through by a Local Area Network.

In addition to the parallel computers, we need a parallel programming language to develop a parallel implementation. Parallel programming is different from sequential programming in many ways. These differences are discussed in the next section.

3.3 How is Parallel Computing different from Serial Computing?

There are some generic issues common to parallel computing which do not arise in serial computing. It is important to understand these issues in order to design efficient parallel implementations. The following issues distinguish parallel computing and serial computing:

- **Decomposition, Task Allocation and Load Balancing:** In parallel computing, the given task is divided into sub tasks that can be executed simultaneously. The task allocation consists of assigning each of these subtasks to a processor and coordinating the activities of all the subtasks. For efficient parallel implementations, the load on different processors should be balanced. Hence, the objective of load balancing is to obtain subtasks with the most possible uniform execution time.

- **Communication:** Communication is one of the major overheads of parallelization. A decomposition strategy should be designed in such a way that communication required between subtasks is as minimum as possible. In the next chapter, we will see that for dynamic shortest paths algorithms, certain decomposition techniques may require considerable amount of communication

between subtasks. Hence, those parallel implementations may not be worthwhile.

We have noted earlier that the information exchange is carried out using an interconnection network between the processors in distributed memory platforms and through the global memory in shared memory platforms. Communication between subtasks in distributed memory implementations should be carried out efficiently by exploiting the topology of interconnection network.

- **Synchronization:** In some parallel algorithms, one or more processors have to wait till the completion of certain computations or arrival of certain data to proceed with the next step. Hence, these processors should be synchronized. The process of synchronization can slow down the parallel implementation. Most software libraries provide many different ways of synchronization. Some of them will be discussed in Section 3.4.3.

  For a given problem, it may be possible to design both synchronous and asynchronous algorithms. Asynchronous algorithms are usually faster, but, are more complex to design [3].

- **Idle Time:** Idle time is the time lost when one or more processors are not computing. This may result due to following factors: load imbalance, synchronization and communication. Hence, for an efficient parallel implementation, idle time should be minimized.

- **Termination Detection:** The information available to each subtask is local. Usually, the termination condition of the given subtask may require a global condition to be satisfied. Hence, special algorithms may be required to detect termination, so that each process can know when it can quit using the information available to it.

Hence, to develop a parallel implementation, we need to first identify a sequential algorithm, identify certain parallelization strategies in that algorithm, choose a

suitable parallel architecture and use the software development tools associated with this parallel system to implement the parallel algorithm.

In the next section, we describe the different dimensions along which an algorithm can be decomposed and the different software libraries to develop the parallel programs.

3.4 How to Develop a Parallel Implementation?

In this section, we first discuss three general decomposition mehods. We then present one way of scheduling the different tasks in the parallel program. Then, discuss the various software libraries that can be used to develop parallel programs.

3.4.1 Decomposition Methods

The decomposition strategy that can be used for a given problem depends on the logic in the solution algorithm of the problem. The following are three general decomposition methods as discussed by Ragsdale [28]:

- **Perfectly Parallel Decomposition:** Certain applications are perfectly parallel by nature. These can be easily decomposed into subtasks with almost no communication between the subtasks.

  For example, calculation of all to one dynamic shortest paths for all departure time intervals for all the destinations in the network in a shared memory system can be considered as a perfectly parallel problem. Each processor can run algorithm `DOT` (see Section 2.7.2) for a subset of destinations. Hence, no communication is required between the subtasks.

- **Domain Decomposition:** Certain problems have a large data domain on which an algorithm is applied. For these problems, we can decompose the domain of computation into sub domains. Each processor updates its data by applying the algorithm to its subdomain while collaborating with other processors. A major bottleneck of this decomposition strategy is the amount of

information required by a processor from other processors to update its data. For example: in traffic problems, shortest paths computations are required for huge networks. The network on which the shortest path algorithm is applied can be viewed as the "domain" of the problem. Certain parallel implementations of shortest path algorithms reported in the literature ([18], [21]) use this domain decomposition strategy. The network is decomposed into subnetworks. One subnetwork is allotted to each processor. Each processor updates the labels of nodes in its subnetwork. To update its labels, each processor needs information about the labels of the nodes in the subnetworks of the other processors.

- **Functional Decomposition:** This decomposition strategy is based on the flow of control in the algorithm. The algorithm is represented as a set of modules/operations, that express the algorithm's functional parts. Hence, if the logic of the solution algorithm permits, modules that can simultaneously executable are allotted to different processors. For example, Dynamic traffic assignment problem can be decomposed into different modules: shortest path generation, network loading, users' behavior model etc (see [19], [22]). Lacagnina and Russo [22] design a parallel implementation of this problem using the functional decomposition technique to decompose the problem.

3.4.2 Master/Slave Paradigm

One of the ways of implementing a parallel algorithm is using the master/slave paradigm. In this paradigm, we have one master process and many slave processes. The master process decomposes the problem into subproblems and spawns out slave processes to solve each of the subproblems. In this technique, all slave process usually report back to the master process after they solve their subproblems. We use the master/slave paradigm in all the parallel implementations described in Chapter 4.

3.4.3 Software Development Tools

An important question that still remains is: how do we develop parallel programs? Depending on the kind of parallel computer being used, a parallel programming language/ software library designed for that computer is used to develop the parallel program. This subsection describes the basic concepts of the software libraries used in Chapter 4 to develop parallel implementations of dynamic shortest paths algorithms.

We use PVM (Parallel Virtual Machine [17]) library to develop distributed memory implementations and Solaris Multithreading (MT) library [24] for shared memory implementations. All implementations are developed in C++, using the appropriate `''.h''` file (`pvm3.h` for the PVM based implementations and `thread.h` for MT implementations).

PVM

The Parallel Virtual Machine(PVM) library uses the message passing model to allow programmers to exploit distributed computing across a wide variety of computer types. A key concept in PVM is that it makes a collection of computers appear as one large *virtual machine*. To use the PVM software, the user needs to write his application as a collection of *tasks*, which are implemented as processes. Tasks access PVM resources through a library of standard interface routines. These routines allow for initiation and termination of tasks across the network as well as for communication and synchronization between tasks.

The following PVM library functions are most frequently used in PVM implementations:

- **Spawning tasks:** `pvm_spawn` command is used to spawn tasks. The syntax of this command is:

```
int numt = pvm_spawn(char *task, char **argv, int flag,
char *where, int ntask, int *tids)
```

task: Character string which is the executable file name of the PVM process to be started. The executable must already reside on the host on which should be started.

argv: Pointer to an array of arguments to the executable (if supported on the target machine), not including the executable name, with the end of the array specified by NULL.

flag: Integer specifying spawn options. For more details, the reader can see [17].

where: Character string specifying the computer on which to start the PVM process. Depending on the value of flag, *where* can be a host name such as "voice.mit.edu". If *flag* is 0, then *where* is ignored and PVM will select the most appropriate host.

ntask: Integer specifying the number of copies of the executable to start.

tids: Integer array of length *ntask* returning the tids of the PVM processes started by this *pvm_spawn* call.

numt: Integer returning the actual number of tasks started. Values less than zero indicate a system error. A positive value less than *ntask* indicates a partial failure. In this case the user should check the *tids* array for the error code(s).

- **Sending Data:** Any information that needs to be communicated should be packed using a group PVM pack functions, *pvm_pk** commands into an active send buffer and then *pvm_send* or *pvm_mcast* should be used to send the information. *pvm_send* is used to send the message to a particular task. But, *pvm_mcast* is used to broadcast the message to a group of n tasks.

  If the same information needs to be sent to a group of n tasks, broadcasting the message would require $\log n$ times the time required to send the information to one task. While sending the information to each of the separately would require n times the time required to send the information to one task. Hence, the sequence of commands used to send an integer (for other data types, the reader is referred to [17]) array would be:

```
int info = pvm_initsend(int encoding)
int info = pvm_pkint(int *np,int nitem, int stride)
int info = pvm_send(int tid, int msgtag)
or
int info = pvm_mcast(int *tids, int ntasks, int msgtag)
```

where

np: integer array atleast $nitem * stride$ items long

nitem: the total number of items to be packed

stride: the stride to be used when packing the items. For example, if $stride = 1$, a contiguous vector is packed, a *stride* of 2 means every other item is packed, so on.

tid: integer task identifier of the destination process

msgtag: integer message tag supplied by the user. ($msgtag \geqslant 0$)

tids: integer array of length at least *ntasks* containing the task *IDs* of the tasks to be sent to.

ntask: integer specifying the number of tasks to be sent to.

info: Integer status code returned by the routine. A value less than zero indicates an error.

encoding: integer specifying the next message's encoding scheme. (for details, the reader can consult [17])

- **Receiving data:** PVM contains several methods of receiving messages at a task. We use only the blocking receive routine, *pvm_recv*, in our parallel implementations. *pvm_recv* places the message into an active receive buffer that is created. Then, a group of *pvm_upk** routines are used to unpack the data from this buffer. Hence, receiving integer data can be done using the following statements:

```
int bufid = pvm_recv(int tid, int msgtag)
int info = pvm_unpkint(int *np, int nitem, int stride)
```

pvm_recv is a blocking receive routine which means that the process will wait till a message with label *msgtag* has been received from the task id *tid*. *nitem* is the number of items of the given data type to be unpacked, and *stride* is the *stride* used in packing (see *pvm_send* for more information).

Using the above commands, we can develop most of the distributed memory applications. As we have noted, developing PVM programs, is not very different from the usual sequential C or Fortran programming. All the PVM programs are a group of sequential codes, with certain communication features embedded in.

Developing multithreaded programs is not as straightforward and needs certain knowledge about the way tasks are scheduled by the operating system. Hence, the next section gives a brief introduction to these concepts and then describes the most common commands required to develop MultiThreaded (MT) programs.

Solaris MultiThreading Library

As mentioned earlier, we develop our shared memory implementations using Solaris Multithreaded library. In multithreaded programs, each sub task is allotted to a *thread*. So the next question: What is a Thread?

Just as multitasking operating system (like UNIX) can work on more than one task concurrently by running more than one process, a process can do the same by running more than a single *thread*. Each thread is a different stream of control that can execute its instructions independently, allowing a multithreading process to perform numerous tasks concurrently.

In Solaris operating system, threads are supported using the concept of *Light Weight Process* (LWP). A lightweight process can be thought of as a virtual CPU that is available for executing code. Each LWP is separately dispatched by the kernel. It can perform independent system calls and incur independent page faults, and multiple LWPs in the same process can run in parallel on multiple processors.

The Solaris multithreaded model can be considered as a two level model, as illustrated in Figure 3.3.

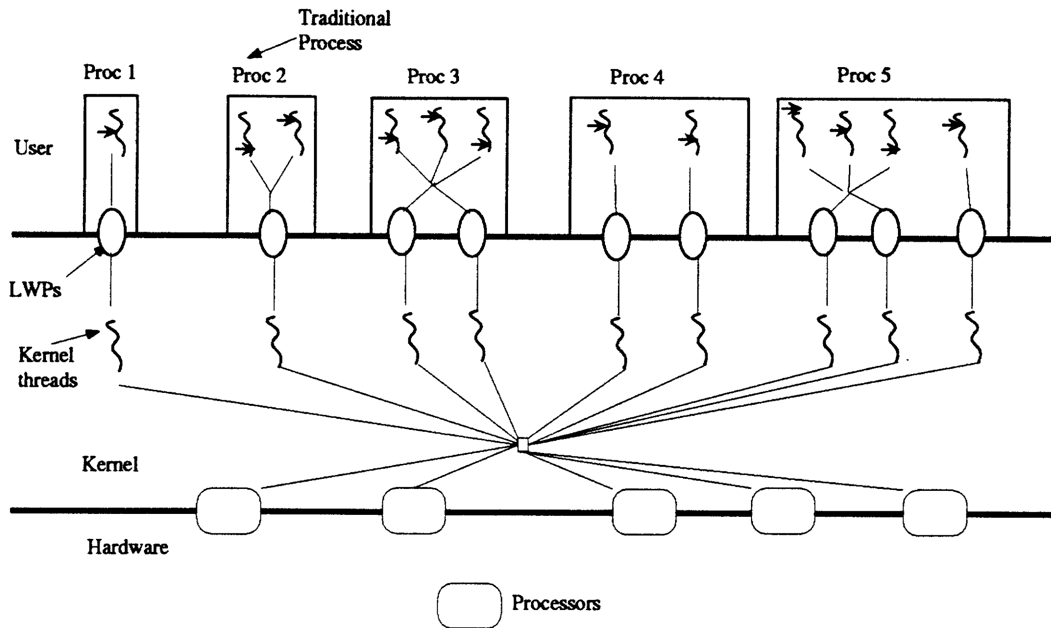


Figure 3.3: The Solaris Multithreaded Architecture (Courtesy: Berg and Lewis [24])

In this model, threads are the portable application level interface. Programmers write applications using the threads library. The library schedules the threads onto LWPs. The LWPs in turn are implemented by kernel threads in the Solaris kernel. These kernel threads are then scheduled onto the available CPUs by the standard kernel scheduling routine, completely invisible to the user. Hence, for concurrent implementation of the threads in multithreaded programs, each thread has to be scheduled to one LWP.

With this brief introduction to the multithreading architecture, we present some basic commands in the threads library which are frequently used in MT implementations.

- **Creating threads:** There is no parent/child relationship between threads as there is for processes. Threads can be created or joined by any thread in the process. In all our MT implementations, we use the master/slave paradigm. The master thread usually creates the required number of slave threads with certain features for a specific subtask. Each slave thread completes its subtask

and returns back to the master process.

The following function is used to create threads:

```
thr_create(void *stackbase, size_t stacksize, void *
(*start_func)(void *arg), void *argument, long flags,
thread_t *newthread);
```

The life of the thread begins with the successful return of the *thr_create* function. The new thread calls the function defined by *start_func* with one argument, *arg*. If more than one argument needs to be passed to *start_func*, the arguments can be packed into a structure, and the address of that structure can be passed to *arg*. The other important argument to the *thr_create* function is *argument* which specifies the type of thread to be created. To create a thread bound to a new LWP, the argument *argument* should be *THR_NEW_LWP*. The thread exits when the *start_func* ends or with the call to function *thr_exit*.

Once the threads are created, the master process waits for the threads to complete the execution of the *start_function* and join it back. This is done using the following function:

```
thr_join(thread_t target_thread, thread_t *departed, void
**status);
```

The *thr_join* function makes the calling process to wait till the completion of the *target_thread*.

- **Locking and UnLocking:** Any concurrent program usually requires a synchronization between the different threads. Recall that in shared memory systems all the threads have access to a global memory. Hence, when dealing with global variables, without synchronization, one thread may start to change some data just as another thread is reading it. The reader thread will get half old data and half new data. To avoid this problem, threads must be able to reliably coordinate their actions.

One of the ways of synchronizing is using the *mutual exclusion locks.* The mutual exclusion lock provides a single, absolute owner for the section of code that it brackets between the calls to *mutex_lock* and *mutex_unlock*. The first thread that calls lock on the mutex gets the ownership, and any subsequent calls to the lock will fail, causing the calling thread to sleep. When the owner calls unlock, one of the sleepers is awakened, made runnable and given the chance to ownership. Any shared global variables have to be locked before modification.

Lewis and Berg [24] report that it takes approximately 48 microsecond to process a local mutex lock with contention by a SUN SPARC Station 10. Moreover, some threads are put to sleep when they do not get the lock. Hence, the use of the locks, though absolutely essential in some cases (global variables), compromises the speed of the parallel program. Hence, care should be taken not to use mutex locks excessively.

Locking and unlocking the code is done using the following functions:

```
mutex_lock(mutex_t *mp);
mutex_unlock(mutex_t *mp);
```

- **Barriers:** In certain parallel implementations, we may require all the threads to proceed to the next step only if they are all done until a certain point in the algorithm. This is achieved by using a barrier. A barrier allows a set of threads to sync up at some point in their code. A barrier has a value associated with it. This value is initialized to the number of threads using the barrier. Each thread reaching the barrier decrements barrier's value by 1. The barrier blocks all the threads calling it until its value reaches zero, at which point it unblocks them all.

  Barriers are not part of the threads library, but can be implemented using the functions available in the threads library. We implement the barrier using *condition variables.*

  A condition variable is a synchronization variable that allows the user to specify arbitrary condition on which to block a thread. Condition variables always have

an associated mutex. A thread obtains the mutex associated with the condition variable and tests the condition under the protection of the mutex. No other thread should change the condition without holding the mutex. If the mutex is true, the thread completes the task, releasing the mutex when appropriate. If the condition is not true, the mutex is released for the user and the thread goes to sleep on the condition variable. When some other thread changes some aspect of the condition, it calls the *cond_signal* or *cond_broadcast* and wakes up the first thread. This thread then reaquires the mutex, reevaluates the condition, and proceeds if the condition is true.

As mentioned earlier, we implement the barrier using these condition variables. A barrier is a C++ class with three variables: *value*, *mutex_t mutex* and *cond_t cond*. The constructor of the class initializes the *mutex* and *cond* and sets the *value* to the number of threads (to be synchronized). Then, at the barrier, we use the following code segment:

```
mutex_lock(&barrier.mutex);
barrier.value = barrier.value - 1;
if(barrier.value == 0)
{ % condition satisfied, wake up the sleeping threads
  mutex_unlock(&barrier.mutex);
  cond_broadcast(&barrier.cond);
}
else
{ % condition not satisfied, sleep on the condition
while(barrier.value > 0)
  cond_wait(&barrier.cond,&barrier.mutex);
mutex_unlock(&barrier.mutex);
}
```

Multithreaded programs often behave differently in two successive runs given identical inputs because of the differences in the thread scheduling order. This makes the

development and debugging of multithreaded programs more difficult than PVM programs or sequential programs.

Hence, the sequence of steps followed by any parallel program described in Chapter 4 can be summed up as: start the master process with the information about the algorithm, decomposition strategy, data set and number of processors. The master process then decomposes the task into different subtasks, starts slaves which are either processes (in distributed memory implementations) or threads (in shared memory implementations), allots one subtask to each slave and waits for the slave tasks to return after completing their subtasks.

Once we have implemented a parallel program, we need certain measures to assess the performance of this parallel program compared to the serial program. The next section describes the various measures used to assess the performance of parallel programs.

3.5 Performance Measures

Chabini et al [9] note that the assessment of a *parallel system* (a combination of an algorithm and a parallel platform) can be done by reporting certain performance measures for different values of the following parameters:

- The size w of the execution: This size represents a measure of the number of basic operations performed by all processors of the parallel machine.
- Number of processors of the parallel machine used, say p.

Chabini et al [9] also note that definition of performance depends on the reason behind using the parallel machine. In the context of this thesis, we wish to reduce the computation time of the dynamic shortest paths algorithms. Hence, we define the performance measures with respect to the computation time, $T(w,p)$. Let the time taken by the serial program to execute the same instance of the problem on the fastest processor of the parallel system $T_s(w)$.

The following performance measures are generally used:

- **Speedup $S(w,p)$:** The speedup is the ratio of the time taken by serial program to that of the parallel program. Hence,

$$S(w,p) = \frac{T_s(w)}{T(w,p)} \tag{3.1}$$

This measure is not robust, as it fails to predict the performance of the parallel program for a larger number of processors than those available on the system.

Usually, we are faced with the following questions: What is the maximum speedup achievable by a given parallel program? or how many processors should we invest in? To answer these questions we require the next performance measure, introduced by Chabini et al [7].

- **Relative Burden $B(w,p)$:** The relative burden measures the deviation from the ideal improvement in time from a uniprocessor execution to a p-processors execution normalized by the serial time:

$$B(w,p) = \frac{T(w,p)}{T_s(w)} - \frac{1}{p} \tag{3.2}$$

Burden can be used to predict both asymptotic and maximum speedup that can be obtained by a parallel program. The asymptotic and maximum speedup measures are obtained in the following manner:

- **Asymptotic speedup:**

Equation 3.2 can be written as

$$S(w,p) = \frac{p}{1 + B(w,p) * p} \tag{3.3}$$

For very small p, the term $B(w,p) * p$ is small, because the overhead of parallelization is small (therefore, burden is small). Hence, most applications can

show an almost linear speedup for small p. From this speedup for small p, we can not estimate what the speedup would be for a higher p. As $B(w,p)$ increases with p, for a large p, equation 3.3 translates to the following equation:

$$\lim_{p \to \infty} S(w,p) = \frac{1}{B(w,p)} \tag{3.4}$$

Thus, we can say that the asymptotic speedup of the given parallel program is $\frac{1}{B(w,p)}$. For instance, if we observe a burden of 0.02 using 8 processors for a certain parallel implementation, we can predict that the asymptotic speedup that can be achieved is $1/0.02 = 50$. The speedup for this instance would be $\approx$ 7, which does not give any idea of this aymptotic speedup.

- **Maximum speedup:** Equation 3.2 translates to the following equation when it is differentiated by p.

$$\frac{\partial B(w,p)}{\partial p} = \frac{\partial S(w,p)}{\partial p} + \frac{1}{p^2} \tag{3.5}$$

At maximum speedup, $\frac{\partial S(w,p)}{\partial p} = 0$. Thus maximum speedup can be obtained by solving the equation $\frac{\partial B(w,p)}{\partial p} = \frac{1}{p^2}$.

We report on both the above performance measures for all the parallel implementations developed in Chapter 4.

Chapter 4

Parallel Implementations of Dynamic Shortest Paths Algorithms

In Chapter 2, we presented the formulations and sequential algorithms used to solve two types of dynamic shortest path problems: one-to-all dynamic shortest paths for one departure time interval, all-to-one dynamic shortest paths for all departure time intervals. Algorithms DOT and DOT-MinCost are used to solve all-to-one dynamic fastest paths and all-to-one dynamic min-cost path problems respectively. They are optimal and therefore, no other algorithm to solve these problems with a better running time complexity can be found.

Dynamic shortest path problem is a fundamental problem in many Intelligent Transportation Systems (ITS) models. Faster than real time solutions of dynamic shortest path problems are critical for the operation and evaluation of Intelligent Transportation Systems. Optimal algorithms like DOT also do not solve realistic dynamic shortest path problems faster than real time. Consider, for instance, the traffic network of the city of Boston. This network can be modeled as a dynamic network with 7000 nodes, 20000 links, 100 time intervals and 700 destinations. To solve for dynamic shortest paths from all nodes to these 700 destinations for all the 100 departure time intervals, although optimal, algorithm DOT would require 1000 seconds on

a SUN SPARC workstation. Hence, an optimal sequential algorithm does not solve a realistic dynamic shortest path problem faster than real time.

In Chapter 3, we introduced the concept of parallel computation. Parallel computation involves using multiple processors simultaneously to solve a given problem. Hence, a given problem is divided into subproblems and each processor solves one subproblem collaborating with other processors. We use this parallel computing paradigm to develop faster implementations of solution algorithms of dynamic shortest paths problems.

A parallel implementation requires three components: firstly, a decomposable solution algorithm for the problem, secondly, a strategy to decompose the operations of this algorithm and finally, a parallel computing environment to implement the algorithm. The algorithms we consider for decomposition are those described in Chapter 2. In this chapter, we present the following:

- Dimensions of dynamic shortest path problems that can be used for their decomposition
- Parallel implementations for different algorithms, decomposition strategies and parallel computing platforms combinations.
- Extensive evaluation of all the parallel implementations.

This Chapter is organized as follows: In Section 4.1, we give an overview of all the parallel implementations developed in this chapter. Sections 4.3 through 4.10 describe the parallel algorithms and experimental evaluation of different parallel implementations. Section 4.11 summarizes this Chapter.

4.1 Parallel Implementations: Overview

In this section, we briefly review the sequential dynamic shortest path algorithms considered for parallel implementations, then present five strategies that can be used to decompose them. We will then discuss the issues common to all the parallel

implementations developed. The details of each specific parallel implementation will be discussed in later sections.

As noted earlier, we use develop parallel implementations of some algorithms presented in Chapter 2 to develop. These algorithms are:

- `ls-heap`: Label setting algorithm using heaps to compute one-to-all shortest paths for FIFO networks.
- `lc-dequeue`: Label correcting algorithm using dequeue for one-to-all shortest paths for FIFO networks.
- `dial-buckets`: Dial's implementation of label setting algorithm using buckets, again for one-to-all shortest paths for FIFO networks.
- `IOT`: Increasing order of time algorithm to compute one-to-all fastest paths for non-FIFO networks.
- `DOT`: Decreasing order of time algorithm to compute all-to-one fastest paths for all departure times.

Dynamic shortest path problems and and their solution algorithms can be decomposed along the following dimensions:

- **Destination**: The all-to-many dynamic shortest paths problem can be decomposed on the destination dimension. The idea is to divide the set of destinations into subsets. Then, allot one subset to one processor and assign each processor the computation of all-to-one dynamic shortest paths for its subset of destinations.
- **Origin**: In many-to-all dynamic shortest path problems, we can use the set of origins as a decomposition dimension. An idea similar to the one described above for destination based decomposition strategy can be employed in this case too.

- **Departure Time**: This is similar to the earlier decomposition strategies. One or many-to-all dynamic shortest paths, for many departure time intervals problems can be decomposed along the set of departure time intervals.

- **Network Topology**: The network is split into subnetworks. One subnetwork is allotted to each processor. Each processor updates the labels of only those nodes, that belong to its subnetwork.

  Parallel implementations of static shortest path algorithms using this decomposition technique have been developed in the literature (see [18], [21]). To the best of our knowledge, no such implementation of dynamic shortest path algorithms has been done.

  In Section 4.7, we use this technique to decompose the operations of algorithm DOT, with a little overhead of synchronization of the threads in a shared memory environment.

- **Data Structure used in the solution algorithm**: Most shortest path algorithms use certain data structures, such as dequeue, 2-queue or buckets, to compute shortest paths efficiently. These data structures offer another dimension of parallelization.

  For example, in label correcting algorithms, all the nodes which are capable of updating the labels of any node in the network are held in a list. This is called the candidates list. In a label correcting algorithm, one would delete the first node in the list and update the labels of the outgoing nodes of this node, then, add the updated nodes to the candidates list. This is called processing of a node. Next, the second node in the list is deleted and the same procedure is repeated. In a parallel environment with p processors, upto p nodes can be removed from the list and processed simultaneously. It is important to note that processors should be coordinated when a label is being updated, as two processors should not try to update that label at the same time. Bertsekas et al [3] have developed parallel asynchronous implementations of static shortest path algorithms based on the same idea. Ziliaskopoulos et al [30] implement

a similar decomposition technique for a dynamic fastest path algorithm. They report a speedup of approximately 1.5 using 4 processors for a network having 500 nodes, 1250 arcs and 240 time intervals.

We designed a parallel version of algorithm IOT by decomposing the buckets used at each time interval in this algorithm (see Section 2.5.2). The implementation of this parallel algorithm is still under development.

The above decomposition strategies can be classified into two main categories: Application level and Algorithm level. Application level decomposition strategies are used to decompose applications of a basic algorithm. The applications considered here are repetitive extensions of the algorithms. For example, one way to compute all-to-many dynamic shortest paths is to apply algorithm DOT iteratively for all the destinations. Hence, the all-to-many shortest paths problem can be solved by an application of algorithm DOT.

Algorithm level decomposition strategies are used to decompose the operations of an algorithm. Hence, of the above decomposition strategies, *destination*, *origin* and *departure time interval* are application level decomposition strategies. *Network topology* and *data structure* used in a solution algorithm are algorithm level decomposition strategies.

To develop a parallel implementation, we need a parallel computing environment in addition to the algorithms and decomposition strategies discussed above. We use two kinds of parallel computing environments: distributed memory and shared memory. In a distributed memory system, each processor has its own local memory and communicates with other processors using an interconnection network. In a shared memory system, there is a global memory and each processor has access to this global memory. A processor communicates with another processor by writing into a memory location in the global memory and another processor reads from this memory location. We studied these parallel platforms in a greater detail in Chapter 3.

A number of parallel implementations can be developed combining the above mentioned three components of a parallel program: algorithm, decomposition strategy

and parallel computing environment. Figure 4.1 illustrates the different parallel implementations developed at the application level. Figure 4.2 demonstrates all parallel implementations developed at the algorithm level.

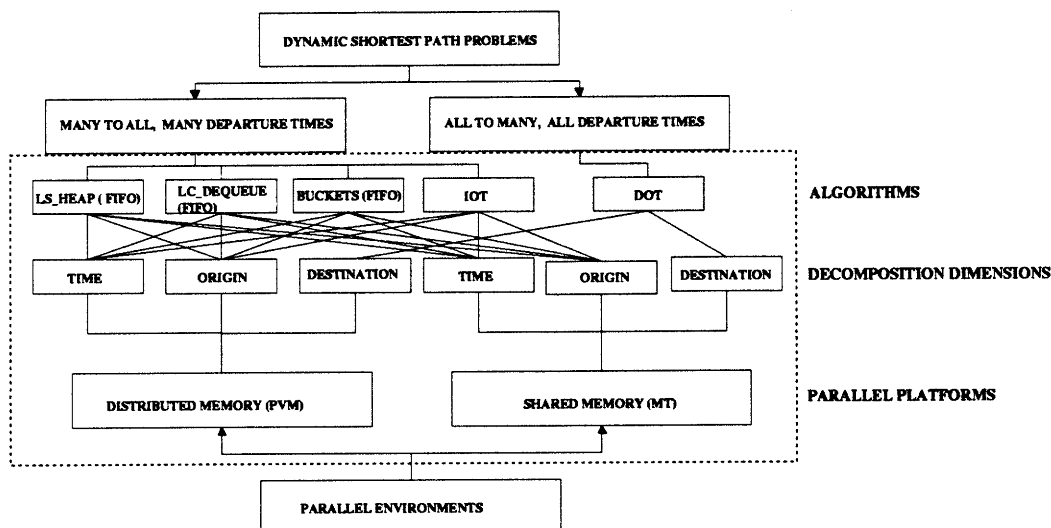


Figure 4.1: Application Level Parallel Implementations

All the parallel implementations are developed using a master/slave paradigm. In this paradigm, the master process knows the whole problem, it decomposes the problem into subproblems and spawns out slaves to solve subproblems. The slaves are processes in distributed memory implementations and threads in the shared memory implementations.

There is an important difference between a distributed and a shared memory implementation. In order to compute shortest paths, every processor needs information about a part of the network or complete network information, depending on the decomposition strategy. On a distributed memory platform, each processor has its own local memory. Hence, the network is replicated and stored in all the local memories. In a shared memory platform, there exists one shared global memory and all the processors have access to this global memory. Therefore, only one copy of the network is present in the global memory and all the threads work on this copy.

For an application level decomposition strategy, all the processors need complete network information because all of them run the same algorithm on the whole network

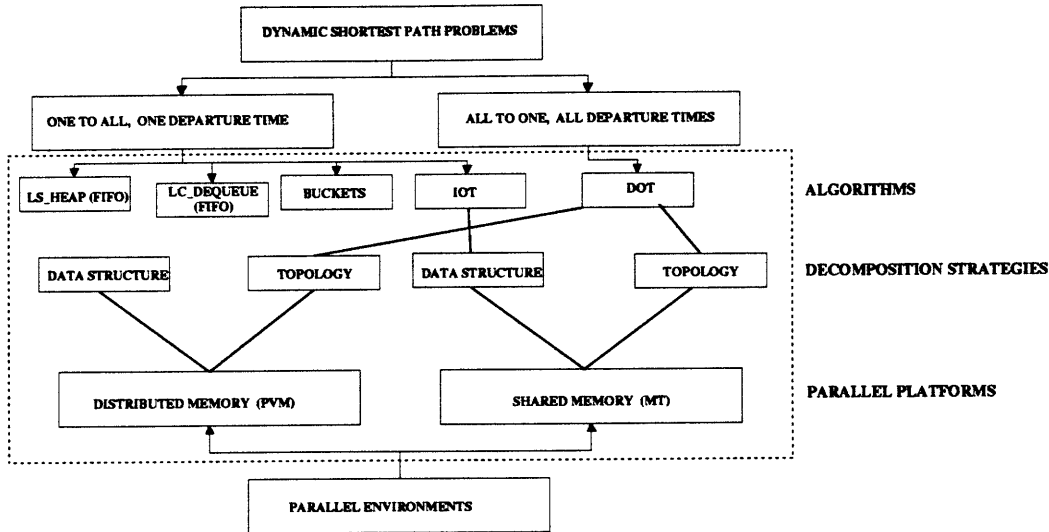


Figure 4.2: Algorithm Level Parallel Implementations

for different destinations/ origins /departure time intervals. Hence, the whole network is replicated in a distributed memory implementation. Figure 4.3 illustrates this idea. In this figure, we illustrate the use of two processors to calculate dynamic shortest paths from all nodes to two destination nodes, namely, node 1 and node 2. The letters $P1$ and $P2$ denote two different processors and $M1$ and $M2$ denote the memory systems of these processors. Processors $P1$ and $P2$ have their own copies of the network. Processor $P1$ calculates dynamic shortest paths from all nodes to the destination node 1 and processor $P2$ calculates shortest paths from all nodes to the destination node 2.

Figure 4.4 illustrates the application level decomposition strategy in a shared memory environment. In this figure, $T1$ and $T2$ are two threads calculating dynamic shortest paths to destinations 1 and 2 respectively. As mentioned earlier, there is only one copy of the network. It is stored in the global memory to which both threads have access.

If we decompose by network topology, each processor in a distributed memory implementation stores its subnetwork. The information about the boundary links need to be communicated between the processors. While, in a shared memory implementation, all the subnetworks reside in the same memory, each thread works on its

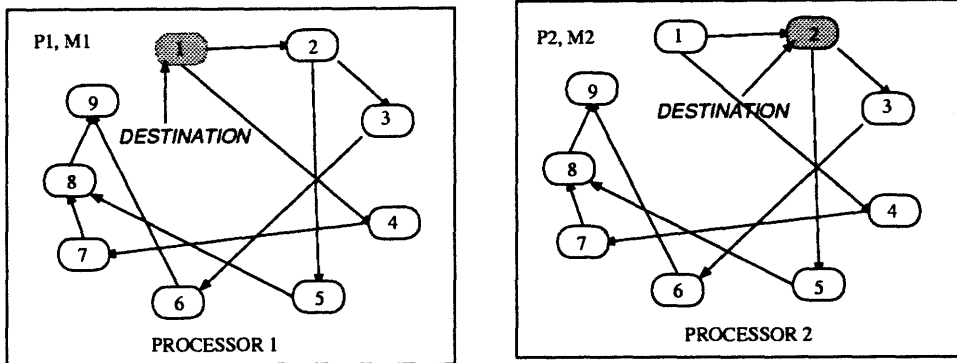


Figure 4.3: Illustration of implementation on a distributed memory platform of an application level (destination-based) decomposition strategy

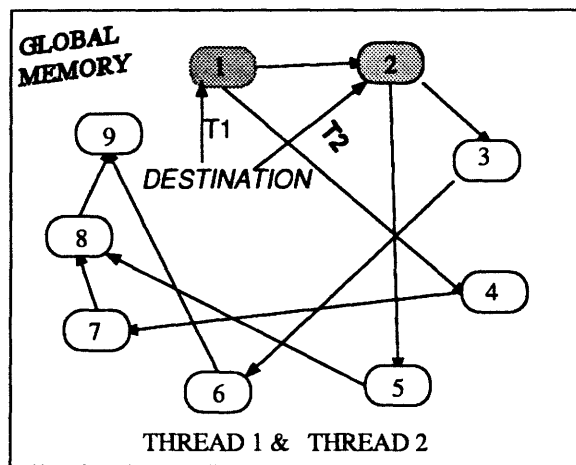


Figure 4.4: Illustration of implementation on a shared memory platform of an application level (destination-based) decomposition strategy

portion of the network. Every thread can access the information in another thread's subnetwork.

Figure 4.5 illustrates the network decomposition technique in distributed and shared memory environments. In this figure, we show a small test network decomposed into two parts. We show that in a distributed memory environment, the subnetworks reside in two different memory units belonging to different processors. We also see that there are some links in the network which go from one subnetwork to the other (these are shown by dotted lines). The labels of the tails and heads of these links need to be communicated between the two processors. In a shared memory environment, both subnetworks are present in the same memory unit. While two different threads work on the two subnetworks, both these threads have access to all the network information. We will see that this makes the shared memory implementations much faster than their distributed memory counterparts.

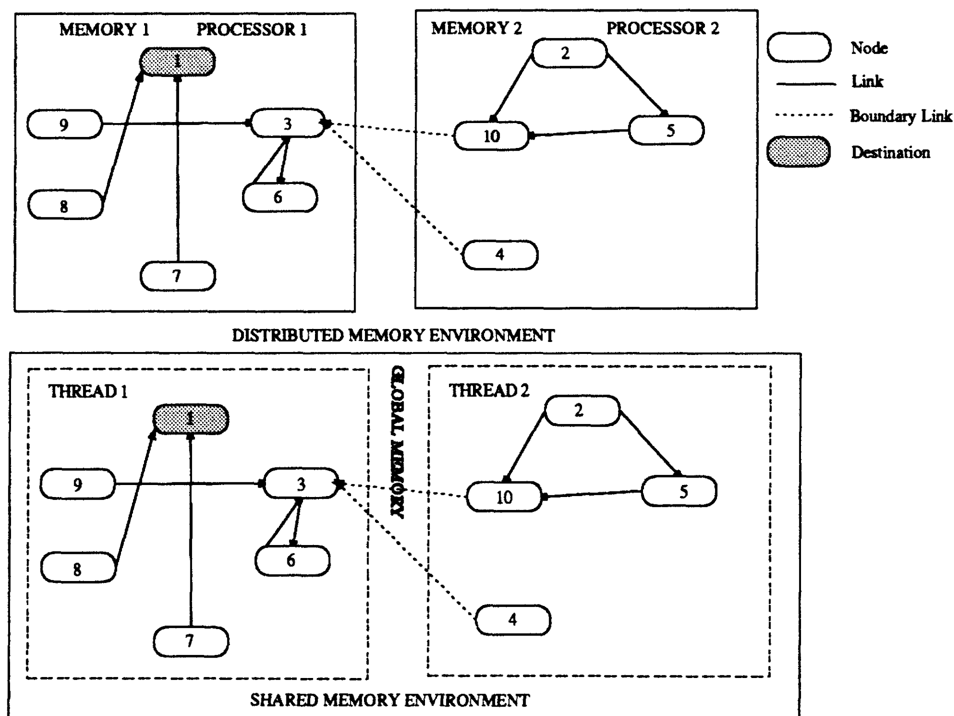


Figure 4.5: Illustration of Decomposition by Network Topology

After this broad overview about parallel implementations, we discuss the specific

implementations in detail in the rest of this chapter. We present the master algorithm, slave algorithm and extensive evaluation for the following implementations:

- Distributed memory application level parallel implementations,
- Shared memory application level parallel implementations,
- Distributed memory implementation of algorithm DOT by decomposition of network topology,
- Shared memory implementation of algorithm DOT by decomposition of network topology and
- Shared memory implementation of algorithm IOT by decomposition of data structure (buckets) used in algorithm IOT.

4.2 Notation

We use the following notation in the description of application level parallel implementations:

$G(N, A, C, D)$	:	dynamic network as defined in Section 2.3,
MP	:	master Process/ thread,
p	:	number of processors,
$P = \{1, \ldots, p\}$	:	set of slave processes/ threads,
H	:	generic algorithm,
k	:	number of origins/destinations/departure
$K = \{1, \ldots, k\}$	:	set of origins/destinations/departure times
K_i	:	subset of K allotted to slave process $i, \bigcup_{i=1}^{p} K_i = K$.

K is decomposed into K_i using the algorithm $DECOMP$. Let $k = q * p + r$, where q is the quotient of the division of k by p and r is the remainder. Then, the algorithm $DECOMP$ is :

```
DECOMP
Set start = 1
for all i ∈ P
    if (i < r) then set end = start + q + 1
    if (i ⩾ r) then set end = start + q
    K_i = {start, ..., end}
end for
```

The algorithm $DECOMP$ given above divides the set K into almost equal subsets. But, this kind of decomposition is not efficient for the set of departure time intervals (when the set of departure time intervals is considered as $\{0, \ldots, M-1\}$). It was observed in Section 2.10 that the time required to compute the dynamic fastest paths travel time labels decreases with the departure time interval. Hence, the above decomposition will lead to difficult time intervals shortest path problems would go to the earlier slave processes and the easier ones will go to the later slave processes. Therefore, this creates a load imbalance. Thus, we use a round robin method to allot departure time intervals to the slave tasks. The loads on different slave tasks are more balanced. With this technique, task 1 computes for $(1, p+1, 2p+1, 3p+1, \ldots)$ departure time intervals, and task 2 computes for departure time intervals $(2, p+2, 2p+2, 3p+2, \ldots)$ and so on.

4.3 Distributed Memory Application Level Parallel Implementations

In a distributed memory implementation, the processors do not share the same memory. Each of them should have its own copy of the network. Therefore, the network information has to be communicated to all the slave processes by the master process. Moreover, dynamic shortest path results are stored in different memory systems. Hence, the master process needs to collect the results from all the slave processes. In the shared memory environments, all the threads have access to a global memory. There is only one copy of the network and all the threads work on this network. The

results are also available in the same memory system, and collection of results is not required. We will see that the collection of results requires some communication time that may hinder the performance of distributed memory implementations. These may not be as efficient as shared memory implementations.

The collection of the results by the master process can be done in two ways: One, the slave process sends the results as soon as the computation for one value of the decomposition dimension, say j is done. Hence, while the slave process computes for the next value of the dimension, $j+1$, the master process receives the data for j. Two, slave process computes for all the values of the decomposition dimension and then sends all the results once, at the end. We can see that the second strategy leads to some idle time of the master process and the slave processes. Hence it is a inefficient strategy. We use the first strategy to collect the results.

The algorithm used to develop a distributed memory implementation of an application level decomposition strategy will be presented in two parts, namely, the master process algorithm and the slave process algorithm.

4.3.1 Master process algorithm

The master process algorithm is:

Master Process (Application level parallel implementation)

1. Read the dynamic network $G(N, A, C, D)$.
2. Decompose K into subsets K_i using algorithm $DECOMP$.
3. Spawn child process $i, \forall\ i \in P$
4. Broadcast network G to all $i \in P$.
5. For all $i \in P$, send K_i to i.
6. While $(K \neq \phi)$

   For all $i \in P$

   If $(K_i \neq \phi)$ then

   · Receive the dynamic shortest path labels for dimension $j \in K_i$ from processor i.

   · $K_i = K_i - \{j\}$, $K = K - \{j\}$.
7. Broadcast the message "quit" to all $i \in P$.
8. Stop.

4.3.2 Slave process algorithm

The slave process algorithm is:

Slave Process (Application level parallel implementation):

/*$i \in P$ denotes a slave process */

1. `Receive the network` G `from the master process` MP.

2. `Receive subset` K_i `from the master process` MP.

3. `For all` $j \in K_i$

   – `Run Algorithm H.`

   – `Send the dynamic shortest path labels for dimension` j `to the master process` MP.

4. `Wait for the message "quit" from the master process` MP.

5. `Stop.`

4.3.3 Run Time Analysis

Let us denote by c, the average time required to communicate one unit of data between two processors. Let $O(H)$ denote the worst run time complexity of algorithm H.

Then, the worst case run time complexity can be given by the following proposition.

Proposition 19 *The worst case run time complexity of the distributed memory application level parallel implementation is given by:*

$$O(m * M * c * \log p + Max(\frac{k}{p} * n * M * c, \ \frac{k}{p} * O(H))) \tag{4.1}$$

Proof: The first term ($O(m * M * c * \log p)$) in the worst case run time complexity denotes the amount of time required to communicate the network to all the processors by the master process. The amount of information to be communicated is in the order of mM. This corresponds to the links travel times for all the time intervals. As we broadcast the information instead of sending it to each processor, the communication time required to send all link travel times to all the processes is $O(mMc * logp)$.

The second term denotes the time that the slave processes take to compute the dynamic shortest path labels and to communicate these labels to the master process. We have noted earlier that we interleave the computation of dynamic shortest paths and the communication of results. The total time taken to compute the dynamic shortest paths for k dimensions by p processors using algorithm H is $O(\frac{k}{p} * O(H))$. The taken to communicate the results is $O(knM * c)$. This corresponds to as nM labels for k dimensions. As these procedures are interleaved. The worst case run time complexity is the maximum of complexities of worst case run time complexities of both procedures. □

In the next section, we discuss the shared memory implementation of application level decomposition strategies.

4.4 Shared Memory Application Level Parallel Implementations

In shared memory implementations, each slave task is allotted to a different thread. In these implementations, only one copy of the network is maintained in the global memory (see figure 4.4). As all the threads have access to all the data and results, the master task need not collect the results from the slave processes.

4.4.1 Master thread algorithm

The master thread algorithm is:

Master Thread (Application level parallel implementation):

```
1. Read the network G(N,A,C,D).
2. Decompose K into subsets K_i.
3. Create child threads i, ∀i ∈ P
4. Wait for the slave threads to join.
5. Stop.
```

4.4.2 Slave thread algorithm

The slave thread algorithm is:

Slave Thread (Application level parallel implementation):

/*$i \in P$ denotes a slave process */

```
1. For all j ∈ K_i
     - Run Algorithm H for j on network G.
2. Exit
```

It can be seen from the above algorithms that there is no explicit communication required in the shared memory implementations. But, the "equivalent" to communication is the contention of the threads to access the same memory location. This increases the memory access time, and hence the running time of the parallel program. This will be noticed in the experimental results presented in the next section.

4.4.3 Run Time Analysis

To obtain the worst case run time complexity of the above shared memory implementation, we denote the percentage of computation time lost due to contention of threads to access a memory location by μ. We assume that μ is a constant. Then, the worst case run time complexity is given by the following proposition.

Proposition 20 *The worst case run time complexity of an application level shared memory implementation is:*

$$O(\frac{k}{p} * O(H) * (1 + \mu)) \tag{4.2}$$

Proof: The computation time required by each thread is equal to $\frac{k}{p} * O(H)$. From the definition of μ, the worst case run time complexity is equal to the above result.$\square$

4.5 Experimental Evaluation of Application Level Parallel Implementations

4.5.1 Numerical tests and results

We have done an extensive evaluation of the above parallel implementations to understand the performance of each algorithm and to compare the performance of the different parallel implementations of the same dynamic shortest path algorithm. The results of the experimental evaluation and the conclusions of this evaluation are presented below. We use two different computer systems for our evaluation: `a distributed network of Silicon Graphics Indy (SGI) workstations` and a `cluster of SUN Symmetric Multiprocessors called Xolas`. These are described in Section 3.2. As mentioned earlier, we use the PVM library to develop the distributed memory implementations. The Solaris MultiThreading (MT) library is used to develop the shared memory implementations. In the plots and tables, `PVM-Xolas` refers to a distributed memory implementation and `MT-Xolas` refers to a shared memory implementation on the Xolas system. `PVM-SGI` refers to a distributed memory implementation on the network of SGI workstations. For all the parallel implementations, we show both the speedup and burden measures as a function of the number of processors.

The evaluations done are presented below:

- **Evaluation of `PVM-Xolas`, `PVM-SGI` and `MT-Xolas` implementations for**

all-to-many dynamic fastest path problem using algorithm DOT and destination based decomposition strategy for a network of 1000 nodes, 3000 links and 100 time intervals: Figure 4.7a shows the speedup curves of PVM-Xolas, PVM-SGI and MT-Xolas implementations of algorithm DOT for a network of 1000 nodes, 3000 links and 100 time intervals. In Figure 4.7a, we note that PVM-SGI implementation shows little speedup. This is due to high communication time on a distributed memory platform. PVM-Xolas implementation, however, shows a good speedup. This is due to the faster communication speed on a Xolas system and also to the interleaving approach used for communication of results. As expected, the MT-Xolas implementation shows significant speedups.

Figure 4.7b shows the burden curves of PVM-Xolas, PVM-SGI and MT-Xolas implementations of algorithm DOT for a network of 1000 nodes, 3000 links and 100 time intervals. The burden measure can be used to estimate the asymptotic speedup of these implementations (see Section 3.5. Figure 4.7b shows the asymptotic speedup of PVM-SGI implementation is $\approx 1/0.25 = 4$. The estimated asymptotic speedup of the PVM-Xolas implementation is $\approx 1/0.01 = 100$.

- **Evaluation of the performance of PVM-Xolas implementation for all-to-many dynamic fastest paths problems using algorithm DOT and destination based decomposition strategy with respect to the network parameters:** We have seen in the previous evaluation that the PVM implementation on a Xolas machine shows significant speedups for a network of 1000 nodes, 3000 links and 100 time intervals. We have also noticed earlier that the execution time of such an implementation is a function of network parameters (see proposition 19). Hence, we would like to see how the speedup of PVM-Xolas implementation changes with the following network parameters: number of nodes, number of links and number of time intervals. This evaluation would help us estimate the asymptotic speedup attainable by a distributed memory implementation on such computer systems for different sizes of net-

works.

- *Number of nodes (n)*: Figure 4.8 shows the variation of speedup and burden with respect to the number of nodes in the network. Figure 4.8a shows that the speedup decreases as the number of nodes increases. Figure 4.8b shows that the burden increases with the number of nodes. Recall that the time required to communicate the results to the master process is in the order of nM. Hence, the time required for communication of results increases with number of nodes. This explains the reduction in speedup and increase in burden with the number of nodes.

- *Number of links (m)*: Figure 4.9 shows the variation of speedup and burden with respect to the number of links in the network. Figure 4.9a may lead us to conclude that the number of links has very little impact on the speedup achieved. We see that the speedup curves are not very sensitive to number of links. Figure 4.9b shows significant differences in the burden values. This figure provides us more insight on performance of this implementation with respect to number of links, which the speedup curves do not provide. Notice that when m is increased from 2000 to 4000, burden decreases from 0.04 to 0.02.

  When m increases, the computation time increases (see Proposition 10). We have noted earlier that the communication time is in the order of nM. It does not depend highly on m. Recall that the burden is proportional to ratio of communication time to the computation time of a parallel implementation. As computation time increases and communication time remains almost the same for this implementation, burden decreases. From the burden values, we can conclude that an estimate of asymptotic speedups of this parallel implementation would be 50 for 4000 links.

- *Number of time intervals (M):* Figure 4.10 shows the variation of speedup and burden with respect to number of time intervals. Figure 4.10a shows that the speedup slightly decreases with the number of time intervals. As

the number of processors we consider is small, we see an almost linear speedup for all time intervals.

Figure 4.10b shows the burden curve for different time intervals. In this figure, we notice that when M is increases from 100 to 300, the burden values increase from 0.02 to 0.035. This is due to the increase in communication time with M. The estimated asymptotic speedup would be 50 for $M = 100$.

- **Comparison of the `PVM-Xolas (without collection of results)`, `PVM-SGI (without the collection of results)`, `MT-Xolas` implementations for many-to-all dynamic fastest path problems using algorithm `DOT` and destination based decomposition strategy** We have noticed in the earlier evaluations that the communication of results back to the master process slows down the PVM implementations. This evaluation concerns the performance of distributed memory implementations without the collection of results by the master process. This comparison can be useful in those network problems, where the results need not be communicated back to the master process. Figure 4.11a and Figure 4.11b show the variation of speedup and burden for `PVM-Xolas (without collection of results)`, `PVM-SGI (without the collection of results)`, `MT-Xolas` implementations of algorithm `DOT` using decomposition by destination, respectively. Figure 4.11a shows that the speedups shown by `PVM-SGI` implementation are smaller than the speedups of `PVM-Xolas` and `MT-Xolas`.This is due to the high communication latency of the distributed network of SGI workstations. `MT-Xolas` implementation shows better speedups than both `PVM-SGI` and `PVM-Xolas` implementations. This is due to the communication requirement in the PVM implementations. Recall that the master process needs to send the network information to all the slave processes in a distributed memory implementation.

  Figure 4.11b shows that the burden for the `PVM-SGI` implementation is approximately 0.1. Hence, the estimated asymptotic speedup of this implementation

is 10. Figure 4.11b also shows that the burden of PVM-Xolas implementation is approximately 0.01. Thus, the estimated asymptotic speedup of this implementation is 100.

From the above evaluations, we conclude that PVM-SGI implementations do not show significant speedups. Hence, these implementations are not considered in further evaluations. With the above evaluations, we have also noted the difference between the PVM implementations with and without the communication of results back to the master process. PVM-Xolas(with collection of results) implementations do not show significant speedups. Hence, in the future evaluations, only PVM-Xolas(without collection of results) implementations are considered. Also note that, from now on, PVM-Xolas implementation will stand for a PVM implementation on the Xolas machine, without the collection of results by the master process.

- **Comparison of the performance of PVM-Xolas and MT-Xolas implementations for many-to-all dynamic fastest paths problems in FIFO networks using algorithm lc-dequeue and origin based decomposition strategy**

  We have noted earlier that both distributed and shared memory implementations of many-to-all dynamic fastest paths problems can be developed by decomposing the set of origins. Algorithm lc-dequeue has been experimentally proved to be the best sequential dynamic shortest path algorithm to compute one-to-all dynamic fastest paths in FIFO networks. Hence, parallel implementations of many-to-all dynamic fastest paths problems in FIFO networks have been developed using this algorithm and a decomposition by set of origins. Figure 4.12 shows the speedup and burden curves for PVM-Xolas and MT-Xolas implementations of many-to-all dynamic fastest paths problems in FIFO networks using algorithm lc-dequeue and origin-based decomposition strategy.

  Figure 4.12a shows that both PVM-Xolas and MT-Xolas implementations show almost linear speedups. Figure 4.12b shows that the burden is low for these

parallel implementations. The asymptotic speedup is estimated to be 1/0.005 = 200.

- **Comparison of the performance of `PVM-Xolas` and `MT-Xolas` implementations for many-to-all dynamic fastest paths problems in non-FIFO networks using algorithm `IOT` and origin based decomposition strategy**

  We have noted earlier that both distributed and shared memory implementations of many-to-all dynamic fastest paths problems can be developed by decomposing the set of origins. Algorithm `IOT` is used to compute the one-to-all dynamic fastest paths in non-FIFO networks. Hence, parallel implementations of many-to-all dynamic fastest paths problems in non-FIFO networks have been developed using this algorithm and decomposition of set of origins. Figure 4.13 shows the speedup and burden curves for `PVM-Xolas` and `MT-Xolas` implementations of many-to-all dynamic fastest paths problems in non-FIFO networks using algorithm `IOT` and origin-based decomposition strategy. Figure 4.13a shows that both `PVM-Xolas` and `MT-Xolas` implementations show almost similar speedups. Figure 4.13b shows that the burden for these implementations is approximately 0.02. Hence, the estimated asymptotic speedup of these implementations is 50.

- **Comparison of the performance of `PVM-Xolas` and `MT-Xolas` implementations for one-to-all dynamic fastest paths problems for many departure time intervals in FIFO networks using algorithm `lc-dequeue` and departure time based decomposition strategy**

  We have noted earlier that both distributed and shared memory implementations of one or many-to-all dynamic fastest paths problems, for many departure time intervals can be developed by decomposing the set of departure times. We have developed `PVM-Xolas` and `MT-Xolas` implementations of one-to-all dynamic fastest paths problems for many departure time intervals in FIFO networks by using algorithm `lc-dequeue` and by decomposing the set of departure

time intervals. Figure 4.14 shows the speedup and burden curves for these implementations. Figure 4.14a shows that speedups of MT-Xolas implementations are smaller than speedups of PVM-Xolas implementations. Proposition 20 shows that the worst case run time complexity of an application level shared memory implementation depends on μ, where μ is the factor indicating the contention of threads. In the MT-Xolas, different threads compute one-to-all dynamic fastest paths for different time intervals, for the same origin. So, the threads may simultaneously require the same network information. This results in contention of threads to read the same memory location. This increases the run time of the MT-Xolas implementation. Thus, we see lesser speedups for this implementation. This is also seen by the very different burden curves shown in Figure 4.14b.

Figure 4.14b shows that the burden for the MT-Xolas implementation is $\approx$ 0.02. Thus, the estimated asymptotic speedup of this implementation is 50. The estimated asymptotic speedup of PVM-Xolas implementation is 200.

- **Comparison of the performance of PVM-Xolas and MT-Xolas implementations for one-to-all dynamic fastest paths problems for many departure time intervals in non-FIFO networks using algorithm IOT and the departure-time based decomposition strategy**

  This evaluation is similar to the previous evaluation. In this case, we do the evaluation for non-FIFO networks using algorithm IOT. Figure 4.15 shows the speedup and burden curves for PVM-Xolas and MT-Xolas implementations of one-to-all dynamic fastest paths problems for many departure time intervals in non-FIFO networks using algorithm IOT and departure-time based decomposition strategy.

  Figure 4.15a shows that MT-Xolas shows significantly smaller speedups that PVM-Xolas implementations. This is due to the same reason described in the previous evaluation. Figure 4.15b shows that the burden of MT-Xolas implementation is about 0.15. Hence, the estimated asymptotic speedup of this

implementation is as low as 6.

- **Evaluation of the performance of all the parallel implementations for different algorithms, decomposition strategies combinations with respect to the following network parameters:** In the earlier evaluation, we have compared the performance of `PVM-Xolas` and `MT-Xolas` implementations for different sequential algorithms and decomposition strategies combinations. Propositions 19 and 20 prove that the run time complexity of each parallel implementation depends on many network parameters. Hence, we evaluate each parallel implementation with respect to the following network parameters: number of nodes, number of links and number of time intervals. The conclusions of these evaluations are not different from earlier evaluations. Hence, we present the speedup and burden curves for all the parallel implementations for all network parameters, but, do not discuss them individually. The following is the list of figures for evaluations with respect to different network parameters.
  - Number of nodes (n) (Figures 4.16, 4.19, 4.22, 4.25, 4.28, 4.31, 4.34, 4.37, 4.40, and 4.43)
  - Number of Links (m) (Figures 4.17, 4.20, 4.23, 4.26, 4.29, 4.32, 4.35, 4.38, 4.41, and 4.44)
  - Number of time intervals (M) (Figures 4.18, 4.21, 4.24, 4.27, 4.30, 4.33, 4.36, 4.39, 4.42, and 4.45)

4.5.2 Conclusions

The experimental evaluation of application level parallel implementations can be summarized as :

- A distributed memory parallel implementation on a network of workstations does not lead to significant speedups. This is due to the high communication latency on a distributed network of workstations.

- Shared memory platforms lead to significant speedups. However, care should be taken so that there is not much contention between the threads (e.g., the decomposition by departure time implementation, see Figure 4.15).

- Asymptotic speedups of the application level parallel implementations on shared memory platforms for network sizes comparable to the ones evaluated varies and is in the range 50-100.

We mentioned in section 4.1 that we consider the following dynamic shortest paths problems, *One-to-all dynamic shortest paths, for one departure time* and *All-to-one dynamic shortest paths, for all departure times* as basic problems. In sections 4.3 and 4.4, we discussed how the applications of these basic problems can be parallelized on both distributed and shared memory platforms. In the rest of this chapter, we describe the various ways in which the solution algorithms of these basic problems can be decomposed. It is important to note that the applications of basic problems are naturally parallel problems. Hence, they require less communication between the subtasks. The algorithm level decomposition requires greater coordination between the subtasks. This coordination increases the communication requirements between the processes in distributed memory environments, which slows down such parallel implementations. We will see that for these decomposition strategies, shared memory platforms perform better than distributed memory platforms.

Not all the decomposition strategies can be studied under the same umbrella as the logic behind coordination of tasks is different for different decomposition strategies. For each of these implementations, we present the notation used in the algorithms, the master process/ thread algorithm and the slave process/ thread algorithm.

4.6 Distributed Memory Implementation of Algorithm DOT by Decomposition of Network Topology

Most shortest paths algorithms would require that the links in a network be processed along the forward star of the origin node or the backward star of the destination node. Algorithm DOT does not require that the links in a network to be processed in any particular order at each time interval, hence, using the network decomposition technique on the algorithm DOT will lead to better speedups, compared to similar decomposition techniques on other dynamic shortest path solution algorithms.

4.6.1 Notation

More notation than the notation described in Section 4.2 is required to describe the master and slave process algorithms. This additional notation is presented next.

Let N be split into N_i, such that $\bigcup_{i=1}^{p} N_i = N$. Let P_i denote the subnetwork to which node i belongs. Hence, if a node $i \in N_j$, then $P_i = j$. Let S_{ij} be the set of links between subnetwork i and subnetwork j, i.e., $S_{ij} = \{(a,b) | a \in N_i, b \in N_j\}$. Let SD_{ij} stand for the set of travel times of the arcs belonging to S_{ij}. Hence $SD_{ij} = \{d_{ab}(t) | (a,b) \in S_{ij}, 0 \leq t \leq M-1\}$. Therefore, the set S_{ii} contains the arcs in the subnetwork of slave process i. Let $\pi_i(t)$ denote the minimum travel time from node i to the destination node q departing at time t. Let S_{ij}^{from} denote the set of start nodes of all the arcs belonging to set S_{ij}. Hence, $S_{ij}^{from} = \{a | (a,b) \in S_{ij}\}$. Similarly, let S_{ij}^{to} denote the set of end-nodes of all the arcs belonging to set S_{ij}, hence, $S_{ij}^{to} = \{b | (a,b) \in S_{ij}\}$. We can see that $S_{ij}^{from} \subset N_i$ and $S_{ij}^{to} \subset N_j$.

4.6.2 Master process algorithm

The algorithm used for computing the fastest paths from all the nodes to a given destination by the master process is:

Master Process

1. Read the network $G(N, A, C, D)$.

2. Compute $\pi_i(M-1) = StaticShortestPath(N, A, D), \forall i \in N$.

3. Divide N into p subsets N_i, $\forall i \in P$.

4. For all $(i, j) \in A$

   set $l = P_i$ *and* $m = P_j$

   $S_{lm} = S_{lm} \cup (i, j)$

4. Spawn p slave processes.

6. For all $i \in P$

   Send S_{ij} to slave process i for all $j \in P$.

   Send S_{ji} to slave process i for all $j \in P$.

   Send D_{ij} to slave process i for all $j \in P$.

   Send to slave process i, destination q

   Send to slave process i, $\pi_s(M-1)$, $\forall s \in N_i$

7. For all $i \in P$

   Receive $\pi_j(t), \forall j \in N_i$, $\forall t$, $0 \leq t \leq M-2$

8. Broadcast the message "quit" to all slave processes.

9. Stop.

4.6.3 Slave process algorithm

The algorithm used for computing the fastest paths from all the nodes to the destination by a slave process is:

Slave Process (DOT-Fastest Paths)

Let the Slave Process id be i, $i \in P$

1. For all $j \in P$

   Receive S_{ij}, S_{ji} and D_{ij} from the master process MP.

2. Receive destination q from the master process MP

3. Receive $\pi_s(M-1)$ *for all* $s \in N_i$ from the master process MP

4. Set $\pi_s(t) = \infty \ \forall t, 0 \leqslant t < M-1, s \in N_i - \{q\}$

5. For t = $M-2$ downto 0

   5.1 For all $(a,b) \in S_{ii}$

      Set $\tau = t + d_{ab}(t)$

      $\tau = min(\tau, M-1)$

      $\pi_a(t) = min(\pi_a(t), d_{ab}(t) + \pi_b(\tau))$

   5.2 For all $j \in P, j \neq i$

      Send $\pi_a(t)$, $\forall a \in S^{to}_{ji}$ to process j

   5.3 For all $j \in P, j \neq i$

      Receive $\pi_a(t)$, $\forall a \in S^{to}_{ij}$, from process j

      For all $(a,b) \in S_{ij}$

         Set $\tau = t + d_{ab}(t)$

         $\tau = min(\tau, M-1)$

         $\pi_a(t) = min(\pi_a(t), d_{ab}(t) + \pi_b(\tau))$

6. Send $\pi_a(t), \forall a \in N_i$, $\forall t$, $0 \leq t \leq M-2$

7. Wait for message "quit" from MP.

8. Stop.

The performance of the above algorithms depends on how the network is decomposed. We use the graph partitioning software package called METIS developed at University of Minnesota to decompose the network. For more information on this package, please refer to the URL *http://www-users.cs.umn.edu/~karypis/metis/metis.html.* This software is used to partition the network with n nodes into p parts, each of them having equal number of nodes.

Following are statistics of results obtained using METIS. To decompose a network of 3000 nodes and 9000 arcs into 2 parts. The first part has 3504 links and the second part has 3518 links. There are 1020 links going from sub network 1 to subnetwork 2, and 958 going from subnetwork 2 to subnetwork 1. For the algorithms above, at every time interval, the slave process needs to send and receive information about the boundary links from or to all the other processes. Hence, in the above example, at every time interval, process 1 needs to send labels of 958 nodes to process 2 and receive information about 1020 nodes from process 2, and vice-versa for process 2. Moreover, the information about the network needs to be sent by the master process to all the slave processes and all the slave processes need to send the computed labels back to the master process. Hence, a great amount of communication is required between the slave processes and the master process. This makes the distributed memory implementation slow. It may even be slower than sequential computation. Note that an MT shared memory implementation would suffer less from this coordination requirement.

4.6.4 Run Time Analysis

Let us by c denote the average time taken to communicate one unit of data between two processors. Then, the worst case run time complexity of the parallel implementation presented in the previous section is given by the following proposition.

Proposition 21 *The worst case run time complexity of the distributed memory im-*

plementation of algorithm DOT by decomposition of network topology is given by:

$$O(mM * c + O(SSP) + (\frac{n}{p} + \frac{m}{p} + \frac{n * c * (p-1)}{p} + n * c) * M), \tag{4.3}$$

where O(SSP) is the worst case run time complexity of the static shortest path algorithm used.

Proof: The master process should communicate the subnetwork to each slave processor and compute static shortest paths for time interval $t = M - 1$ ($O(SSP)$). The total amount of information to be sent to all the processors is in $O(mM)$. From the definition of c, the time taken to communicate the network information is $O(mM * c)$. The slave process should initialize labels ($\frac{nM}{p}$), process $\frac{m}{p}$ links, communicate the labels of the boundary nodes ($\frac{n*c}{p}$) to other processors($p-1$) for each time interval and send the labels of its nodes for all time intervals ($\frac{nM}{p} * c * p$). Hence, the worst case run time complexity of the distributed memory implementation is $O(mM * c + O(SSP) + (\frac{n}{p} + \frac{m}{p} + \frac{n*c*(p-1)}{p} + n * c) * M)$. □

4.7 Shared Memory Implementation of Algorithm DOT by Decomposition of Network Topology

In the shared memory implementations, the network is stored in one global memory. Hence, the shared memory implementations lead to a significant speedup of the algorithm DOT.

4.7.1 Notation

We use the same notation as used for the distributed memory implementation (see Section 4.6.1). It is important to note that all the threads need to be synchronized for every time interval. In a decreasing order of time algorithm, the labels of nodes for time interval t should be set before computing labels for time interval lesser than t. For this synchronization, we use a synchronization barrier as described in page 79. Let

us denote by $SYNCRONIZATION_BARRIER(x)$, the function that synchronizes x threads.

4.7.2 Master thread algorithm

The algorithm of the master process is:

Master Thread

1. Read the network $G(N, A, C, D)$.
2. Compute $\pi_i(M-1) = StaticShortestPath(N, A, D)$, $\forall i \in N$.
3. Initialize labels : $\pi_i(t) = \infty$, $\forall i \in N - \{q\}$, $\forall t \in [0, ..., M-2]$, $\pi_q(t) = 0, \forall t, 0 \leq t \leq M-2$
4. Divide N into p subsets N_i, $\forall i \in P$.
5. For all $(i, j) \in A$

   set $l = P_i$, $m = P_j$

   $S_{lm} = S_{lm} \cup (i, j)$
6. Create p slave threads
7. Wait for all the threads to join back.
8. Stop.

4.7.3 Slave thread algorithm

The sequence of steps used by the slave process is :

Slave Thread

```
/* i ∈ P is the slave process */
```

1. For t = $M-2$ downto 0

   1.1 For all $(a, b) \in S_{ii}$

   * Set $\tau = t + d_{ab}(t)$
   * $\tau = min(\tau, M-1)$
   * $\pi_a(t) = min(\pi_a(t), d_{ab}(t) + \pi_b(\tau))$

   1.2 For all $j \in P,\ j \neq i$

   * For all $(a, b) \in S_{ij}$
     - Set $\tau = t + d_{ab}(t)$
     - $\tau = min(\tau, M-1)$
     - $\pi_a(t) = min(\pi_a(t), d_{ab}(t) + \pi_b(\tau))$

   1.3 SYNCHRONIZATION_BARRIER(p)

2. Exit

From the above algorithms, it is clear that the only overhead of parallelization in a shared memory implementation is the synchronization barrier at every time interval. If the load on each thread is balanced, the lost time at the barrier is minimum. Hence, it will be seen that the shared memory implementation shows significant speedups.

4.7.4 Run Time Analysis

Let us denote the average time lost at the synchronization barrier for every time interval by η. Let us denote the percentage of computation time lost due to contention of threads to access the same information from the global memory by μ. Hence, the worst case run time complexity of the above implementation is given by the following proposition.

Proposition 22 *The worst case run time complexity of shared memory implementation of algorithm DOT by decomposition of network topology is:*

$$O(nM + SSP + ((\frac{m}{p} + \eta) * M)(1 + \mu)) \tag{4.4}$$

Proof: The time taken by the master process to initialize labels and to compute static shortest paths for time interval $t = M - 1$ is $O(nM + SSP)$. The time taken by the threads to compute the dynamic shortest path labels is $\frac{m}{p} * M$ as the number of links to be processed by each thread is equal to the $\frac{m}{p}$. The time taken at the synchronization barrier for all the time intervals is $O(\eta * M)$. Then, from the definition of μ, it follows that the time taken to compute dynamic shortest path labels by each thread is $O(((\frac{m}{p} + \eta) * M)(1 + \mu))$. □

Two more points need to be noted about the parallel implementations based on network topology decomposition. First, the static shortest path computation for time interval $t = M - 1$ is not done in parallel. Any parallel implementation of static shortest path algorithms available in the literature can be used for this purpose. Second, an ideal network partitioning algorithm should split the network into balanced subnetworks. That is, for each subnetwork, the sum of number of links within the subnetwork and the number of its boundary outgoing links should be equal. The problem of partitioning the network satisfying this condition is an NP-Hard problem. Hence, we use an existing graph partitioning software (METIS) to decompose the network. Note that any other network partitioning algorithm can be used to decompose the network, and the results obtained may be different from those presented in this chapter.

4.8 Experimental Evaluation of Algorithm Level Parallel Implementations

The distributed and shared memory implementations of the network decomposition strategy have been extensively evaluated. We discuss the evaluation procedure and

the conclusions of this evaluation in this section. We use the same notation for implementations as discussed in the Section 4.5. As the static shortest path computation for the time interval $t = M - 1$ is not done in parallel, the execution time used to derive the performance measures in the following implementations is the total execution time minus the time required for static shortest path computation.

4.8.1 Numerical tests

Following are the evaluations that were conducted:

- **Comparison of the performance of `PVM-SGI`, `PVM-Xolas` and `MT-Xolas` implementations of algorithm DOT using network decomposition for a network of size $n = 1000$, $m = 3000$, $M = 100$:** Figure 4.46 shows the speedup and the burden for all parallel implementations of algorithm `DOT` using network decomposition. Figure 4.46a shows that the distributed memory parallel implementations are slower than the sequential implementations. This is due to the high communication requirements of the distributed memory implementations. Figure 4.46a also shows that `MT-Xolas` implementation leads to good speedups. Figure 4.46b show that burden of `MT-Xolas` implementation is ≈ 0.05. Hence, the estimated asymptotic speedup of this implementation is 20.

- **Evaluation of the performance of the `MT-Xolas` implementation of algorithm DOT using network decomposition, with respect to different network parameters :** Proposition 22 proves that the shared memory implementation of algorithm `DOT` is dependent on the following network parameters: number of nodes, number of links and number of time intervals. We evaluate the performance of the `MT-Xolas` with respect to these network parameters.

  - *Number of Nodes (n)*: Figure 4.47 shows the speedup and burden curves for the `MT-Xolas` implementation of algorithm `DOT` using network decomposition with respect to the number of nodes. Figure 4.47a shows that the performance of the `MT-Xolas` implementation is not highly sensitive to this parameter.

- *Number of Links (m)*: Figure 4.48 shows the speedup and burden curves for the `MT-Xolas` implementation of algorithm `DOT` using network decomposition. Figure 4.48a shows the speedup decreases slightly with the increase in the number of links.
- *Number of Time Intervals (M)* Figure 4.49 shows the speedup and burden curves for different time intervals. Figure 4.49a shows the speedup decreases with number of time intervals. The time taken for synchronization of threads is given by $\eta * M$, where η is the average time lost at the synchronization barrier for every time interval. Hence, as M increases, this time loss increases. Thus, as M increases, speedup shown by `MT-Xolas` implementation decreases.

4.8.2 Conclusions

The above experimental evaluation can be summarized as :

- Distributed memory implementations on both the network of SGI workstations and a SMP machine are slower than the sequential implementations due to high communication requirements of this decomposition strategy.
- Shared memory implementations lead to a speedup of $\approx$ 4.5 for a network similar to a traffic network, for 6 processors. Hence, this implementation has an asymptotic speedup of about 20 . Only algorithm `DOT` leads to such a high speedup with this decomposition technique.

In the above sections, we have discussed the different ways algorithm `DOT` and its applications may be parallelized. Extensive evaluation of the parallel implementations of algorithm `DOT` shows promising results.

Let us assume that we have as many number of processors as we want. Then, is there an efficient massively parallel implementation of algorithm `DOT`. Hence, we would like to find out the idealized maximum speedup that can be obtained for algorithm `DOT` on a "ideal parallel computer" which will be defined in the next section.

Hence, in the next section, we first define an "ideal parallel computer" and then develop a parallel implementation for algorithm DOT suitable to such a parallel computer.

4.9 Idealized Parallel Implementation of Algorithm DOT

We define an *ideal parallel computer* as a shared memory parallel computer having as many processors as required by the parallel algorithm and with the memory access time as a constant regardless of the number of processors in use. In the next subsection, we discuss the notation used to describe the parallel DOT algorithm.

4.9.1 Notation

In describing the parallel algorithm designed for this ideal parallel computer, we use statements of the general form given in expression 4.5

$$\texttt{for x} \in \texttt{S in parallel do statement(x)} \qquad (4.5)$$

This parallel statement means that we assign a processor to each element x of set S, and then carry out in parallel the instructions in *statement*(x) for every such element, using x as the data.

The ideally parallel DOT algorithm is referred to as `DOT-parallel`. In `DOT-parallel`, we use a technique similar to the network decomposition technique. We use m processors to run the main loop of algorithm DOT. Each link in the network is allotted to one processor. We use nM processors to initialize the labels for all nodes at all time intervals. We assume a parallel static shortest path algorithm called $StaticShortestParallel(\pi, q)$ which returns the all-to-one static shortest paths in the minimum time possible. The input π to this algorithm denotes the link costs while q denotes the destination.

4.9.2 Algorithm DOT-parallel

The algorithm DOT-parallel is:

Step 0 (Initialization):

$$\begin{aligned}
&\forall(\mathtt{i} \neq \mathtt{q})\ \forall(\mathtt{t} < \mathtt{M}-1)\ \mathtt{in\ parallel\ do}\ \pi_{\mathtt{i}}(\mathtt{t}) = \infty \\
&\forall(\mathtt{t} < \mathtt{M}-1)\ \mathtt{in\ parallel\ do}\ \pi_{\mathtt{q}}(\mathtt{t}) = 0 \\
&\pi_{\mathtt{i}}(\mathtt{M}-1) = \mathtt{StaticShortest}(\mathtt{d}_{\mathtt{ij}}(\mathtt{M}-1), \mathtt{q}, \mathtt{p}) \quad \forall \mathtt{i}
\end{aligned} \tag{4.6}$$

Step 1 (Main Loop):

$$\begin{aligned}
&\mathtt{For\ t} = \mathtt{M}-2\ \mathtt{down\ to\ 0\ do} \\
&\quad \mathtt{For\ (i,j)} \in \mathtt{A\ in\ parallel\ do} \\
&\quad\quad \phi_{\mathtt{ij}} = \mathtt{d}_{\mathtt{ij}}(\mathtt{t}) + \pi_{\mathtt{j}}(\mathtt{t} + \mathtt{d}_{\mathtt{ij}}(\mathtt{t})) \\
&\quad \mathtt{For\ i} \in \mathtt{N\ in\ parallel\ do} \\
&\quad\quad \pi_{\mathtt{i}}(\mathtt{t}) = \min_{\mathtt{j} \in \mathtt{A(i)}} \phi_{\mathtt{ij}}
\end{aligned} \tag{4.7}$$

In algorithm DOT-parallel, the maximum number of processors required at any instant of time is nM (to initialize the labels of all nodes for all time intervals). All threads first compute label $(d_{ij}(t)+\pi_j(t+d_{ij}(t)))$ and store it in a temporary variable, ϕ_{ij}. Then, the minimum label for all the nodes is computed in parallel.

Brassard and Bratley [5] show that the minimum of n numbers can be computed using $\log n$ operations. They use a maximum of $n/2$ processors and a binary tree representation to compute the minimum. For completeness, we describe this procedure to compute minimum in the next section. The same procedure can be used to compute the minimum label of each node in algorithm DOT-parallel.

4.9.3 Computing the minimum using a complete binary tree

This technique to compute the minimum is illustrated by an example. Suppose we want to compute the minimum of n integers. Assume that n is a power of 2. If it is not, the required number of very large integers can be added to the data set. The n

elements are placed at the leaves of a complete binary tree, as illustrated in Figure 4.6. In the first step, the minimum of the elements lying beneath each internal node at level 1 are calculated in parallel. In the second step, the minimum of elements lying beneath each internal node at level 2 are calculated in parallel; and so on, until at the $(\log n)^{th}$ step, the value obtained at the root gives the minimum of all n integers.

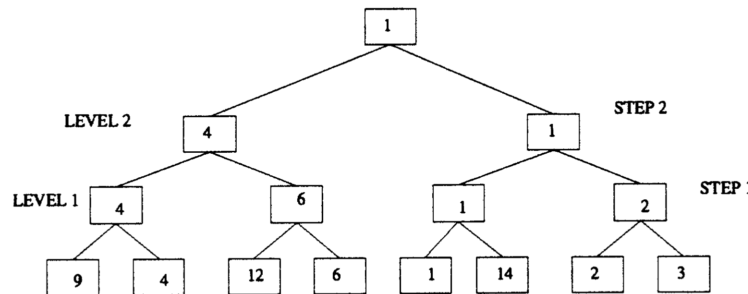


Figure 4.6: Computing Minimum with a Complete Binary Tree

4.9.4 Run time analysis of algorithm `DOT-parallel`

We use the binary tree technique described in the previous subsection to compute the minimum for each node at every time interval in algorithm `DOT-parallel`. The worst case run time complexity of algorithm `DOT-parallel` is given by the following proposition.

Proposition 23 *The worst case run time complexity of algorithm DOT-parallel is $O(PSSP + M * \log k)$, where $PSSP$ is the best possible worst case run time complexity of a parallel static shortest path algorithm and k is the maximum out-degree of the network (i.e., the out-degree of a node if the number of outgoing links of a node).*

Proof: The complexity can be computed in a straightforward manner by counting the number of operations. As the initialization of labels is done in parallel using nM processors, this step takes constant time. By the definition of $PSSP$, the static shortest path computation for time interval $t = M - 1$ is in $O(PSSP)$. In the main loop, computing the labels (ϕ_{ij}) requires constant time as m processors are

used on a ideal parallel computer. The maximum number of operations required to computes the minimum for each node is $\log(\max_{i \in N}(|A(i)|))$. Recall that we use the procedure described in Section 4.9.3 to compute the minimum. The maximum number of outgoing links of a node in the network is in the order of the maximum out-degree of the network (k). Hence, the complexity of the main loop is $O(M * \log k)$. Thus, the worst time complexity of algorithm `DOT-parallel` is $O(PSSP + M * \log k)$.

□

Note that the maximum value of k is $n - 1$ as the maximum possible number of links in the network is $n(n-1)$. Thus, the order of algorithm `DOT-parallel` is $O(PSSP + M * \log n)$.

For a traffic network, the maximum degree is usually 3. Hence, the worst case run time complexity of algorithm `DOT-parallel` for such networks will be $O(PSSP + M)$. The idealized maximum speedup for algorithm `DOT` is given by the following proposition.

Proposition 24 *The algorithm DOT-parallel is approximately $\frac{m}{\log k}$ times faster that algorithm DOT. Hence, the idealized maximum speedup of algorithm DOT is $\frac{m}{\log k}$.*

Proof: The worst case run time complexity of algorithm `DOT` is $O(nM + mM + SSP)$, where SSP is the worst case run time complexity of the static shortest path algorithm (see Proposition 10). In Proposition 23, we proved that the worst time complexity of algorithm `DOT-parallel` is $O(PSSP + M * \log k)$, where $PSSP$ is the order of best parallel static shortest path algorithm. Assuming that the time required to compute the static shortest paths is an order of magnitude lesser than the computation time of the main loop in algorithm `DOT` and `DOT-parallel`, we can see that algorithm `DOT-parallel` is $\frac{m}{\log k}$ times faster than the sequential algorithm `DOT`. Hence, the speedup of algorithm `DOT-parallel` is $\frac{m}{\log k}$. □

We have noted earlier that a data structure used in the solution algorithm of the problem can be used as a decomposition dimension. In the section, we describe a way of parallelizing algorithm `IOT` by decomposing the buckets used at each time interval in algorithm `IOT`.

4.10 Shared Memory Implementation of Algorithm IOT by Decomposition of Data Structure

Algorithm IOT is used to compute one-to-all dynamic fastest paths for one departure time interval(see Section 2.5.2). Recall that in algorithm IOT, we maintain a list of nodes that have been reached for each time interval and process only those nodes at each time interval. These lists of nodes were called *buckets*. In this parallel implementation, we decompose the computation for buckets. In the sequential implementation, one node from the bucket is removed and processed. In the parallel implementation using p processors, at most p nodes are removed and processed simultaneously. The processing of nodes requires synchronization between the processes. In this section, we present the master thread and slave thread algorithms for this parallelization strategy. Parallel implementation of this algorithm is under development at the time of writing of this thesis., We do not discuss its experimental evaluation in this thesis.

4.10.1 Notation

We use the same notation used to describe algorithm IOT in Section 2.5.2. Let x denote a mutex lock(see Section 3.1 for information on mutex locks). Then, we assume that function $lock(x)$ obtains a mutex lock x. Also, let function $unlock(x)$ unlock the mutex lock x. The thread that obtains the lock x first is the sole owner of the code segment between the calls to functions $lock(x)$ and $unlock(x)$. Another thread requesting the lock will be made to sleep till the first thread unlocks the lock x. One lock is associated with each global variable. Hence, if a global array variable X has x elements, lock $X(i)$ denotes the lock associated with the ith element of X. Thus, if one thread tries to change the variable $X(i)$, it should request the mutex lock $X(i)$ using the function call $lock(X(i))$.

Note that in algorithm IOT, all the threads should complete computation for one time interval t before proceeding to the next time interval. We use a synchronization barrier to synchronize all the threads at the end of every time interval. Thus, we as-

sume a function *SYNCHRONIZATION_BARRIER(x)* that does this syncronization. This function is similar to the one described in Section 4.7.1. The variable x denotes the number of threads that need to be synchronized at the barrier.

The decomposition dimension in this implementation is the bucket $Q(t)$. Hence, let us assume that we have a function $i^{th}PartOf(X, i, p)$, which runs algorithm DECOMP(see Section 4.2) to decompose the set X into p parts and returns its ith part.

The master thread and the slave thread algorithms are described using this notation in the next two subsections.

4.10.2 Master thread algorithm

In this parallel implementation, the master thread needs to initialize variables, create slave threads, wait for the slave threads to return and compute static shortest paths for time interval $t = M - 1$, if necessary. The master thread algorithm is:

Master Thread Step 0 (Initialization):

$$f_i = \infty, \quad \forall i \in N - \{s\}$$
$$w_i(t) = \infty, \quad \forall i \in N \text{ and } \quad t_o \leqslant t \leqslant M$$
$$w_s(t_o) = 0$$
$$f_s = 0$$
$$Q(t_o) = \{s\}$$
$$S = \{s\}$$

Step 1 (Creating and Waiting for Threads):

Create p slave threads

Wait for p threads to join back

Step 2 (Static Shortest Path Computation for $t \geqslant M - 1$):

$$\text{if}(|S| < n)$$
$$\quad \sigma = SSP(G, d(M-1), w_i(M-1), s)$$
$$\quad \text{for all } i \notin S$$
$$\quad\quad f_i = \sigma_i$$

4.10.3 Slave thread algorithm

The slave thread starts running the function *IOTthread*(i), where i is the id number of the slave thread. The slave thread algorithm is:

```
/* i ∈ P be a slave process */

 For  t = t₀...M − 1
   Qᵢ(t) = iᵗʰPartOf(Q(t), i)
   For  all  i ∈ Qᵢ(t)
     if  (i ∉ S)  then
       lock(S)
       S = S ∪ {i}
       unlock(S)
     if  (|S| = n)  then
       Wake threads sleeping at the SYNCHRONIZATION BARRIER
       RETURN.
     For  all  j ∈ A(i)
       τ = min(M − 1,  t + dᵢⱼ(t))                                    (4.8)
       lock(wⱼ(τ))
       if  (wⱼ(τ) = ∞)  then
         lock(Q(τ))
         Q(τ) = Q(τ) ∪ {j}
         unlock(Q(τ))
         wⱼ(τ) = τ − t₀
         lock(fⱼ)
         fⱼ = min(fⱼ, τ)
         unlock(fⱼ)
       unlock(wⱼ(τ))
   if(|S| < n)
     SYNCHRONIZATION_BARRIER(p)
```

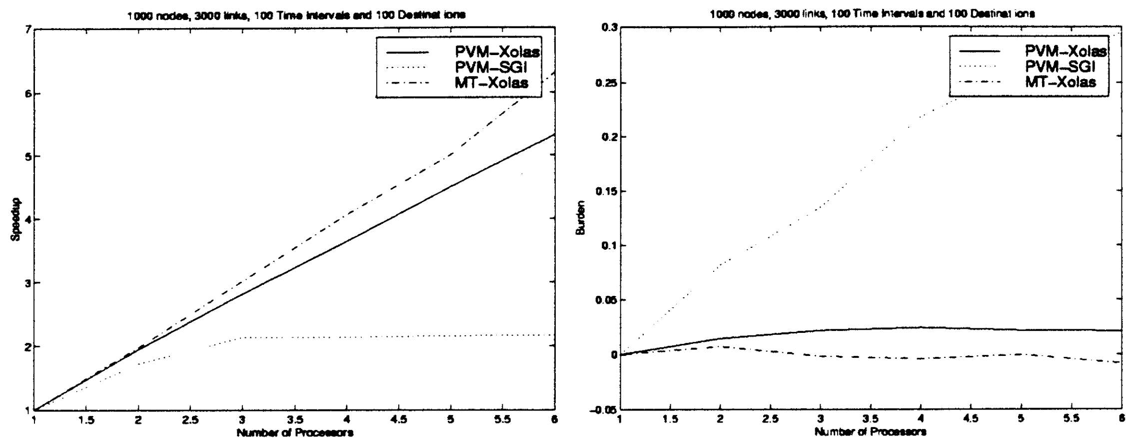
The speedup of the above algorithm is highly dependent on the number of elements present in the bucket at any time interval. This number depends on the network topology and also on the values of link travel times. As noted earlier, this algorithm is still under development. Hence, an evaluation of this algorithm for test networks

is not presented in this thesis.

4.11 Summary

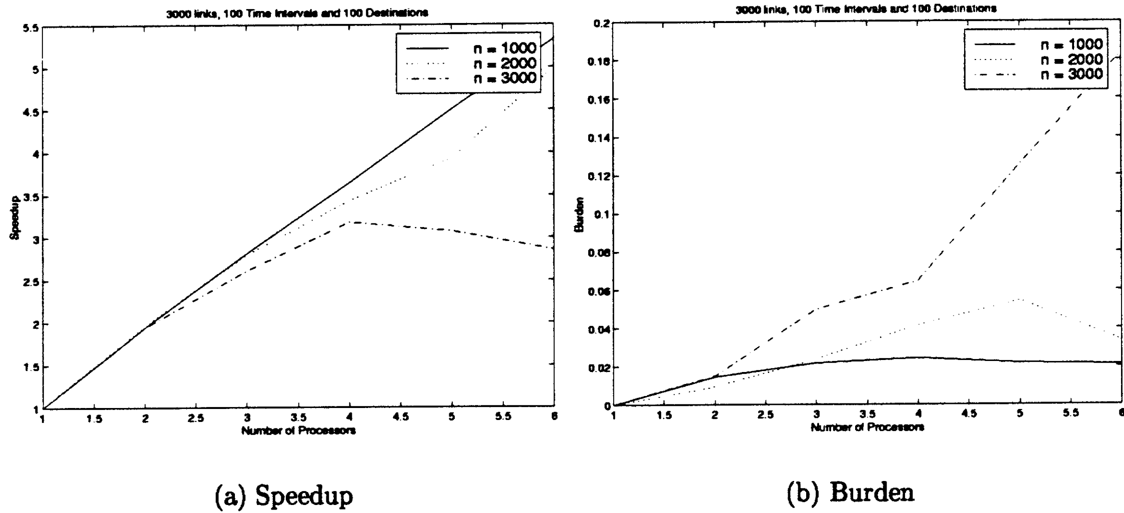
The discussion in this chapter can be summarized as:

- Dynamic shortest path problems may be decomposed for parallel implementatios along five dimensions: (1) destination, (2) origin, (3) departure time interval, (4) network topology and (5) data structure used in the solution algorithm of the problem. We identified two levels of parallel implementations:
  - Application level: Destination, origin and departure time are the application level decomposition strategies.
  - Algorithm level: Network topology and data structure are algorithm level parallel implementations.
- Both applications level and algorithms level parallel implementations for algorithms `ls-heap`, `lc-dequeue`, `dial-buckets`, `IOT` and `DOT` are developed for both shared and distributed memory platforms.
- Extensive testing of the parallel implementations on random networks indicate that shared memory platforms lead to significant speedup of optimal sequential dynamic shortest path algorithms.
- The shared memory parallel implementation of algorithm `DOT` using network decomposition leads to better speedups than the parallel implementations of other dynamic shortest path algorithms using the same decomposition strategy.
- The idealized maximum speedup of algorithm `DOT` is proved to be approximately equal to $\frac{m}{\log k}$, where m is the number of links and k is the maximum out-degree of a node in the network.



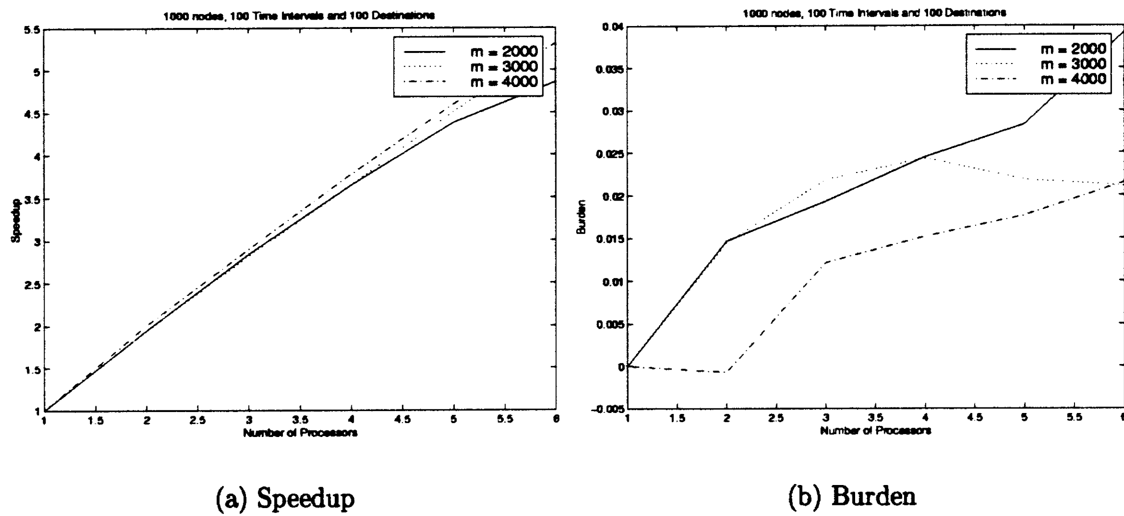
(a) Speedup (b) Burden

Figure 4.7: Decomposition by Destination(with collection of results by the master process in PVM implementations)



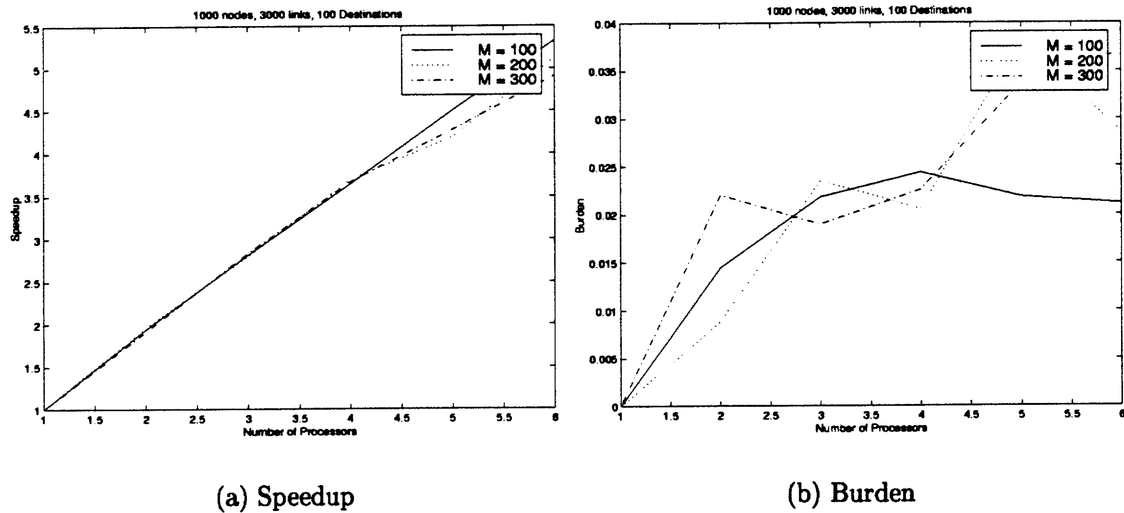
(a) Speedup (b) Burden

Figure 4.8: Performance of (PVM, Destination, DOT Implementation: varying number of nodes



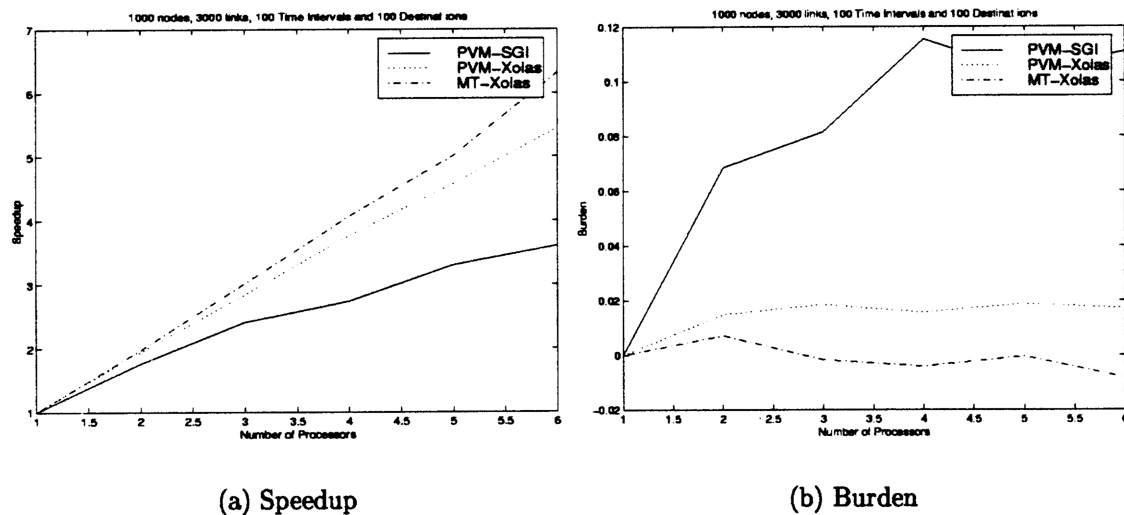
(a) Speedup (b) Burden

Figure 4.9: Performance of (PVM, Destination, DOT Implementation: varying number of links



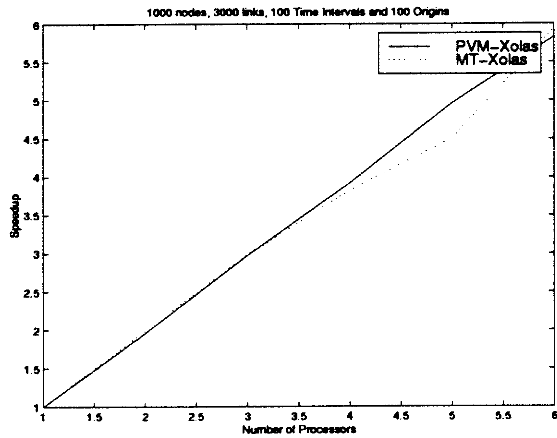
(a) Speedup (b) Burden

Figure 4.10: Performance of (PVM, Destination, DOT Implementation: varying number of time intervals

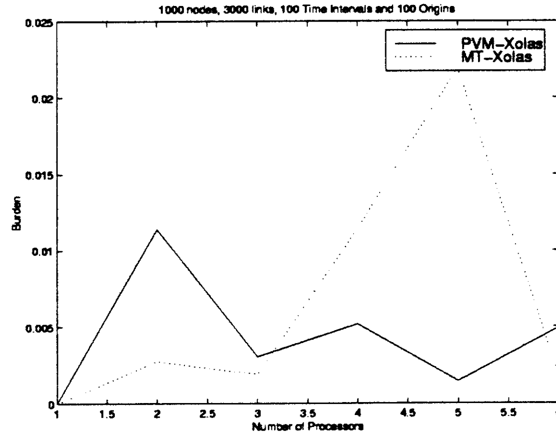


(a) Speedup (b) Burden

Figure 4.11: Decomposition by Destination(without collection of results by the master process in PVM implementations)

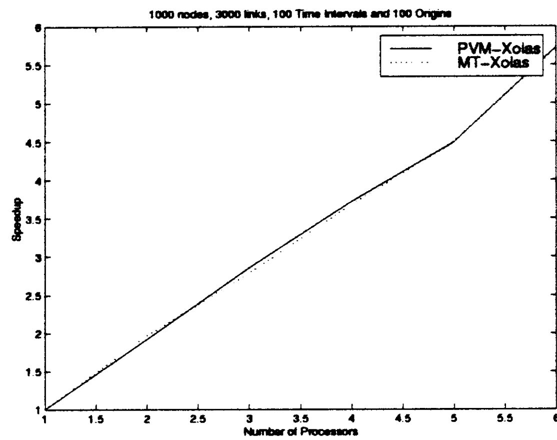


(a) Speedup

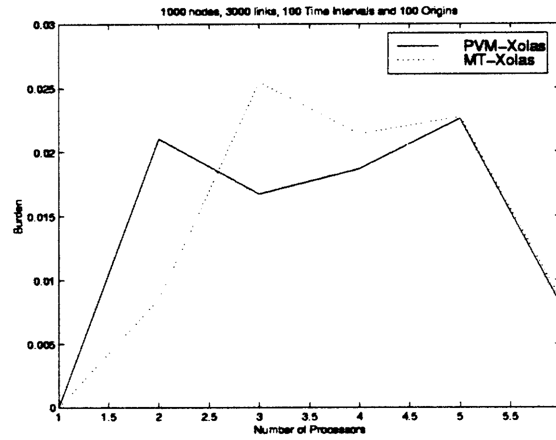


(b) Burden

Figure 4.12: Decomposition by Origin (FIFO networks)

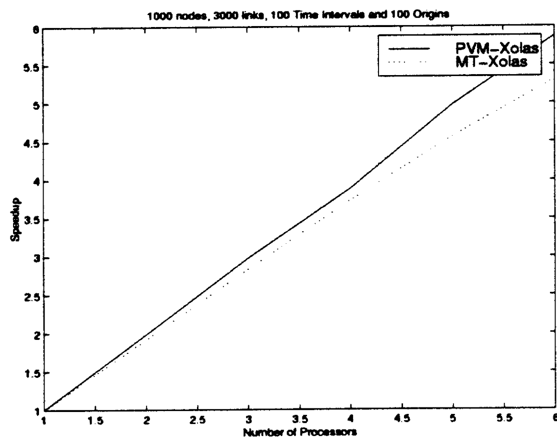


(a) Speedup

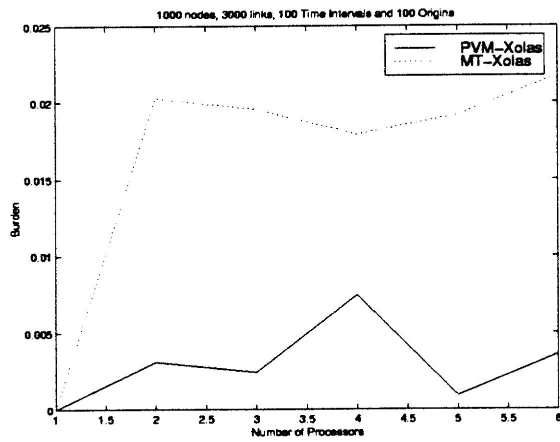


(b) Burden

Figure 4.13: Decomposition by Origin (non-FIFO networks)

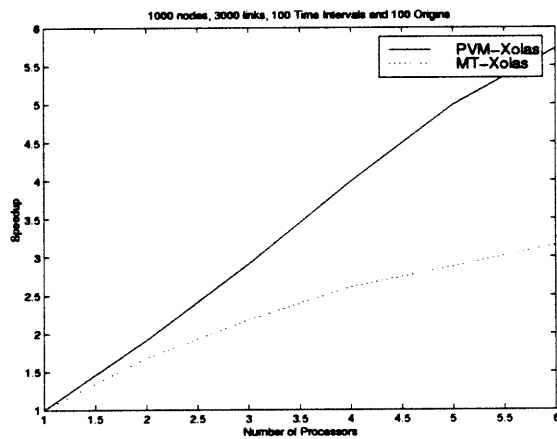


(a) Speedup

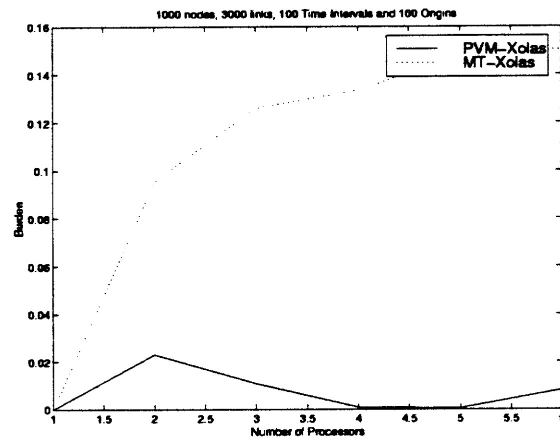


(b) Burden

Figure 4.14: Decomposition by Departure Time (FIFO networks)

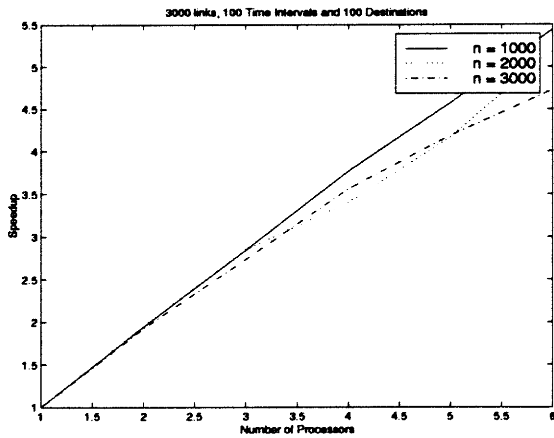


(a) Speedup

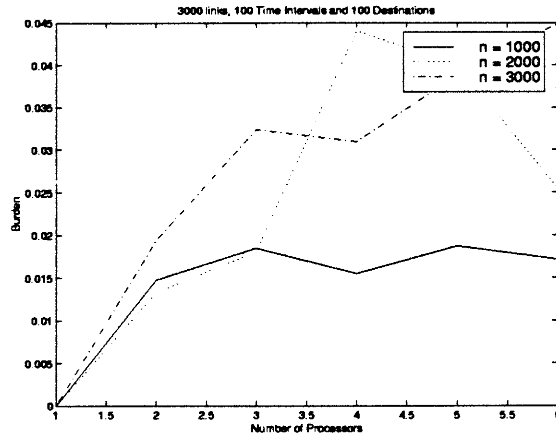


(b) Burden

Figure 4.15: Decomposition by Departure Time (non-FIFO networks)

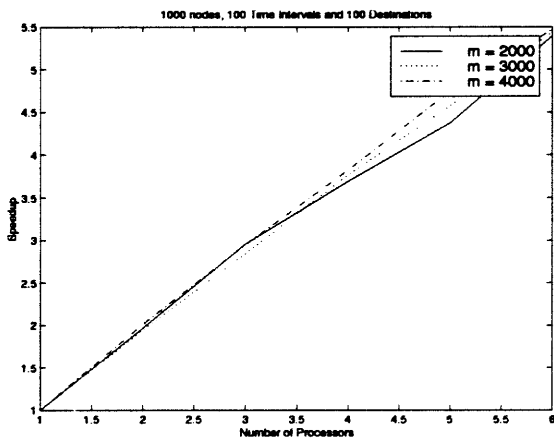


(a) Speedup

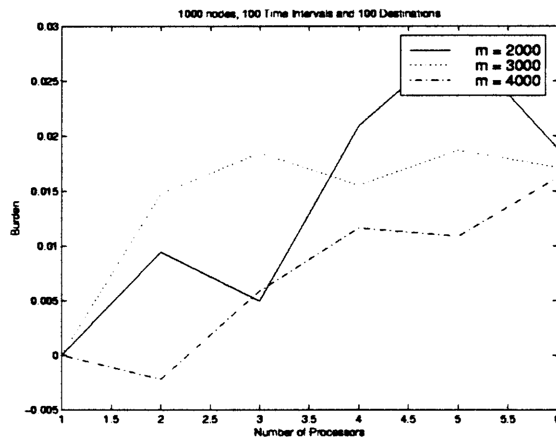


(b) Burden

Figure 4.16: Performance of (PVM (without collection of results by the master process), Destination, DOT) Implementation: varying number of nodes

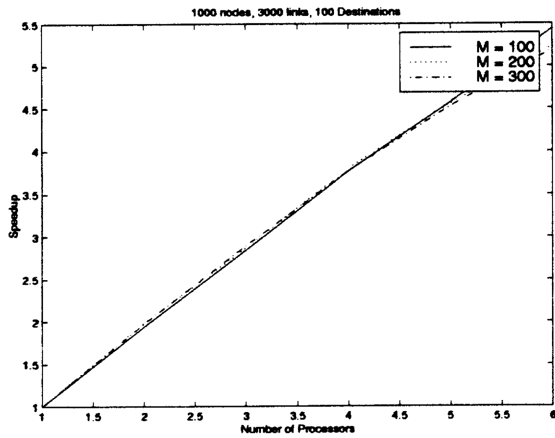


(a) Speedup

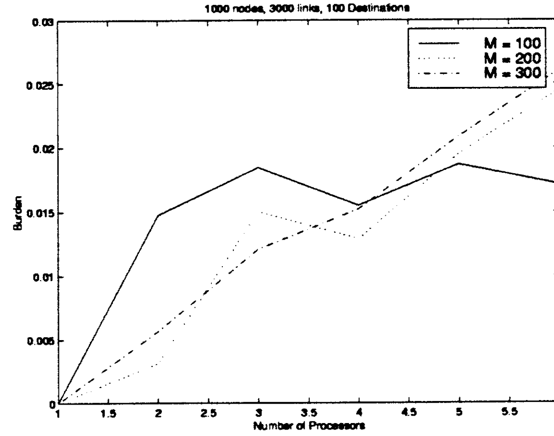


(b) Burden

Figure 4.17: Performance of (PVM (without collection of results by the master process), Destination, DOT) Implementation: varying number of links

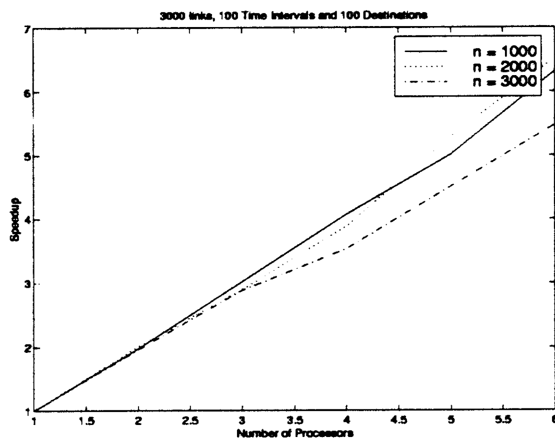


(a) Speedup

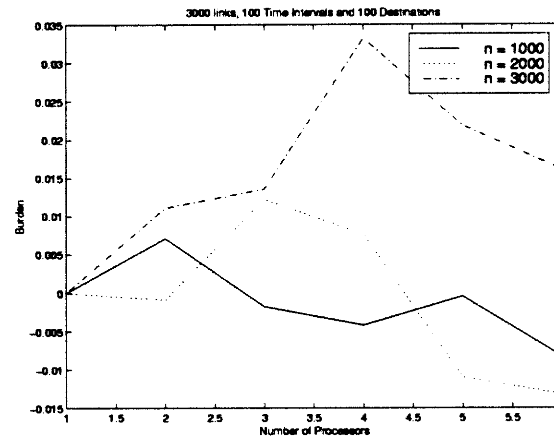


(b) Burden

Figure 4.18: Performance of (PVM (without collection of results by the master process), Destination, DOT) Implementation: varying number of time intervals

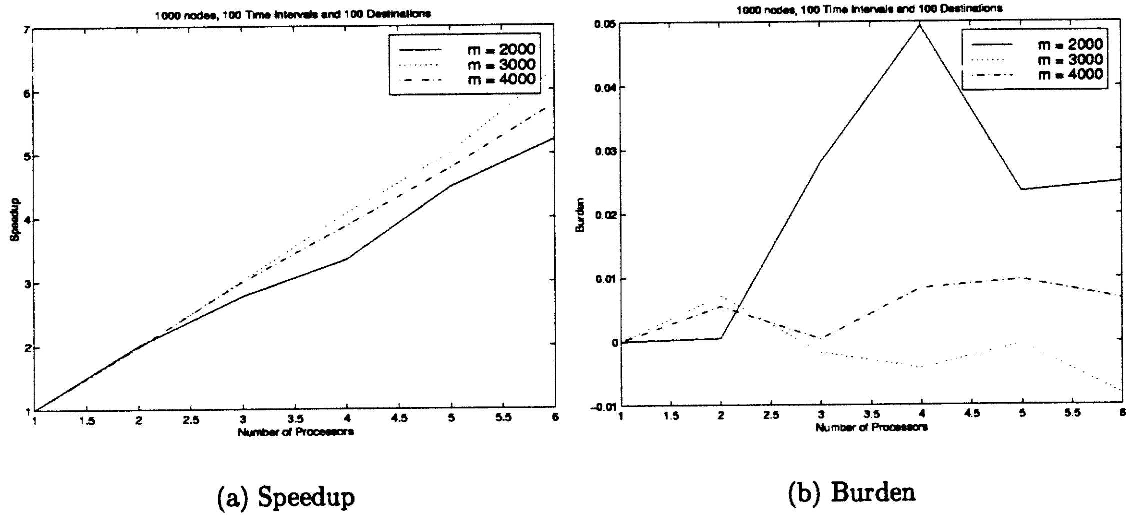


(a) Speedup



(b) Burden

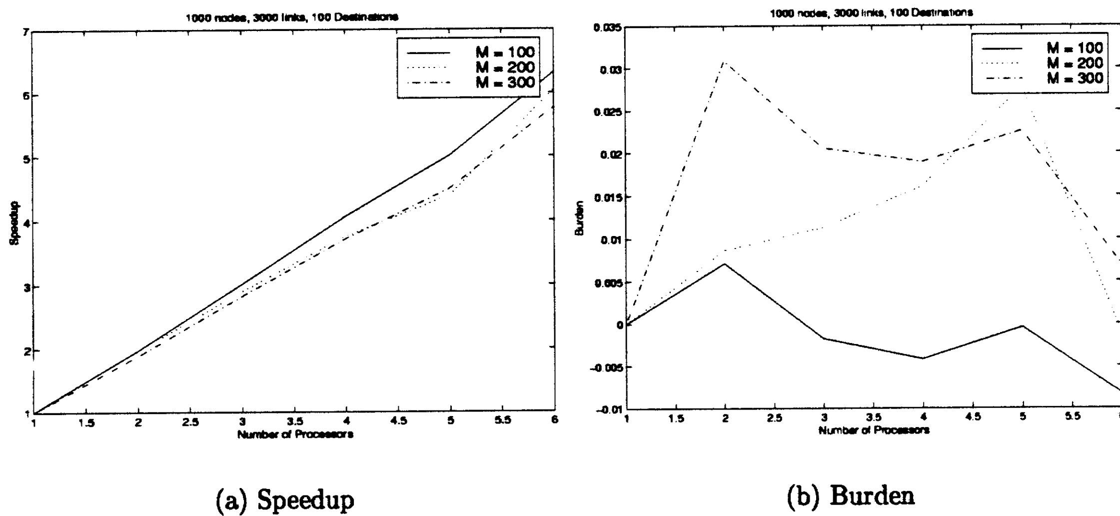
Figure 4.19: Performance of (MT, Destination, DOT) Implementation: varying number of nodes



(a) Speedup

(b) Burden

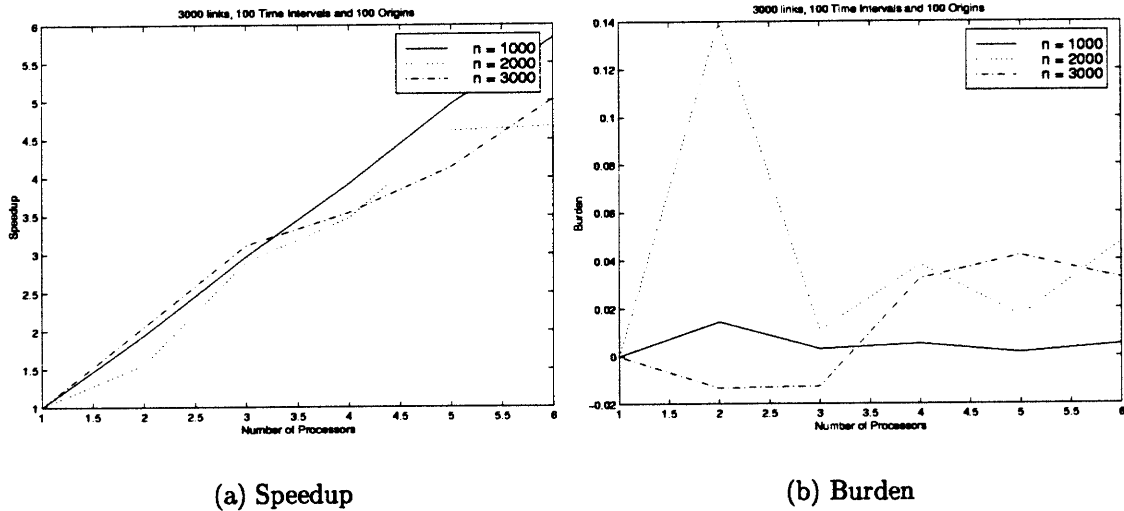
Figure 4.20: Performance of (MT, Destination, DOT) Implementation: varying number of links



(a) Speedup

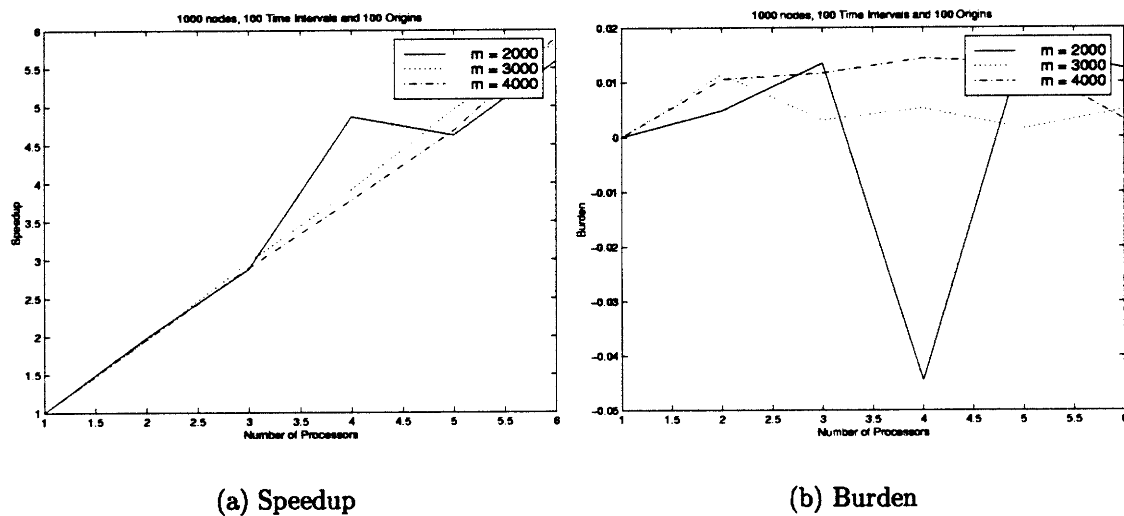
(b) Burden

Figure 4.21: Performance of (MT, Destination, DOT) Implementation: varying number of time intervals



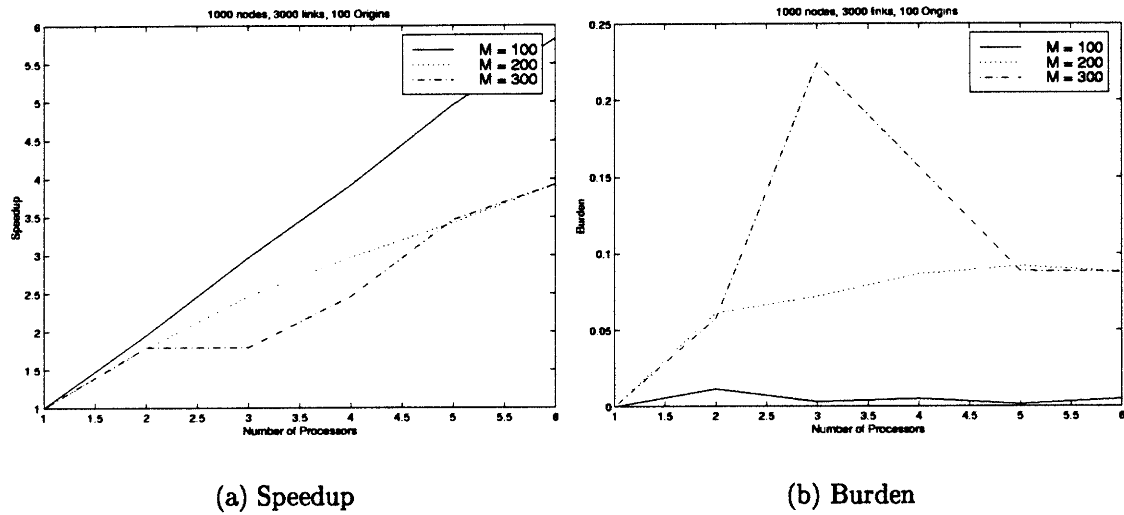
(a) Speedup (b) Burden

Figure 4.22: Performance of (PVM, origin, `lc-dequeue`) Implementation: varying number of nodes



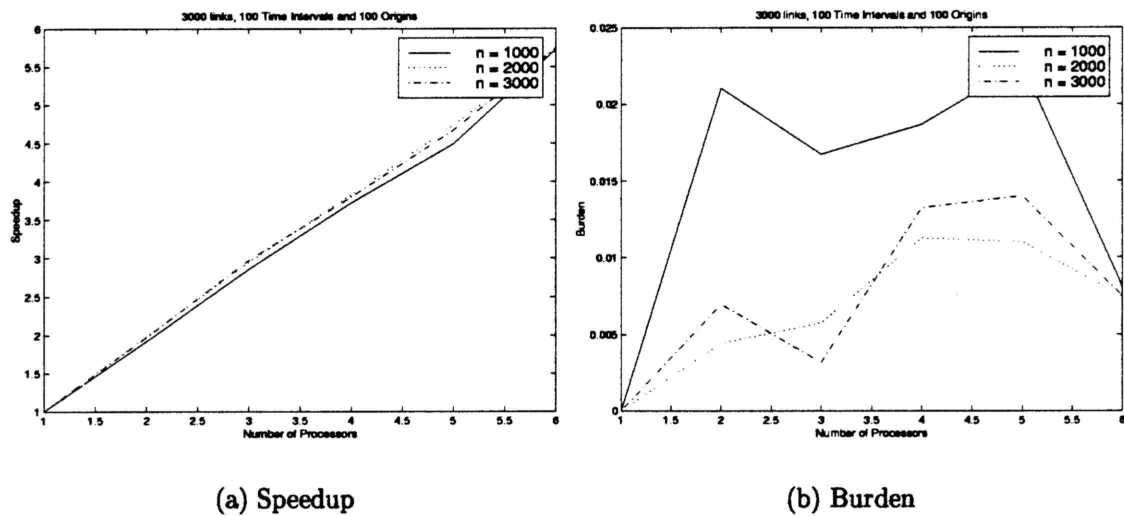
(a) Speedup (b) Burden

Figure 4.23: Performance of (PVM, origin, `lc-dequeue`) Implementation: varying number of links



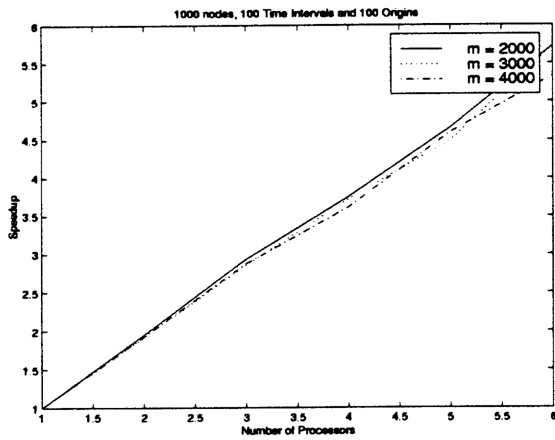
(a) Speedup (b) Burden

Figure 4.24: Performance of (PVM, origin, `lc-dequeue`) Implementation: varying number of time intervals

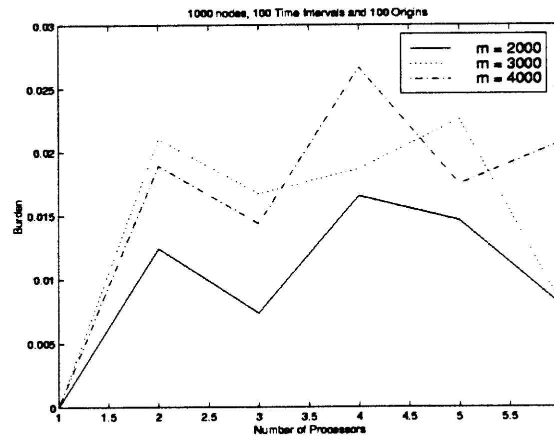


(a) Speedup (b) Burden

Figure 4.25: Performance of (PVM, origin, `IOT`) Implementation: varying number of nodes

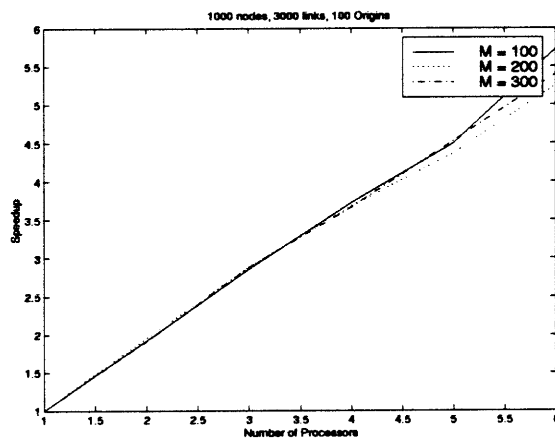


(a) Speedup

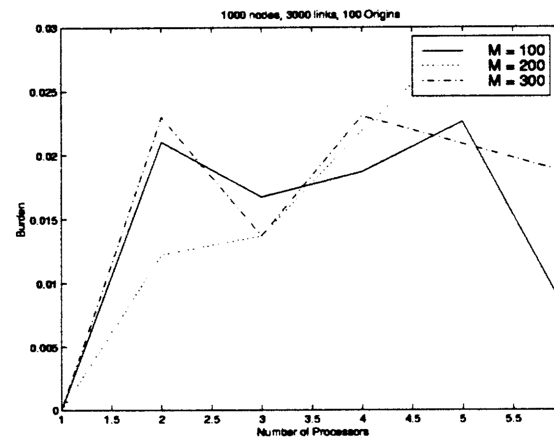


(b) Burden

Figure 4.26: Performance of (PVM, origin, IOT) Implementation: varying number of links

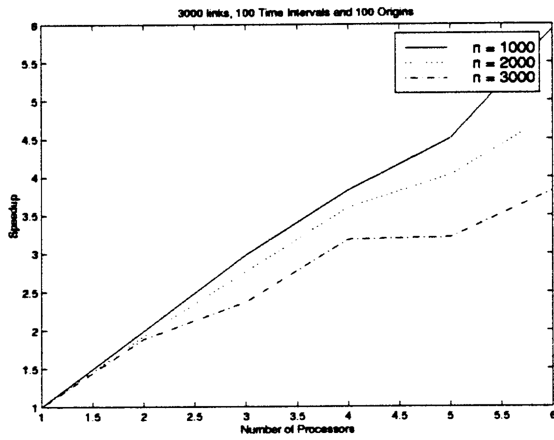


(a) Speedup

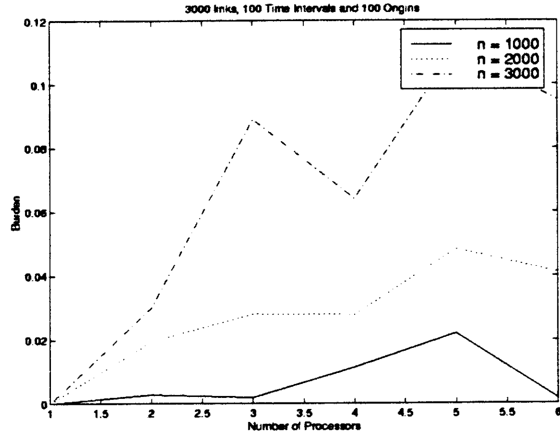


(b) Burden

Figure 4.27: Performance of (PVM, origin, IOT) Implementation: varying number of time intervals

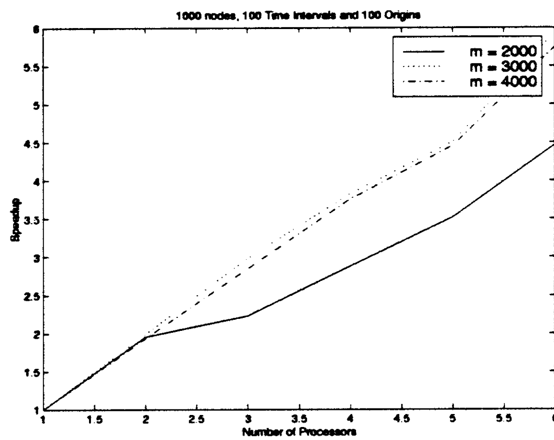


(a) Speedup

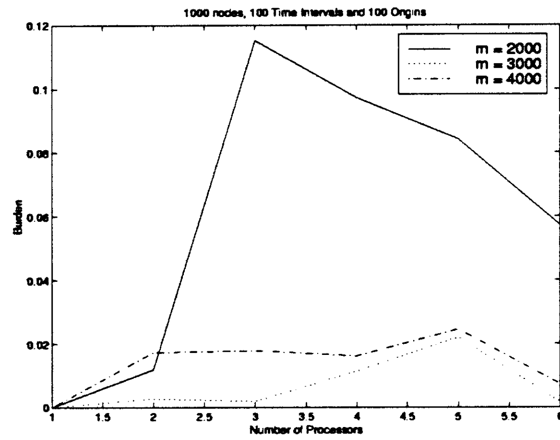


(b) Burden

Figure 4.28: Performance of (MT, origin, `lc-dequeue`) Implementation: varying number of nodes

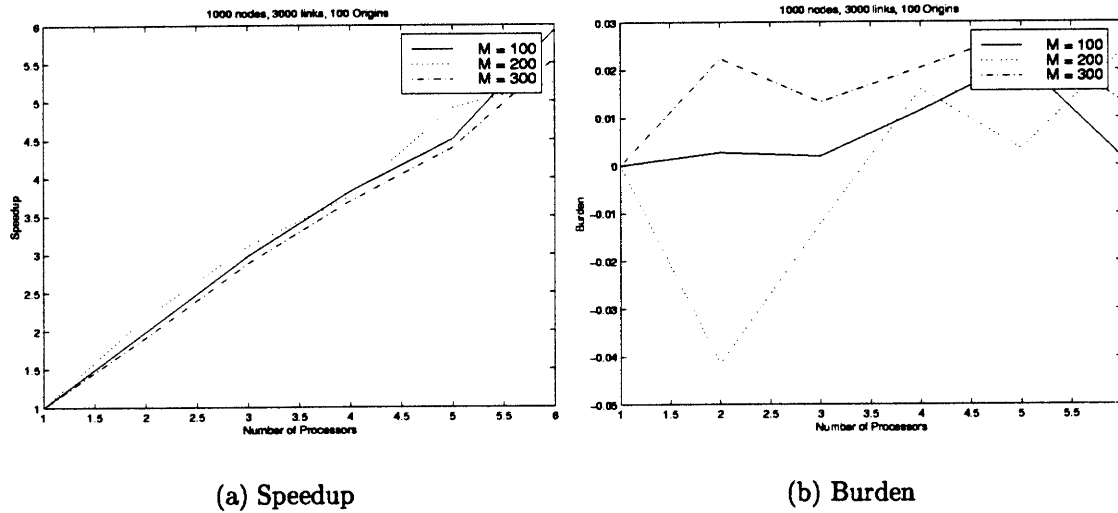


(a) Speedup



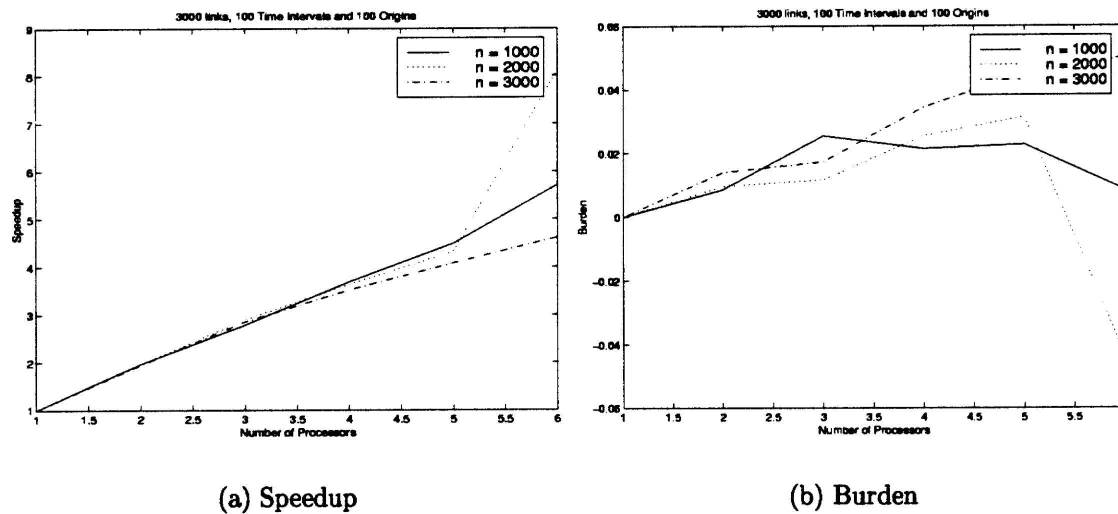
(b) Burden

Figure 4.29: Performance of (MT, origin, `lc-dequeue`) Implementation: varying number of links



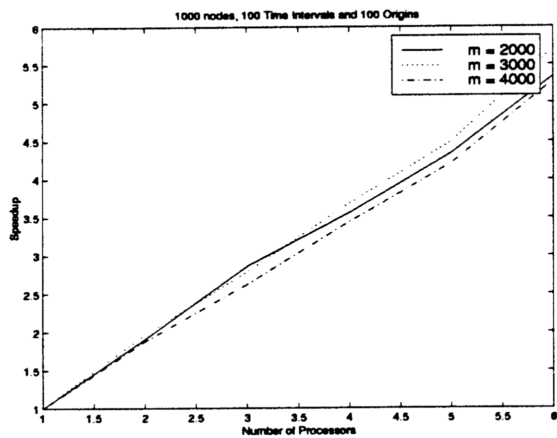
(a) Speedup (b) Burden

Figure 4.30: Performance of (MT, origin, `lc-dequeue`) Implementation: varying number of time intervals

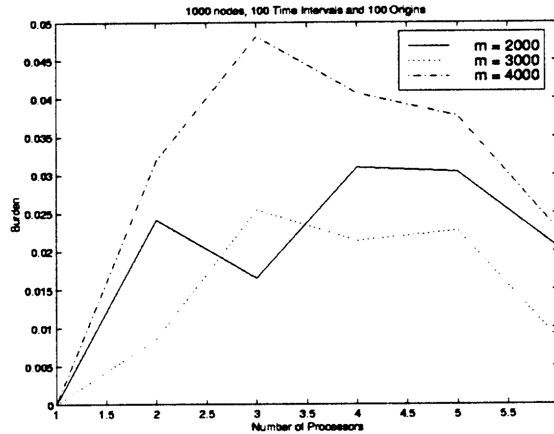


(a) Speedup (b) Burden

Figure 4.31: Performance of (MT, origin, `IOT`) Implementation: varying number of nodes

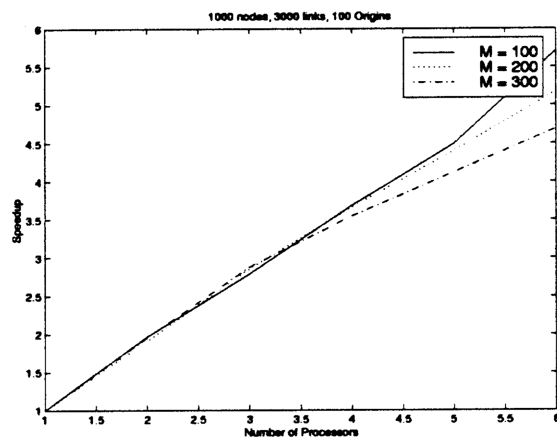


(a) Speedup

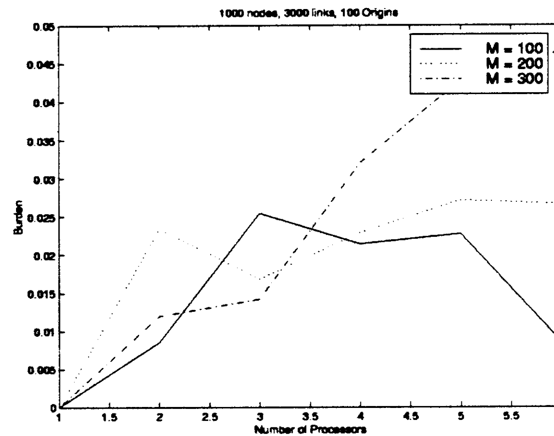


(b) Burden

Figure 4.32: Performance of (MT, origin, IOT) Implementation: varying number of links

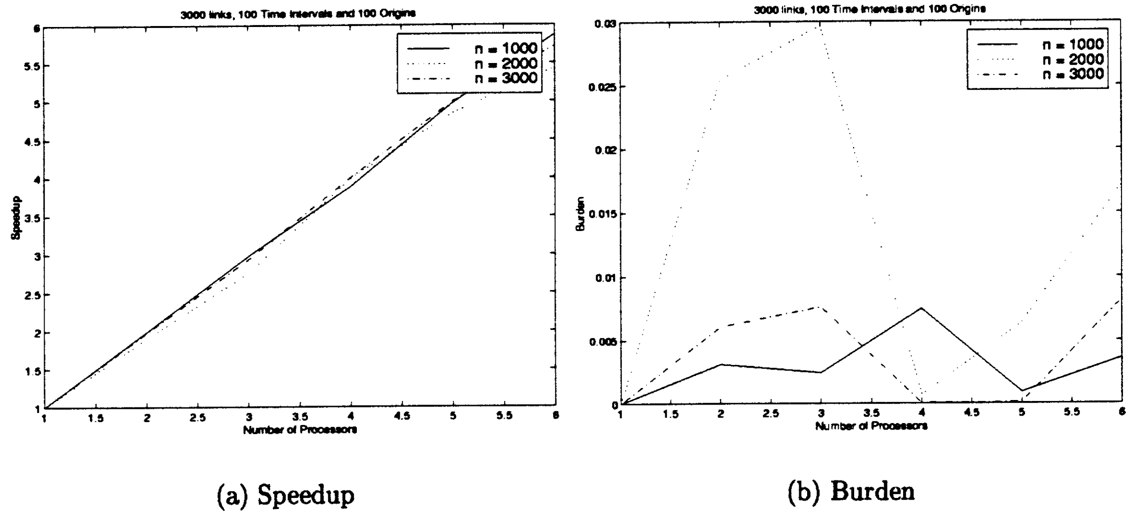


(a) Speedup



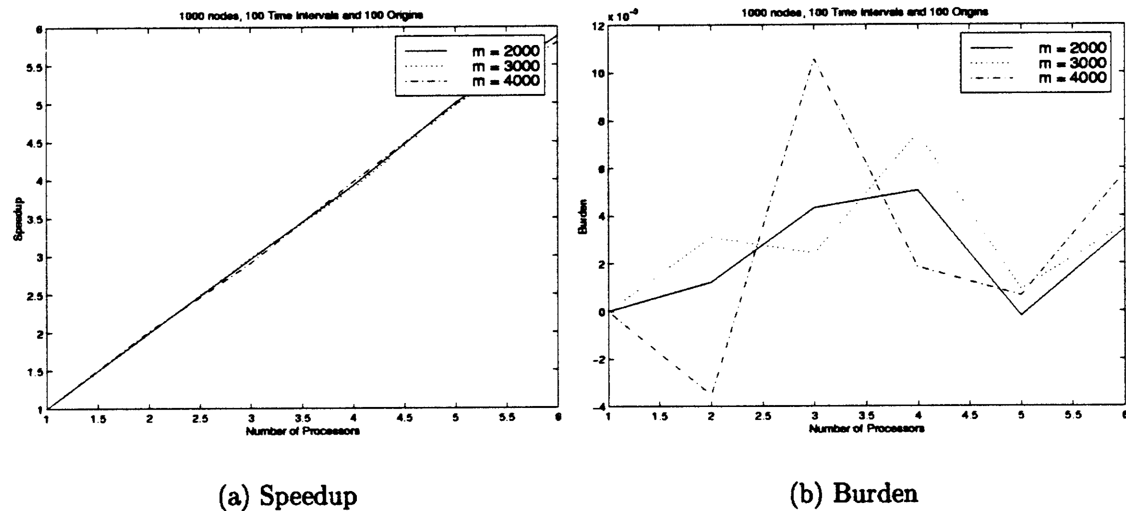
(b) Burden

Figure 4.33: Performance of (MT, origin, IOT) Implementation: varying number of time intervals



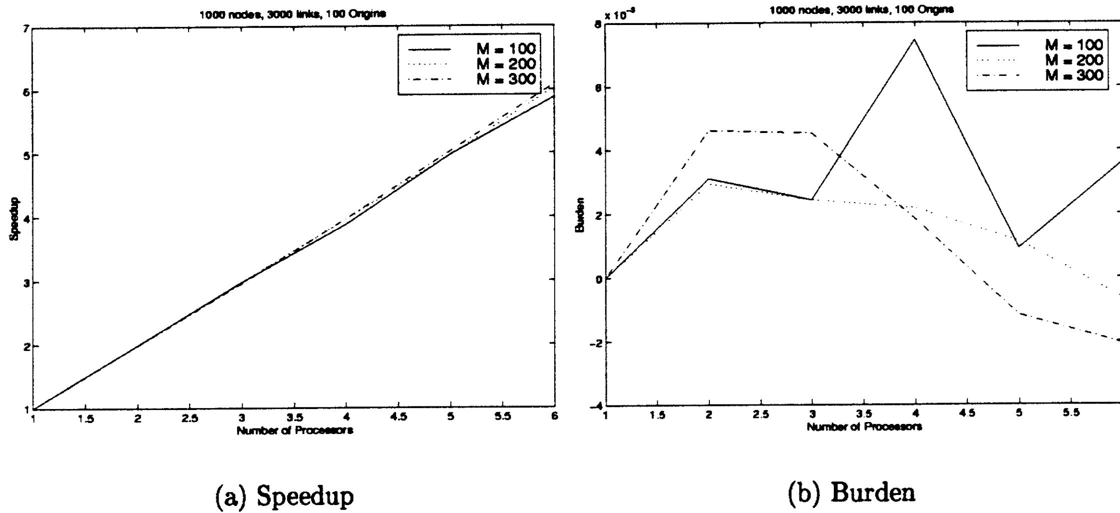
(a) Speedup (b) Burden

Figure 4.34: Performance of (PVM, Departure Time, `lc-dequeue`) Implementation: varying number of nodes



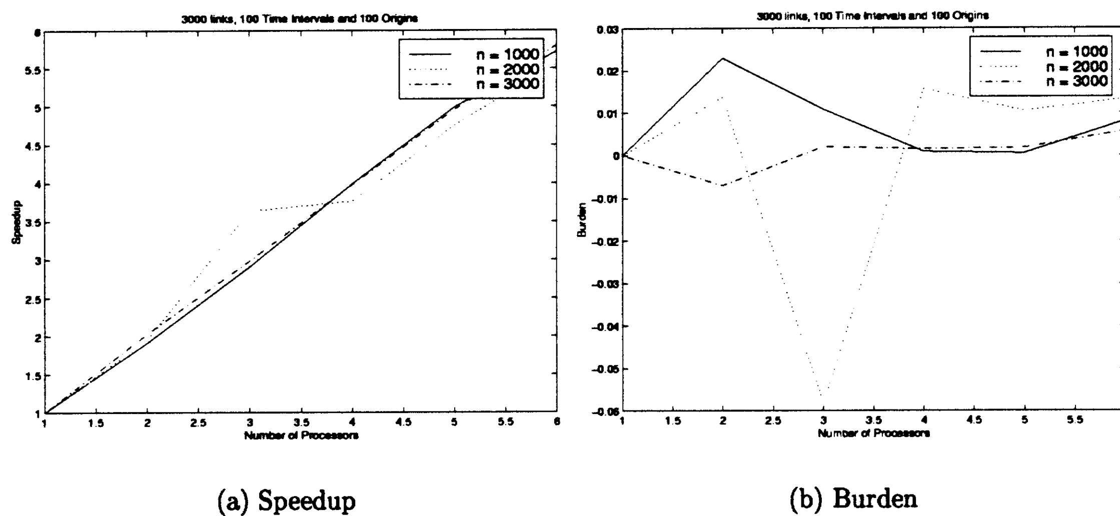
(a) Speedup (b) Burden

Figure 4.35: Performance of (PVM, Departure Time, `lc-dequeue`) Implementation: varying number of links



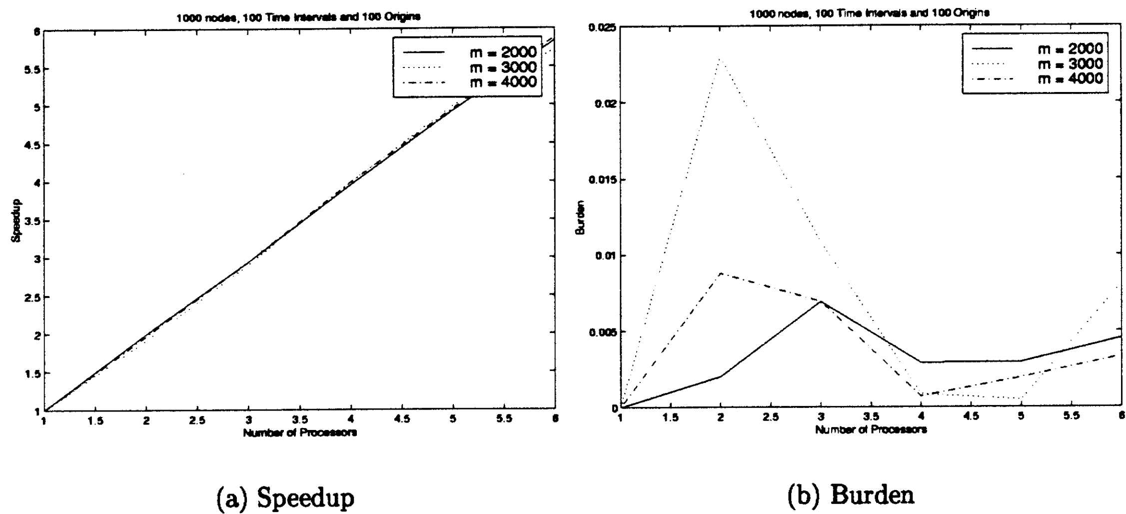
(a) Speedup (b) Burden

Figure 4.36: Performance of (PVM, Departure Time, `lc-dequeue`) Implementation: varying number of time intervals



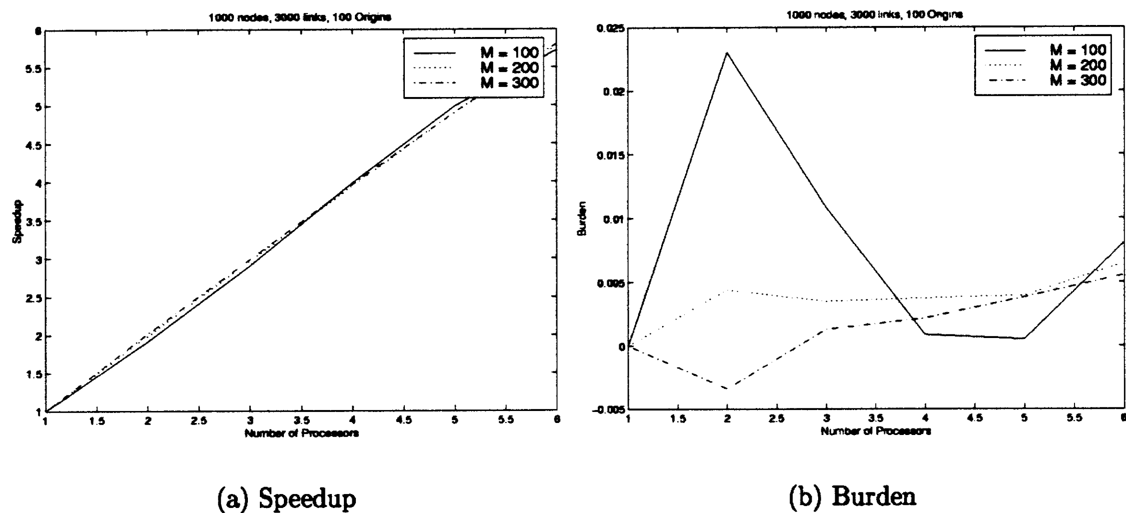
(a) Speedup (b) Burden

Figure 4.37: Performance of (PVM, Departure Time, `IOT`) Implementation: varying number of nodes



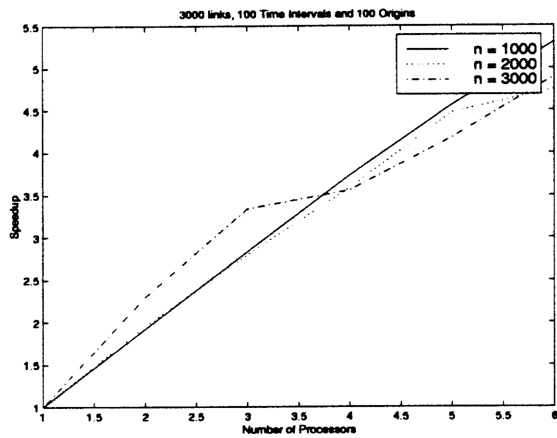
(a) Speedup (b) Burden

Figure 4.38: Performance of (PVM, Departure Time, IOT) Implementation: varying number of links

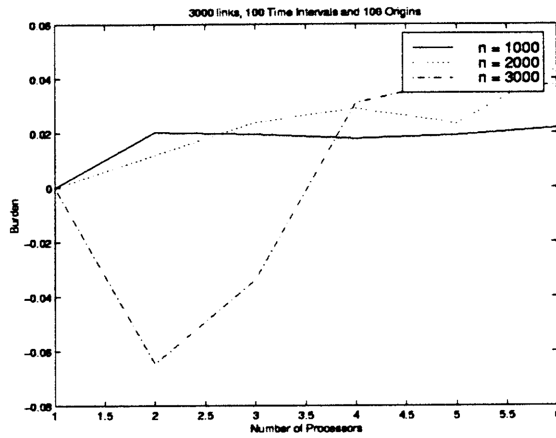


(a) Speedup (b) Burden

Figure 4.39: Performance of (PVM, Departure Time, IOT) Implementation: varying number of time intervals

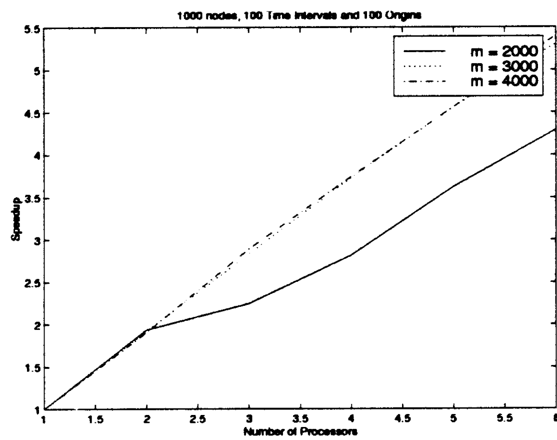


(a) Speedup

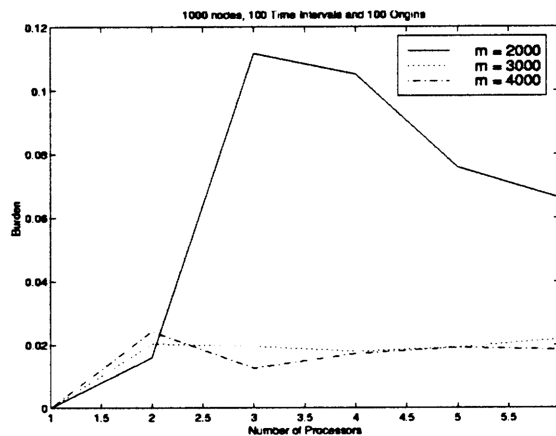


(b) Burden

Figure 4.40: Performance of (MT, Departure Time, `lc-dequeue`) Implementation: varying number of nodes

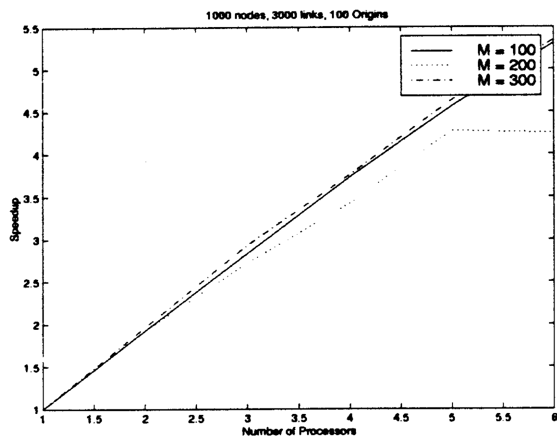


(a) Speedup

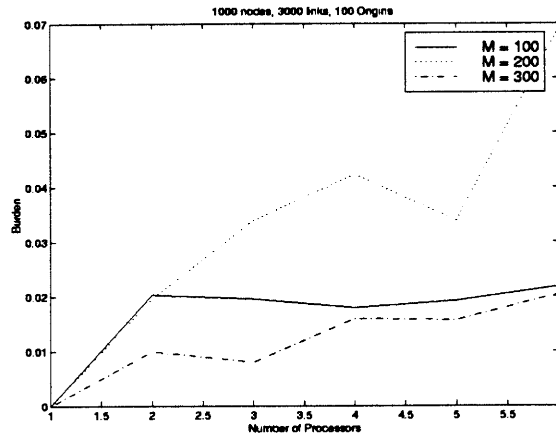


(b) Burden

Figure 4.41: Performance of (MT, Departure Time, `lc-dequeue`) Implementation: varying number of links

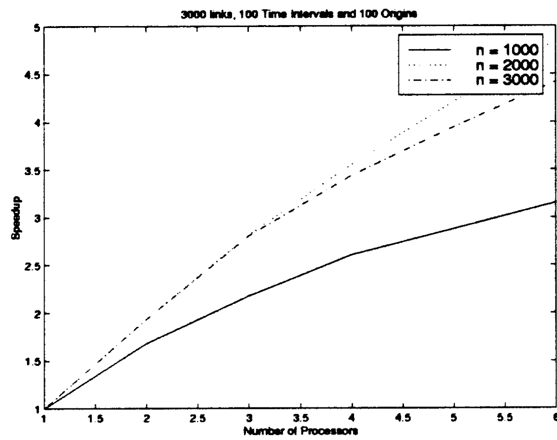


(a) Speedup

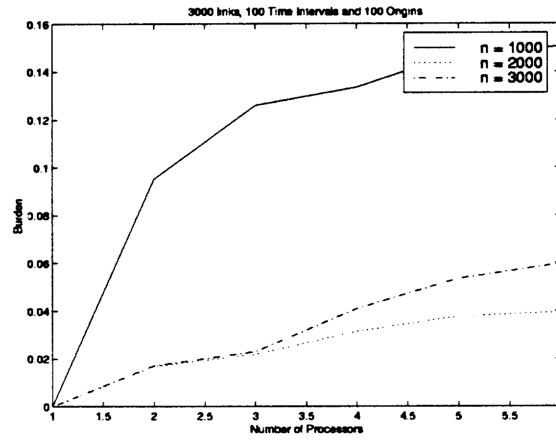


(b) Burden

Figure 4.42: Performance of (MT, Departure Time, `lc-dequeue`) Implementation: varying number of time intervals

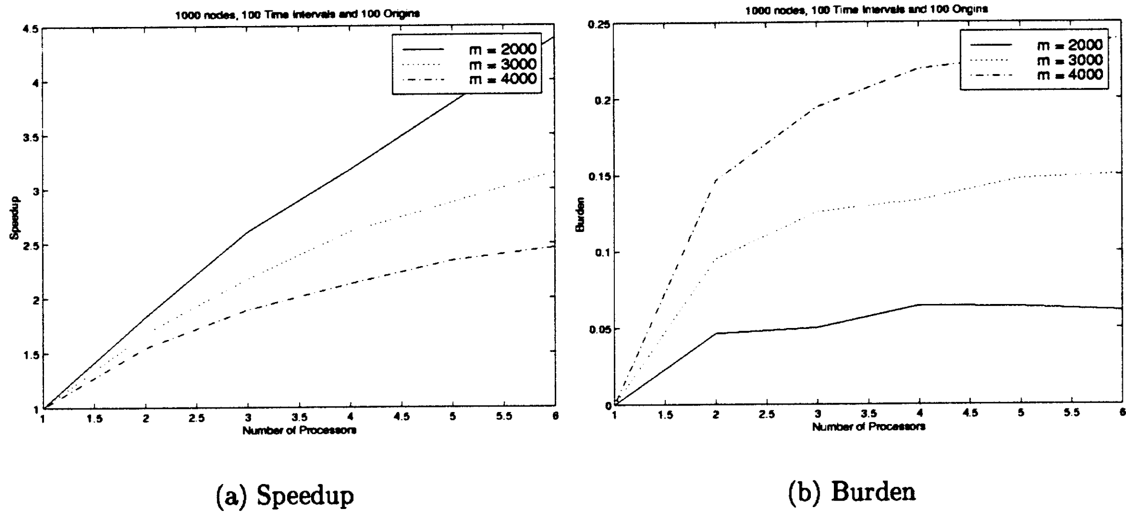


(a) Speedup



(b) Burden

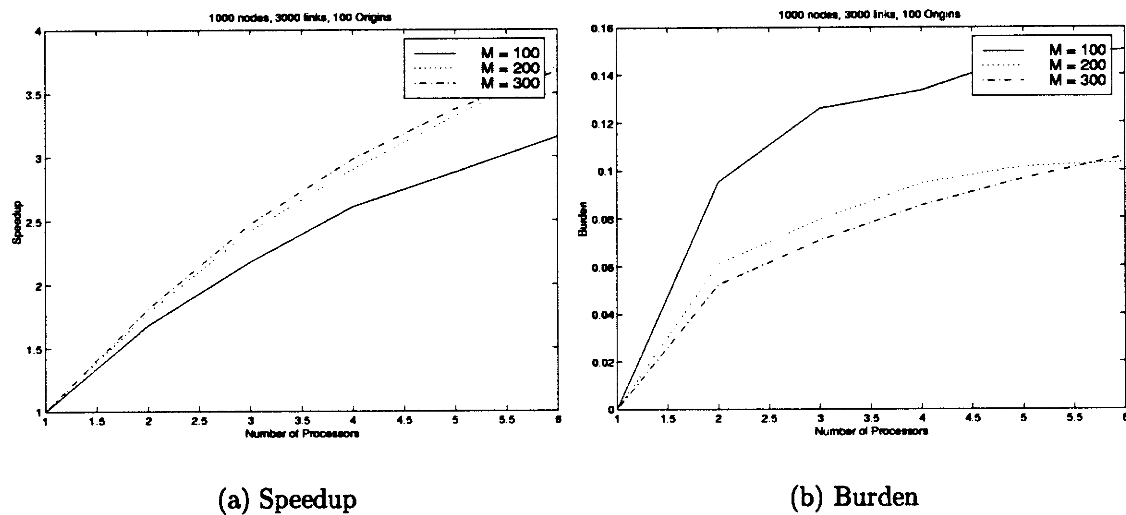
Figure 4.43: Performance of (MT, Departure Time, `IOT`) Implementation: varying number of nodes



(a) Speedup

(b) Burden

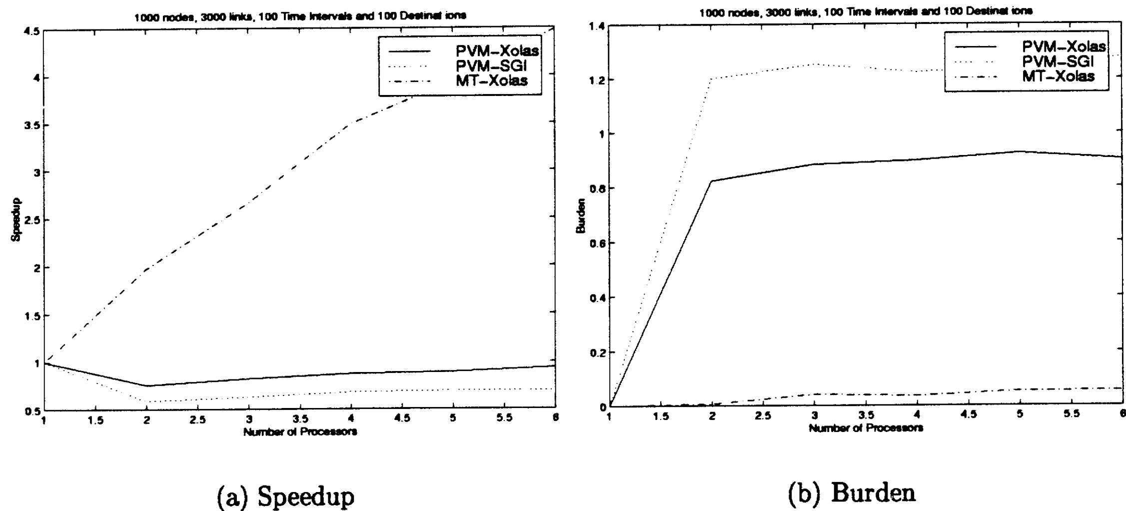
Figure 4.44: Performance of (MT, Departure Time, IOT) Implementation: varying number of links



(a) Speedup

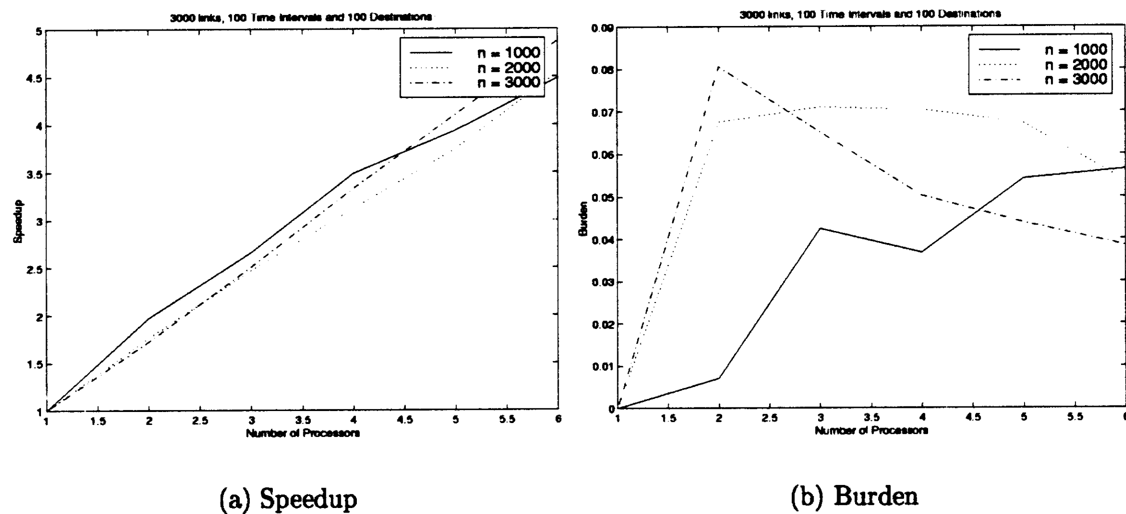
(b) Burden

Figure 4.45: Performance of (MT, Departure Time, IOT) Implementation: varying number of time intervals



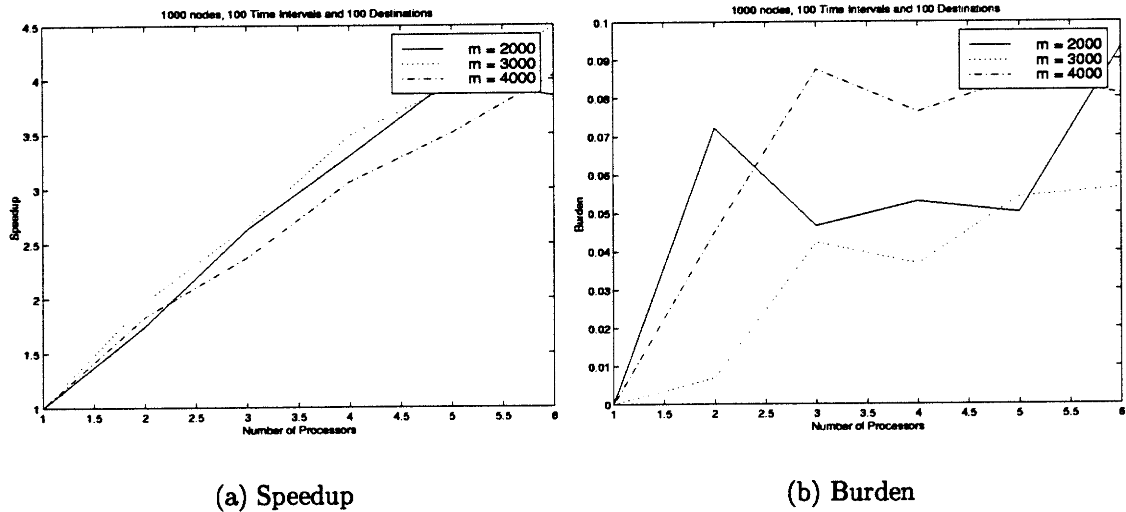
(a) Speedup (b) Burden

Figure 4.46: Decomposition by Network Topology



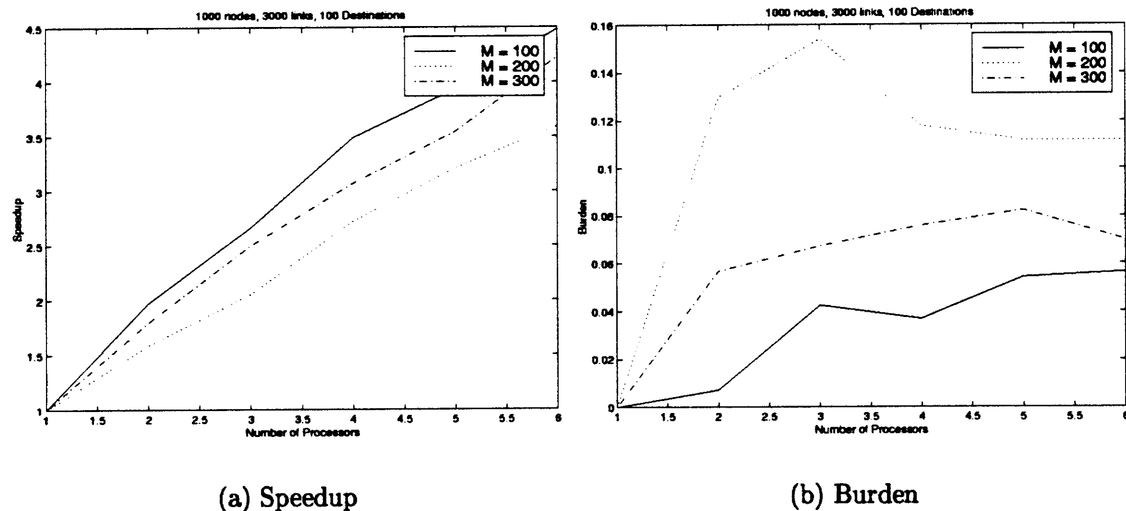
(a) Speedup (b) Burden

Figure 4.47: MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of nodes)



(a) Speedup (b) Burden

Figure 4.48: MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of links)



(a) Speedup (b) Burden

Figure 4.49: MT implementation of Decomposition by Network Topology of algorithm DOT (Varying number of time intervals)

Chapter 5

Application of Dynamic Shortest Paths Algorithms to the Solution of Dynamic Traffic Assignment Models

5.1 Introduction

In the recent years, there has been an increasing interest in Intelligent Transportation systems (ITS) concepts and applications. The goal of ITS is to improve mobility, safety, air quality and productivity. This is achieved by using traffic control and management strategies such as pre-trip or en-route route guidance, signal optimization and ramp metering.

Advanced Traffic Management Systems (ATMS) and Advanced Traveler Information Systems (ATIS) are the two building blocks of ITS. One of the integral parts of ATMS/ATIS is its capability of routing vehicles in response to changing traffic conditions. To support the evaluation and operation of ATMS/ATIS, models that predict future traffic conditions are required. Dynamic Traffic Assignment (DTA) models are used for this purpose. Dynamic traffic assignment models are developed using two

main approaches: analytical([8], [27]) and simulation([2], [20]). In this chapter, we apply the optimal dynamic shortest path algorithms developed in Chapter 2 to the solution of analytical dynamic traffic assignment models. For this application, we use the DTA framework, solution algorithms and computer implementations developed by Chabini and He [8].

For efficient solutions, the DTA model proposed by Chabini and He [8] works on only on a subset of paths in the network. The DTA model uses a time-dependent path generation component to change this subset of paths by adding a new path to the subset when the new path becomes competitive. The eligibility of the new path can be decided based upon a variety of criteria.

The existing implementation of the DTA software in Chabini and He [8] and He [19] does not integrate the time dependent path generation component. In this implementation, the subset of paths is assumed to be fixed and is not altered. We develop one way of implementing time dependent path generation component. Algorithm DOT is used to generate dynamic fastest paths for each OD pair and each time interval. These paths are added to the subset of paths, if they are not already present. In this chapter, we describe the implementation of time-dependent path generation component in detail. We conclude that the existing array representation of the paths storage data structure is not suitable when a dynamic set of paths is required. Hence, we design a new representation of this data structure which is more suitable for this purpose.

This chapter is organized as follows: Section 5.2 describes the conceptual framework for analytical dynamic traffic assignment problem developed by Chabini and He [8]. Section 5.3 describes the data structures used to store paths in the existing implementation of the DTA model. Section 5.4 describes the integration of the dynamic shortest path algorithms into this DTA framework. We also present details of computer implementation of this integration. Section 5.5 presents the experimental evaluation of this computer implementation. Section 5.6 presents the new representation of the paths storage data structure Finally, Section 5.7 summarizes this chapter.

5.2 A Conceptual Framework for Dynamic Traffic Assignment Problem

A modeling framework for the dynamic traffic assignment problem, as given by Chabini and He [8], is shown in Figure 5.1. This framework contains:

- **Users' Behavior Model Component :** This component takes as input the dynamic Origin-Destination(O-D) trips and the subset of paths between each O-D pair. The dynamic O-D trips are the time-dependent traffic demand for each O-D pair. The users' behavior model component then assigns the dynamic O-D trips among the subset of paths according to users' route choice behaviors. Chabini and He [8] model three classes of users' route choice behaviors. They are:
  - **Class 1:** This class of users are those who either do not have real-time traffic information and use their habitual routes, or those who disregard the information and continue to use their habitual routes. Therefore, the departure flow of each path is known.
  - **Class 2:** This class of users can be used to describe users who receive or have partial traffic information about the network conditions and determine their routes based on their "perceived" rather than actual travel times. Thus, each users' perceived travel time is a random variable with certain distribution.
  - **Class 3:** This class of users are those who choose routes with minimum travel time. These users have access to real-time traffic information and fully comply with the route guidance. Therefore, the routes used by this type of users have minimum actual travel time.
- **Dynamic Network Loading Model Component :** The path flows obtained from the users' behavior model and the link performance models are used as input to the network loading model. This generates the link-based network

conditions, which are then used to compute the path-based network conditions such as path travel times. These path travel times can be used for two purposes, First, they are used as an input to the users' behavior model to assign O-D trips. Second, which is more relevant in this thesis, they are used by the dynamic shortest paths algorithm to generate the paths that need to be added to the original subset of paths. There are two approaches to solve the network loading model : simulation models [29] and analytical models [8]. Chabini and He [8] use analytical models.

- **Link Performance Model Component:** This component is used by the network loading component to generate the link performance measures, such as link travel time, generalized cost etc.

- **Time-Dependent Path Generation Model Component:** This is the component which dynamically generates the subset of paths based on certain criteria. As mentioned earlier, the computer implementation developed in Chabini and He [19] does not include this component. In the rest of the chapter, we demonstrate one way of incorporating this component in the solution of analytical DTA models.

The implementations in Chabini and He [8] achieve computational efficiency by using certain efficient data structures. A good understanding of these data structures is needed to incorporate the time dependent path generation component. Hence, the data structures relevant to the path generation component are discussed in the next section.

5.3 Subpath Data structure

One of the major challenges in the computer implementation of the solution algorithm of DTA models is the representation of paths. There are a large number of paths in a network. Hence, storing and processing them should be as efficient as possible.

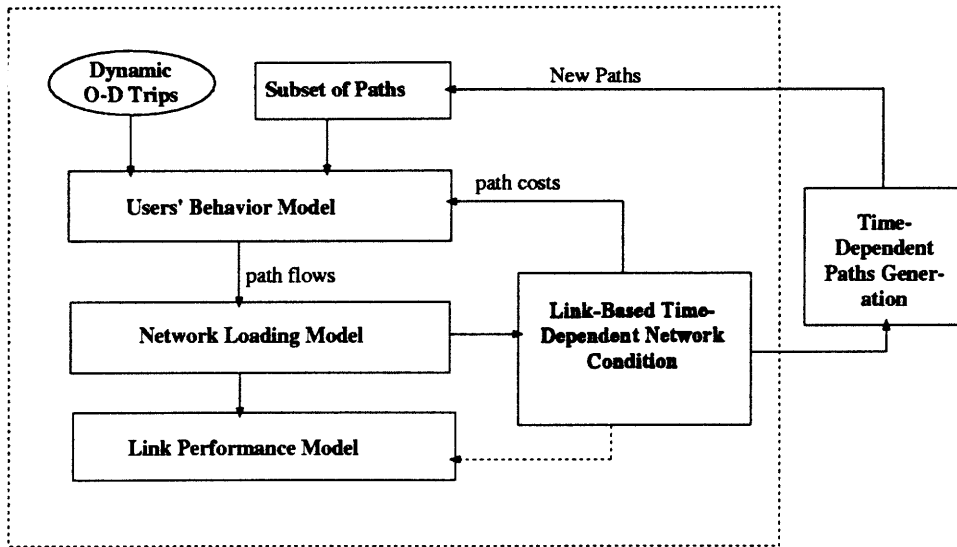


Figure 5.1: A Framework for Dynamic Traffic Assignment Models

A path is a sequence of links. One approach to store a path is to store its sequence of links in a array. In such a representation, paths are independent from each other. This representation is not efficient because it may lead to multiple storing of the same link, as many paths may share that link. For example, the network in Figure 5.2 has two O-D pairs $(1,5)$ and $(2,5)$ with one path between each of them. An independent array representation of paths would be [1,3,4] for OD pair (1,5) and [2,3,4] for OD pair (2,5). In this example, links 3 and 4 are shared by both paths and are repeated in this array representation of paths.

Chabini and He [8] develop a new data structure called `subpath data structure` to represent paths. The main idea behind this data structure is not to repeat the shared part of paths in the representation of paths. This idea is explained using an example. Table 5.1 shows the subpath table for the network in Figure 5.2.

Each of the original paths in the network is given an id number. In Table 5.1, the path [1,3,4] is given id 0. The sequence of links in this path can be obtained by going through the subpath table. As the subpath number of [1,3,4] is 0, the first link is at position 0 in the subpath table. The next link of this subpath is 1 and its next subpath number is 2. The subpath number 2 gives the next link as 3. The next

subpath number of the subpath 2 is 3. The subpath number 3 gives the next link as 4. For this subpath, the next subpath number is -1, which indicates the end of the path. Notice that links 3 and 4 are stored only once in the subpath table. The subpath table avoids link repetitions when two or more paths to a destination node share the same subpath to that destination node.

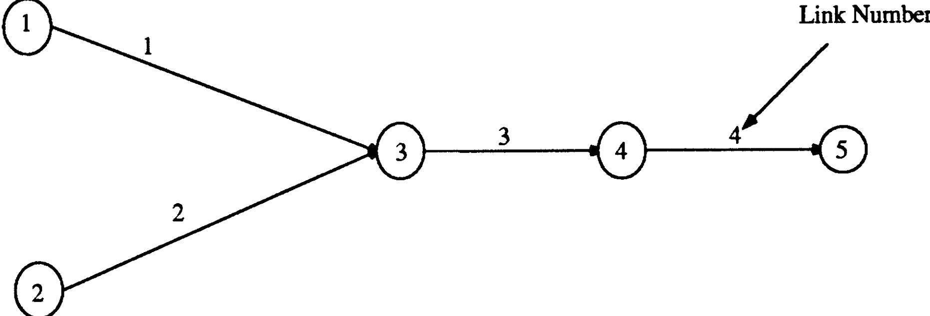


Figure 5.2: A simple network to illustrate the subpath table data structure

subpath number	next-link	next-subpath
0	1	2
1	2	2
2	3	3
3	4	-1

Table 5.1: The subpath table for the network shown in Figure 5.2

A subpath data structure can be represented in a computer memory as a two dimensional array or as a tree. In the computer implementation developed by Chabini and He [8], the subpath data structure is stored as a two dimensional array and it is called as `sub path table`. We will see that the two-dimensional array is not efficient when the set of paths is dynamic. This is due to the static nature of an array representation. Adding an entry to the subpath table requires $O(p)$ operations, using this representation, where p is the size of the subpath table.

Hence, we then designed the tree representation of the sub path data struture in Section 5.6.

5.4 Integration of Dynamic Shortest Path Algorithms into Analytical Dynamic Traffic Assignment Model Framework

It was noted that considering all the paths in the network by a DTA solution algorithm is not efficient. Hence, an efficient algorithm would first start with a subset of paths and then use a time dependent path generation component to update its subset of paths. The time dependent path generation component generates new paths. These paths may be added if certain criteria are satisfied.

In the implementation presented in this chapter, a new path is added between one OD pair if it is faster than the existing paths between that OD pair. This condition is chosen because equipped users (such as class 3 users described in Section 5.2) usually request for shortest paths in the network. We use algorithm `DOT` to compute the fastest paths because this algorithm is proved to be optimal (see Chapter 2)

To integrate dynamic shortest path algorithms into the existing DTA framework, we need two algorithms. First, an algorithm to update the subset of paths. Second, an algorithm to incorporate the time dependent path generation component into the existing DTA framework. The algorithm used to update the subset of paths is called `UPDATE`. The algorithm used to integrate the time dependent path generation component into the existing DTA framework is called `INTEGRATE`. We present the notation used in the algorithms, and the statements of algorithms `UPDATE` and `INTEGRATE` algorithms in the rest of this section.

5.4.1 Notation

Let us denote the dynamic network by $G(N, A, D, C)$ as discussed in Section 2.3. Recall that N is the set of nodes in the network, A is the set of arcs, D is the set of time dependent link travel times and C is the set of link costs. Also, recall that $d_{ij}(t)$ denote the travel time on link (i, j) during time interval t. The number of time intervals is denoted by M. So t belongs to $\{0, \ldots, M-1\}$. Let RS denote the set of

OD pairs. The ordered pair (r,s) denotes an OD pair, where the index r is an origin node and s is a destination node. Let us denote the set of destinations in the network by S, $S = \{s|(r,s) \in RS\}$. Let us denote by K_{rs} the subset of paths between OD pair (r,s). Denote by $k_{rs}(t)$ the fastest path in the subset K_{rs}, between OD pair (r,s) departing node r at time interval t.

We assume avaialble an implementation algorithm DOT such that $\pi_{iq}(t) = DOT(d_{ij}(t), q)$, $\forall i \in N, \forall t, 0 \leq t < M$, where q is the destination node and $\pi_{iq}(t)$ denotes the fastest path between node i and node q departing node i at time interval t. We also assume that $cost(\pi)$ returns the travel time on path π. In the next section, we present the algorithm UPDATE.

5.4.2 Algorithm UPDATE

As mentioned earlier, algorithm UPDATE is used to update the subset of paths. This algorithm takes as input the link travel times. The algorithm UPDATE is:

Algorithm UPDATE

For all $s \in S$

  $\pi_{is}(t) =$ DOT$(d_{ij}(t), s)$

  For all r such that $(r,s) \in RS$

    For $t = 0, \ldots, M-1$

      if$(cost(k_{rs}(t)) > cost(\pi_{rs}(t)))$ then

        $K_{rs}(t) = K_{rs}(t) \cup \pi_{rs}(t)$

        $k_{rs}(t) = \pi_{rs}(t)$

Run time analysis of algorithm UPDATE

The worst case run time complexity of algorithm UPDATE is given by the following proposition.

Proposition 25 *The worst case run time complexity of the algorithm UPDATE is $O(|S| * O(DOT) + |RS| * M * p)$, where p is the number of entries in the subpath table and $\theta(DOT)$ is the exact complexity of algorithm DOT.*

Proof: The complexity can be obtained by counting the number of operations in the algorithm `UPDATE`. We require to compute algorithm `DOT` for all the destinations in the network $(O(|S| * O(DOT))$. We find at most one new path for each OD pair at each time interval. Hence, the maximum number of new paths found is in $O(|RS| * M)$. We have noted earlier that the number of operations required to add a new element in an array requires $O(p)$ operations. Hence, the order of the algorithm `UPDATE` is $O(|S| * O(DOT) + |RS| * M * p)$. □

5.4.3 Algorithm INTEGRATE

The algorithm `INTEGRATE` is used to incorporate the path generation component into the existing DTA framework. In this algorithm the DTA implementation described in Chabini and He [8] is considered as a black box. Figure 5.3 presents the algorithm `INTEGRATE`.

5.5 Experimental Evaluation

The path generation component was incorporated into the software system developed by Chabini and He [8] and He [19]). We evaluate its performance by using two different networks. One is a small test network with 9 nodes and 12 links. It is the same network used by Chabini and He [8] and He [19] to verify the models and solution algorithms of the DTA software system. We use this network to illustrate the validity of the integration og path generation component into the existing DTA software system. This network is shown in Figure 5.4. Recall that class 3 users always have perfect information and choose a fast path. These users are affected by adding a faster path to the subset of paths. Below is a scenario such that all the users are class 3 users.

For this test, the fastest path between one OD pair is not included in the subset of paths, and this path is generated back using algorithm `UPDATE`. Then, the changes in the flows and travel times of class 3 users with and without having the fastest path in the sub path table are shown. Also, the impact of adding a shortest path on the

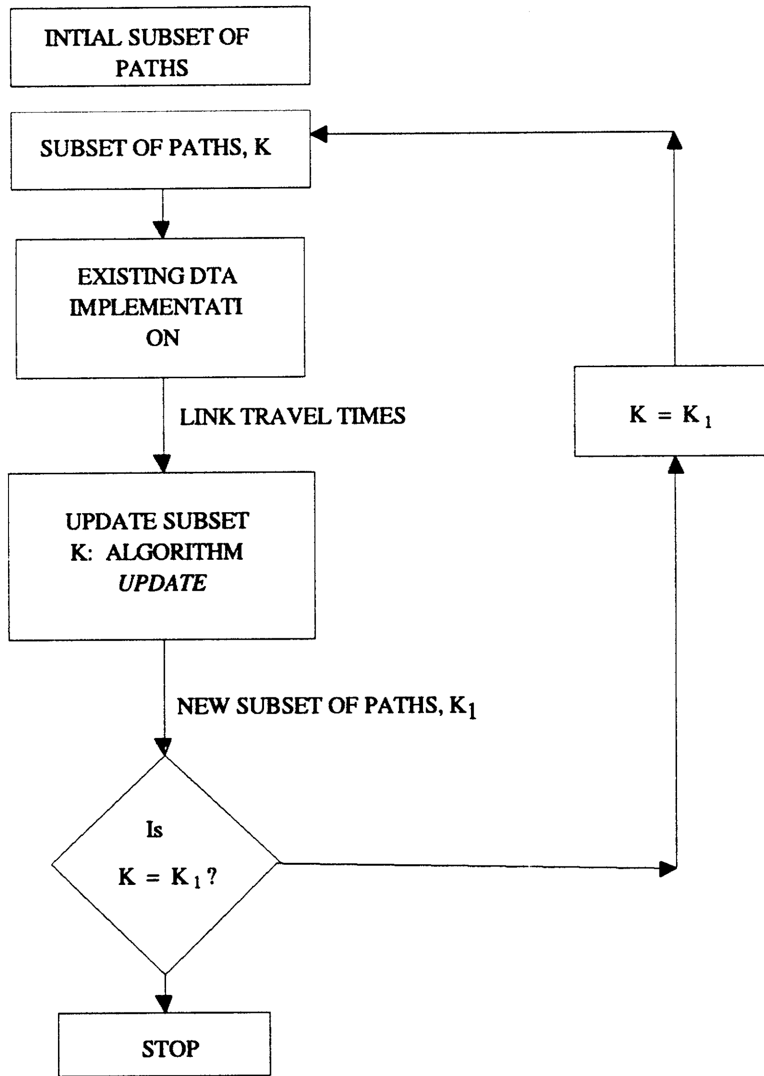


Figure 5.3: Algorithm **INTEGRATE**

convergence rate is shown.

Second, we use the Amsterdam A10 beltway network shown in Figure 5.5 to demonstrate the computation speed of the above algorithms for a realistic network.

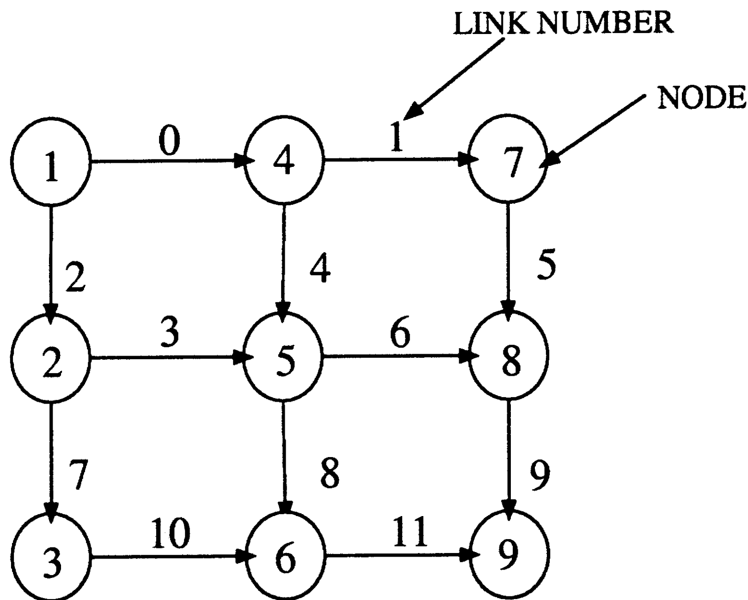


Figure 5.4: Test Network

Small Test Network

The test network is shown in Figure 5.4. The shortest path between origin node 1 and destination node 9 was obtained by running the DTA code on the complete set of paths. This shortest path is computed as sequence of links 2-3-6-9. For the evaluation of algorithms `UPDATE` and `INTEGRATE`, this shortest path is removed from the set of paths. The initial subpath table was constructed without the shortest path between origin node 1 and destination node 9 in it. Table 5.2 shows this subpath table. Then, using algorithms `DOT` and `UPDATE`, the path 2-3-6-9 is generated back. The current implementation of algorithm `UPDATE` does not recognize the fact that the subpath 3-6-9 already exists in the subpath table. It adds the whole path to the subpath table. The updated subpath table is shown in Figure 5.3. The added rows in the subpath table are shown in boldface.

The differences in the travel times and flows on different paths with and without the new faster path are noted. The Table 5.4 shows the travel times without the addition of the new faster path. Table 5.5 shows the path travel times after updating the subpath table. We can see from the table that there are initially 5 paths between the OD pair (1,9). A new path is added to this list of paths. Also, this added path is faster than the rest of the paths for earlier time intervals.

Table 5.6 and Table 5.7 show the flows on different paths before and after adding the new fater path.

We run ten iterations of network loading algorithm with and with out the fastest path between OD pair (1,9). The measure of consistency [8] is reported in Table 5.8. We notice that the measure of consistency is lesser with the addition of the new path. At convergence, this measure is equal to zero.

Amsterdam Beltway Network

The Amsterdam A10 beltway consists of two 32-km freeway loops which intersect with five major freeways and have 20 interchanges of various sizes (75 ramp intersections). This network is shown in Figure 5.5. The network serves local and regional traffic and acts as a hub for traffic entering and exiting Holland. Chabini and He [8] and He [19] uses this network to demonstrate the computational performance of the DTA software system on a realistic network.

We have verified the validity of the algorithm UPDATE using a small test network in the previous evaluation. We are now interested in the running time complexity of the path generation module.

Amsterdam Beltway network has 213 nodes, 310 links and time frame is divided into 2215 time intervals. There are 1134 OD pairs and approximately 1400 paths to start with. The subpath table has 27080 entries. We run the DTA code and then use the algorithm UPDATE to update the subset of paths. After one iteration of algorithm UPDATE, 321 new paths are added. The subpath table size is increased to 32000. The computation time taken by the different components is given in Table 5.9. Table 5.9 shows that updating the subpath table requires a lot of computation time.

Next Link	Sub Path	Index
0	20	0
0	21	1
0	22	2
2	25	5
0	18	6
2	19	7
6	14	8
8	16	9
2	17	10
10	16	11
0	15	12
5	14	13
9	-1	14
1	-1	15
11	-1	16
7	-1	17
4	-1	18
3	-1	19
1	13	20
4	8	21
4	9	22
3	8	23
3	9	24
7	11	25

Table 5.2: Initial subpath table for the network in Figure 5.4

Next Link	Sub Path	Index
0	23	0
0	24	1
0	25	2
0	27	3
0	28	4
2	**6**	**5**
3	**7**	**6**
6	**8**	**7**
9	**-1**	**8**
0	21	9
2	22	10
6	17	11
8	19	12
2	20	13
10	19	14
0	18	15
5	17	16
9	-1	17
1	-1	18
11	-1	19
7	-1	20
4	-1	21
3	-1	22
1	14	23
4	11	24
4	12	25
3	11	26
3	12	27
7	14	28

Table 5.3: Updated subpath table for the network in Figure 5.4

time(t)	Path-1	Path-2	Path-3	Path-4	Path-5
7.250000	10.733	8.710	8.721	8.460	8.582
7.255000	10.611	8.807	8.797	8.606	8.667
7.260000	10.712	9.033	8.873	8.829	8.840
7.265000	10.562	9.176	9.060	9.131	9.046
7.270000	10.468	9.483	9.330	9.347	9.365
7.275000	10.455	9.860	9.792	9.691	9.676
7.280000	10.437	10.194	10.522	9.908	9.968
7.285000	10.560	10.268	11.064	10.403	10.407
7.290000	10.919	10.802	10.756	10.705	10.722
7.295000	11.539	11.356	11.962	10.716	10.748
7.300000	12.092	11.846	12.518	11.446	11.500
7.305000	12.924	12.659	13.182	12.229	12.148
7.310000	12.698	12.537	13.019	13.417	13.531
7.315000	13.586	13.109	13.649	13.955	13.984
7.320000	12.852	12.327	12.747	15.298	15.399
7.325000	13.332	12.909	13.438	14.252	14.268

Table 5.4: Path Travel Times (minutes) for OD pair (1,9) before the path generation

time(t)	Path-1	Path-2	Path-3	Path-4	Path-5	**Path-6**
7.250000	10.770	8.709	8.720	8.463	8.584	**8.521**
7.255000	10.642	8.805	8.787	8.616	8.675	**8.611**
7.260000	10.747	9.040	8.835	8.836	8.852	**9.042**
7.265000	10.637	9.275	9.024	9.060	9.009	**9.348**
7.270000	10.474	9.532	9.298	9.318	9.337	**9.547**
7.275000	10.459	9.843	9.714	9.695	9.687	**9.804**
7.280000	10.449	10.158	10.410	9.967	9.987	**9.878**
7.285000	10.634	10.298	10.856	10.652	10.581	**10.293**
7.290000	10.795	10.655	11.077	10.851	10.805	**10.573**
7.295000	11.271	11.015	11.359	11.284	11.123	**10.812**
7.300000	11.805	11.663	11.765	12.125	11.947	**12.056**
7.305000	12.819	12.631	12.366	12.215	12.195	**12.428**
7.310000	12.858	13.026	12.644	13.407	13.494	**13.678**
7.315000	13.843	13.618	13.553	13.341	13.339	**13.490**
7.320000	13.318	13.095	12.855	14.655	14.596	**14.588**
7.325000	14.043	13.683	13.497	13.401	13.422	**13.570**

Table 5.5: Path Travel Times (minutes) for OD pair (1,9) after the path generation

time(t)	Path-1 Flow	Path-2 Flow	Path-3 Flow	Path-4 Flow	Path-5 Flow
7.250000	0.0000	4.3319	0.0000	38.9867	0.0000
7.255000	0.0000	8.6637	0.0000	34.6548	0.0000
7.260000	0.0000	24.3793	0.0000	85.3274	12.1896
7.265000	0.0000	37.6770	18.838	18.8385	113.0311
7.270000	0.0000	24.2785	145.67	24.2785	48.5570
7.275000	0.0000	85.5289	85.528	28.5096	85.5289
7.280000	0.0000	94.5956	63.063	126.127	31.5319
7.285000	0.0000	200.071	33.345	66.6904	33.3452
7.290000	0.0000	169.748	33.949	33.9496	101.8489
7.295000	0.0000	133.380	33.345	133.380	33.3452
7.300000	0.0000	133.380	0.0000	166.726	33.3452
7.305000	0.0000	94.5956	31.531	94.5956	94.5956
7.310000	0.0000	114.038	28.509	114.038	28.5096
7.315000	0.0000	121.392	0.0000	72.8356	48.5571
7.320000	0.0000	94.1926	0.0000	56.5156	37.6770
7.325000	0.0000	60.9482	0.0000	48.7585	12.1896

Table 5.6: Path Flows for OD pair (1,9) before the path generation

We will demonstrate in the next section that by using a different representation of the subpath data structure, the time required to update the existing subset of paths can be considerably reduced.

5.5.1 Conclusions

The experimental evaluation can be summarized as:

- The correctness of algorithms `UPDATE` and `INTEGRATE` is demonstrated.
- The current implementation of algorithm `UPDATE` is slow. This hinders the real time solution capability of the analytical DTA software system.
- The speed of a dynamic shortest path algorithm is very crucial to the real time solution of DTA problems.

The current computer implementation of the process of updating the set of paths is slow due to the following reasons:

time(t)	Path-1 Flow	Path-2 Flow	Path-3 Flow	Path-4 Flow	Path-5 Flow	Path-6 Flow
7.250000	0.0000	0.0000	0.0000	34.6548	0.0000	8.6637
7.255000	0.0000	0.0000	0.0000	12.9956	0.0000	30.3230
7.260000	0.0000	0.0000	24.3793	48.7585	24.3793	24.3793
7.265000	0.0000	18.8385	56.515	0.0000	94.1926	18.8385
7.270000	0.0000	0.0000	145.67	24.2785	48.5570	24.2785
7.275000	0.0000	28.5096	114.03	28.5096	85.5289	28.5096
7.280000	0.0000	94.5956	63.0637	63.0637	31.5319	63.0637
7.285000	0.0000	66.6904	66.6904	33.3452	33.3452	133.3808
7.290000	0.0000	67.8993	67.8993	33.9496	33.9496	135.7985
7.295000	0.0000	100.03	33.3452	33.3452	33.3452	133.3808
7.300000	0.0000	166.72	33.3452	0.0000	0.0000	133.3808
7.305000	0.0000	126.12	31.5319	0.0000	31.5319	126.1274
7.310000	0.0000	114.0	57.0193	28.5096	0.0000	85.5289
7.315000	24.2785	72.8356	24.2785	24.2785	24.2785	72.8356
7.320000	0.0000	37.6770	75.3541	37.6770	18.8385	18.8385
7.325000	0.0000	24.3793	36.5689	24.3793	12.1896	24.3793

Table 5.7: Path Flows for OD pair (1,9) after the path generation

Measure of Consistency for class 3 before path generation	464.4902
Measure of Consistency for class 3 after path generation	299.00

Table 5.8: Performance measures before and after the path generation

Procedure	CPU time (sec)
Network Loading	208.55
Algorithm DOT for 39 destinations	60.59
Inserting subpath entries into the subpath table	721.20

Table 5.9: Performance of the time dependent path generation component on the Amsterdam Network

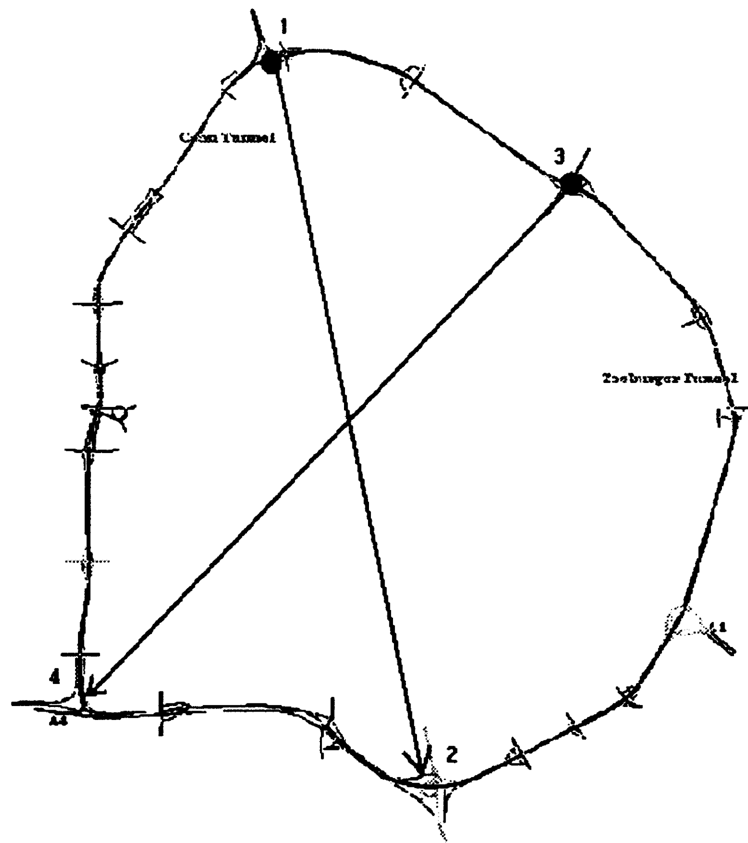


Figure 5.5: Amsterdam Beltway Network

- The two-dimensional array representation of the subpath data structure is static in nature. Hence, inserting an element into the array requires $O(p)$ operations, where p is the size of the array.

- Algorithm `UPDATE` does not minimize the number of subpaths in the subpath table. When a new path is added to the subpath data structure, the existing subpath of the new path in the subpath table is not identified. This will lead to repetitious storage of the same subpaths.

To overcome these disadvantages of the existing array implementation of the subpath data structure, we design a new implementation of the same subpath data structure, called `sub path tree`. This is presented in the next section.

5.6 Subpath Tree - A New Implementation of the Subpath Data Structure

An important requirement of the new representation of paths is the flexibility to add a new path. The data structure should also provide a simple way to find the existing subpath of the new path.

In the subpath tree data structure, subpaths are stored as a tree. The root of the tree indicates all the destinations in the network. Its value is -1. The nodes of the tree are the links in the path and each of them points to the next link on the path and so on. For the small network shown in Figure 5.4, the subpath tree is given by Figure 5.6. Let us consider two paths between the OD pair (1,9) having sequence of links [0,4,8,11] and [2,3,8,11], sharing the same sub path [8,11] to the destination node 9. Both these paths merge into one subpath in the subpath tree data structure.

5.6.1 Adding a new path to the subpath tree

To demonstrate the procedure used to add a new path to the subpath tree, the fastest path between OD pair (1,9) is not included in Figure 5.6. We delete it from the list of

paths to generate it back using the path generation component. The fastest path is the sequence of links [2,3,6,9]. To add this new path into the subpath tree, we start with the last link in the path and iteratively traverse the tree to find the longest subpath of the new path. The longest path of the new path [2,3,6,9] is [3,6,9]. Hence, link 2 is added as a branch to node 3. The updated subpath tree is shown in Figure 5.7.

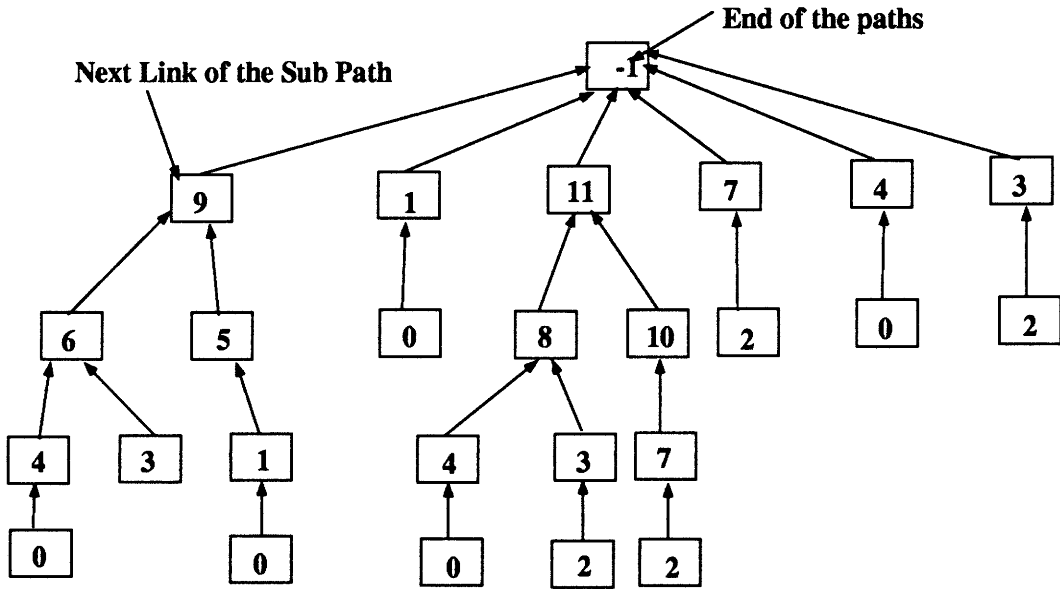


Figure 5.6: Sub Path Tree for the network in Figure 5.4

Proposition 26 *The worst case run time complexity of adding a new path to the sub path tree is $O(L * k)$, where L is the maximum number of links in a path in the network and k is the degree of the network.*

Proof: Adding a new path to the subpath tree is proportional to the height of the sub path tree and the number of children to be scanned at each level. The height of the subpath tree is equal to the maximum number of links in a path in the network (L). A node in this subpath tree indicates a link, say l, in the network. The number of children of this node are at most equal to the number of incoming links of the upstream node of this link, l. Hence, the worst case run time complexity of adding a new path to this data structure is $O(k * L)$. The number of incoming links at a node

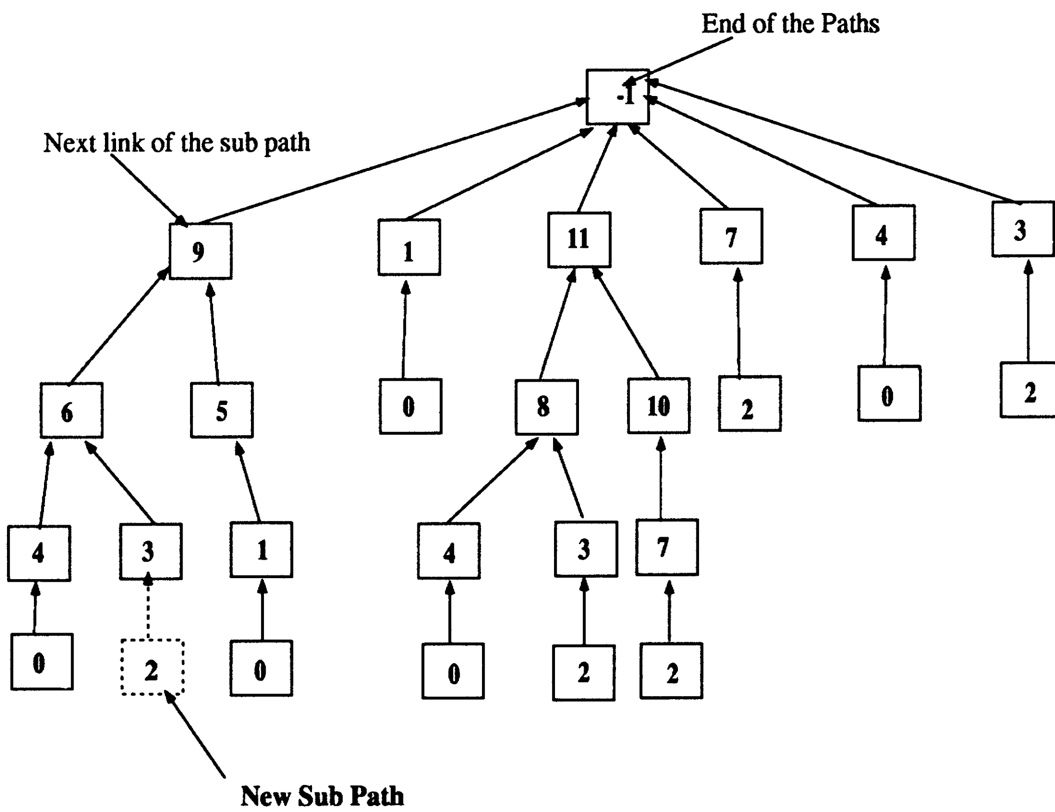


Figure 5.7: Updated Sub Path Tree for the network in Figure 5.4

is proportional to the degree of the network (k). For a traffic network, this degree is usually 3. Hence the worst time complexity for traffic networks is equal to $O(L)$. □

It is proved that the tree representation is much faster than the array representation. One of the major requirements of any data structure used to represent paths is to be able to provide the number and the list of paths between a certain OD pair (r, s). This can be easily obtained in the case of `subpath table` data structure by storing the array indices of the paths between the OD pair (r, s) in a list. Something similar can be implemented in the tree representation too. A list of pointers to all the first links of the paths should be maintained. For example, for the network shown in Figure 5.4, to keep track of the paths between the OD pair (1,9), a list of pointers to its paths in the subpath tree as shown in Figure 5.8 may be used.

The computer implementation of this data structure in the DTA system is under development.

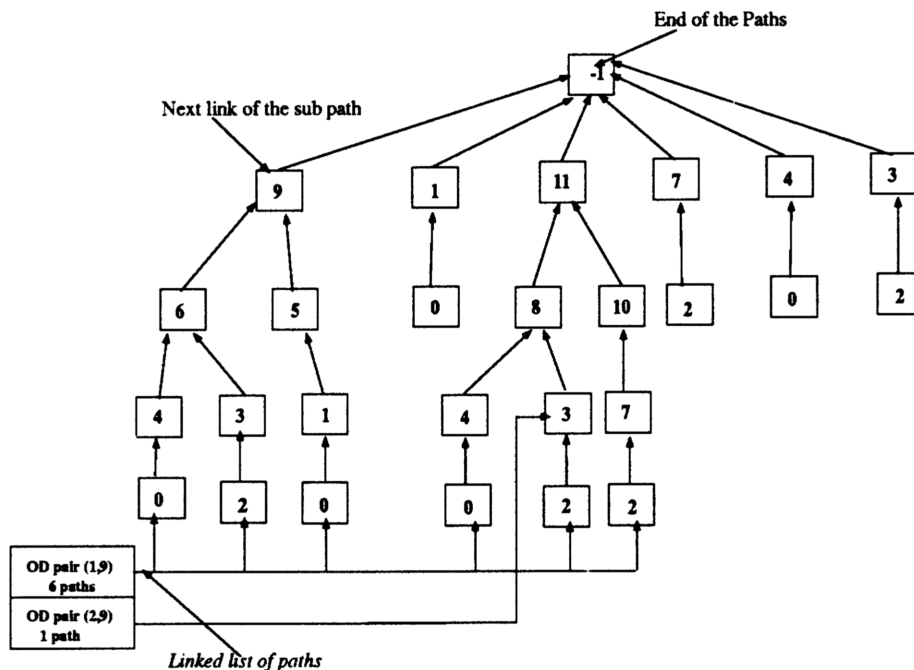


Figure 5.8: An extended sub path tree

5.7 Summary

The application of dynamic shortest path algorithms to the solution of analytical dynamic traffic assignment is discussed. Algorithm `DOT` was applied to the solution of analytical dynamic traffic assignment models developed by Chabini and He [8]. For efficient solutions, the DTA model in Chabini and He [8] optimizes on only a subset of paths. Algorithms `DOT` was used to dynamically update this subset of paths. Algorithms required to update the subset of paths (`UPDATE`) and to integrate the path generation component into the DTA model (called `INTEGRATE`) were developed. The validity of these algorithms was proved for a small test network. Experimental results on a realistic network suggest that algorithm `UPDATE` requires high amount of computation time. This is due to the data strcuture used to store paths in a network. We designed a new representation of the `subpath` data structure. In the new representation, the `subpath` data structure is represented as a tree. This new data structure is called `subpath tree`. This would constitute an afficient representation of dynamic set of paths.

Chapter 6

Summary and Future Directions

6.1 Summary

Intelligent Transportation Systems (ITS) promise to improve the efficiency of the transportation networks by using advanced processing, control and communication technologies. The analysis and operation of these systems require a variety of models and algorithms. Dynamic shortest paths problems are fundamental problems in the solution of most of these models. ITS solution algorithms should run faster than real time in order for these systems to operate in real time. Optimal sequential dynamic shortest paths algorithms do not meet this requirement for real size networks. High performance computing offers an opportunity to speedup the computation of dynamic shortest path solution algorithms.

The main objectives of this thesis are: (1) To review of efficient sequential dynamic shortest paths algorithms, (2) Efficient parallel implementations for shared memory and distributed memory platforms and (3) Application of dynamic shortest path algorithms to the solution of dynamic traffic assignment models.

To develop parallel implementations, a systematic study of the formulations and efficient sequential algorithms for dynamic fastest paths and dynamic minimum cost paths problems is presented. For each class of dynamic shortest path problems, two kinds of shortest path questions were answered: the computation of one-to-all dynamic shortest paths for one departure time and the computation of all-to-one

dynamic shortest paths for all departure time intervals. We have demonstrated that algorithm DOT developed by Chabini [10] is the most efficient algorithm to compute all-to-many dynamic shortest paths for all departure time intervals. No better algorithm can be discovered to solve all-to-many dynamic shortest paths for all departure time intervals.

Although algorithm DOT is optimal, experimental results show that DOT does not compute dynamic fastest paths fast enough in order for ITS applications. High performance computing provides an opportunity to improve the computation speed of these dynamic shortest path algorithms.

In this thesis, we develop parallel implementations for the two types of coarse grained parallel platforms: (1) Distributed Memory and (2) Shared Memory. PVM library was used to develop distributed memory implementations. Solaris Multithreading library was used to develop shared memory implementations. Distributed memory implementations are tested on a network of SGI workstations and on a cluster of Symmetric Multiprocessor called Xolas. Shared memory implementations are tested on Xolas.

In order to develop parallel implementations, we exploit five decomposition dimensions present in dynamic shortest path algorithms: (1) destination, (2) origin , (3) departure time, (4) network topology and (5) data structure. Parallel implementations of 22 triples of (algorithm, decomposition strategy, parallel computing environment) are developed, analyzed and evaluated. An extensive experimental evaluation demonstrates that shared memory platforms perform better than distributed memory platforms in the implementations of dynamic shortest path algorithms.

Algorithm DOT was applied to the solution of analytical dynamic traffic assignment models developed by Chabini and He [8]. Algorithms required to update the subset of paths and to integrate the path generation component into the DTA model were developed. An improved data structure to store paths is designed. This data structure promises to support the development of improved computer implementations of DTA algorithms.

6.2 Future Directions

This research can be extended in several directions. These directions are given below

6.2.1 Hybrid Parallel Implementations

This is a straightforward extension of the parallel implementations developed in this thesis. In this thesis, we looked at distributed memory implementations and shared memory implementations. One can also consider a hybrid implementation combining both types of implementations. For example, if we need to compute dynamic shortest paths from all nodes to 100 destinations in a network. Assume we have with us 4 SMP machines with 8 processors each. Then, we could decompose the problem based on destination. Hence, allotting 25 destinations to each machine. Each of these tasks can then be implemented as a multithreaded implementation. Any decomposition strategy, either by destination or by network topology can be used.

6.2.2 Network Decomposition Techniques

We have seen that a decomposition based on network topology also leads to good speedups for algorithm `DOT`. We mentioned that we use a graph partitioning algorithm called `METIS` to decompose the network. This program partitions the network into sub networks with approximately equal number of nodes. The performance of the parallel implementation highly depends on the way the network is partitioned. Hence, other network decomposition strategies may lead to better speedups. These need to be implemented, analyzed and evaluated.

6.2.3 More decomposition strategies

Other dimensions of problems can be used to develop parallel implementations. For example, we have seen that the main loop in algorithm `DOT` is a nested loop with one sub loop on departure time and another on the links in the network. So, if the minimum travel time on the links is known, a parallel implementation based on the

departure time interval can be developed.

6.2.4 Implementation and evaluation of the `subpath tree` data structure

This is one of the major research directions that can stem from the work of this thesis. A new representation of paths called `subpath tree` has been designed, but has not been implemented. Hence, the path generation component and the DTA software system can be developed using the `subpath tree` as an underlying data structure. This implementation should lead to very efficient solution algorithms for analytical dynamic traffic assignment models.

6.2.5 Parallel implementation of the DTA software system

The conceptual framework proposed for a Dynamic Traffic Assignment problem is modular in nature. Hence, it offers many parallelization avenues. The decomposition strategies developed in this thesis or functional decomposition can be used to decompose the operations of the DTA problem. Therefore, this and more parallellisation strategies can be researched and implemented.

Appendix A

Parallel Implementations

In this appendix, we discuss certain features of the PVM and MT programs developed that the reader needs to be familiar with to understand the parallel programs and to use them. We first present the directory structure and then discuss the implementations.

All the implementations are present in the root directory called `parallel`. Figure A.1 describes the directory tree. For the one-to-all implementations, both the decomposition techniques i.e., origin and departure time interval are given by the same executable. Hence, the specific decomposition technique is chosen in the command line argument. A general help for all these implementations:

- The codes are written using C++. The compiier used is *g++*.
- The executable name without any arguments will written the list of command line arguments that need to be used.

A.1 PVM Implementations

In all the PVM implementations, the files *makemaster* and *makechild* are used to the build the master and slave executables respectively. Each of the parallel implementations has its *makemaster* and *makechild* in its directory. The following should be noted before running any PVM program:

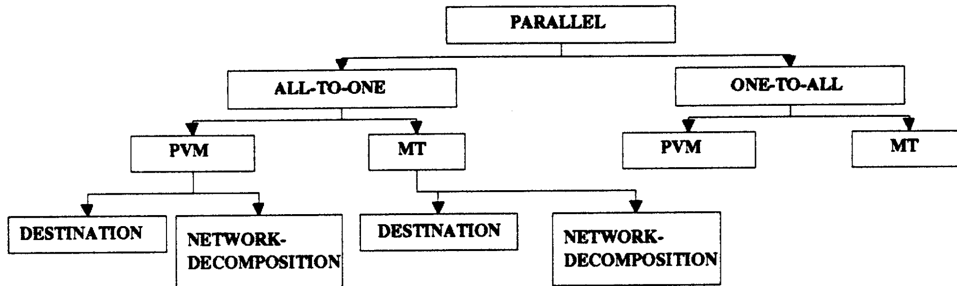


Figure A.1: Directory Structure

- *All the PVM implementations were developed using the PVM version 3.0*
- *PVM Installation:* PVM software can be downloaded from the URL: *http:// www.netlib.org.* PVM should be installed following the instructions given at the time of download. Then, the PVM installation creates two environment variables, *PVM_ROOT* and *PVM_ARCH*. And the *libpvm3.a* is in the directory *PVM_ROOT/bin/PVM_ARCH.* This library should be used for linking any PVM executable.
- *.rhosts and .pvm_hosts files:* To use PVM, one should have available in the home directory, the *.rhosts* file listing all the hosts available on the virtual machine. An example *.rhosts* file used on the xolas machine is given in page 178.

.rhosts file	
# hostname	login name
xolas0	sridevi
xolas1.lcs.mit.edu	sridevi
xolas3.lcs.mit.edu	sridevi
xolas4.lcs.mit.edu	sridevi
xolas-wf	sridevi

Another important file is *.pvm_hosts* file. This file tells the PVM daemon where to look for the slave process executable. Each of our PVM implementations has

a *.pvm_hosts* file in its sub directory. The example *.pvm_hosts* used for the decomposition by destination of algorithm `DOT` is given in page 179.

.pvm_hosts file	
#	Configuration file used for starting PVM programs
xolas0	ep=$HOME/parallel/alltoone/PVM/destination
	wd=$HOME/parallel/alltoone/PVM/destination
xolas-wf	ep=$HOME/parallel/alltoone/PVM/destination
	wd=$HOME/parallel/alltoone/PVM/destination

- Adding a PVM job to the Load Sharing Facility(LSF) queue: LSF queuing system is used on the xolas machines. To have all the resources of a host available to you or if the job is too big, we need to submit the job to the LSF queue. More information about this queuing system is found at the URL: *http://xolas.lcs.mit.edu/LSF/index.html.* This URL has the LSF user guide and programmers' guide. A PVM job can be added to the queue using a *pvmjob* script file. More information on these script files is available in page 134 of the LSF users guide. Each of our PVM implementation has a *pvmjob* script file in its own subdirectory. The command to be used to submit the pvm job to a queue is:

```
bsub -q <queuename> pvmjob    <executablename>    <command line
arguments >
```

There are many queues available on the xolas system. The information about each queue is present in the file */etc/queues*. The script file *pvmjob* writes the *.pvm_hosts* file and starts the pvm daemon using this file.

A.2 MT implementations

MT implementations are much easier to run than the PVM implementations. The compiling and running is like any sequential C++ programs. Again, the make files

are present in the respective directories. The `g++` compiler is used with the linker `-lthread`. An important point to note about the MT implementations is that all the computation times are measured in terms of wall clock time, as any other function returning the time taken returns the CPU time used by the process, which is the total time taken by all the threads in the process.

Appendix B

Time-Dependent Path Generation Module in Dynamic Traffic Assignment

The path generation module is incorporated into the DTA software system developed by He [19]. The information about the DTA software system should be obtained from He [19]. In this appendix, we describe the additions/ changes made to this DTA software to incorporate the time-dependent path generation module.

The following points should be noted before using the DTA software with the path generation module:

- The DTA software system is coded in C++. The various input files required by this software system are described by He [19]. The same input files are required even by the modified DTA software.

- One extra input called the *number of updates* is required by the DTA software system. This input indicates the number of times the sub path table has to be updated. This input is given in a interactive mode, i.e., when the dta software is run, the user is prompted for this input.

- We added a new function to this software, which is used to add an entry into

the path table. This function is called *InsertEntries* and is part of the *main.cpp* file.

- Note that in algorithm DOT, we assumed that the travel times are positive integers. But, in DTA models, the travel times take real values. Hence, algorithm DOT was extended to take real travel time values. The C++ codes for this extended DOT are present in the DTA directory. The changes required to change algorithm DOT are minor.

- Figure B.1 shows the sequence of steps used to incorporate the path generation module in the DTA software system in the form of a flow chart.

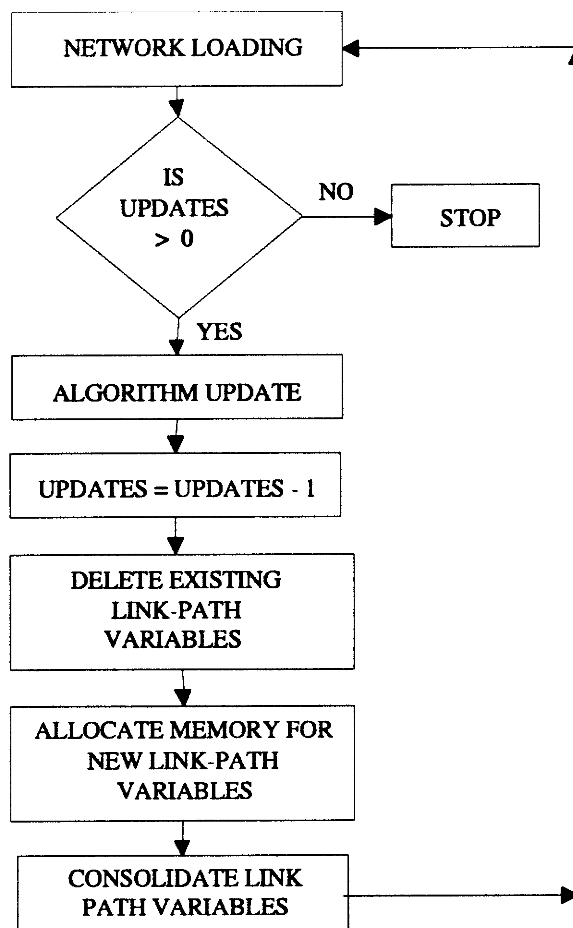


Figure B.1: Flowchart of the Integration of Path Generation Module into Dynamic Traffic Assignment Software

References

[1] Ahn, B.H. and J.Y.Shin, "Vehicle Routing with time windows and time-varying congestion.", *Journal of Operational Research Society*, 42, 393-400, 1991.

[2] Moshe Ben-Akiva, Michel Beirlaire, Haris Koutsopoulos, Rabi Mishalani , "DynaMIT: a simulation-based system for traffic prediction. ", *Paper presented at the DACCORD Short Term Forecasting Workshop* ,Delft, The Netherlands, Febrauary 1998.

[3] D.P. Bertsekas, F. Guerriero and R. Musmanno, " Parallel Asynchronous Label Correcting Methods for Shortest Paths ", *Journal of Optimization Theory and Applications*, Vol. 88, No. 2, pp. 297-320, 1996.

[4] Dimitri Bertsekas, John N. Tsitsikilis, "Parallel and Distributed Computation: Numerical Methods", PrenticeHall, Englewood Cliffs, New Jersey 07632.

[5] Brassard G., Bratley P., " Fundamentals of Algorithmics ", PRENTICE HALL, Englewood Cliffs, New Jersey 07632.

[6] Chabini I., Brian C Dean, "Optimal Algorithms for Computing Shortest Paths in Dynamic Networks with Bounded Waiting at Vertices ", MIT-ITS report, 1997.

[7] Chabini, I., Gendron, B., "Parallel Performance Measures Revisited. ", The Proceedings of *High Perfmance Computing Symposium 95*, Montreal, Canada, July 10-12, 1995.

[8] Chabini, I. , He, Y., "An analytical approach to dynamic network loading: formulation, algo rithms, and computer implementations. ", *Submitted to Trans-*

portation Science, 1997.

[9] Ismail Chabini, Michael Florian and Eric Lesaux, "High Performance Computation of Shortest Routes for Intelligent Transportation Systems Applications.", *'Steps Forward'. Proceedings of the Second World Congress of Intelligent Transport Systems '95*, Yokohama,pp. 2021-6 vol.4.

[10] Ismail Chabini, "A New Algorithm for Shortest paths in discrete dynamic networks", *Preprints of 8th IFAC/IFIP/IFORS Symposium*, Chania, Greece, pp. 551-557, volume II (1997).

[11] Chang, G.-L.; Junchaya, T., " A Dynamic Route Assignment Model for Guided and Unguided Vehicles with a Massively Parallel Computing Architecture", *Mathematical and computer modeling*, Vol. 22, No. 4,pp. 377-395 (1995).

[12] K.L.Cooke and E.Halsey., "The shortest route through a network with time dependent internodal transit times.", *Journal of Math. Anal. Appl.*, Vol 14, 1996, pp. 492-298.

[13] Dijkstra, E.W , "A Note on Two Problems in Connection with Graphs.", *Numer. Math.*, 1, 269-271, 1959.

[14] R.B.Dial, F.Glover, D.Karney, and D.Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees.", *Networks* 9, 1979, pp.215-248.

[15] E.W.Dijkstra, "An Appraisal of Some Shortest Path Algorithms.", *Operations Research*, Vol. 17, 1969, pp.395-412.

[16] Flynn, M.J. "Some computer organizations and their effectiveness.", *IEEE Transactions on Computers*, C-21(9), pp.948-960 (1972).

[17] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchak, Vaidy Sunderam, " PVM : A Users' Guide and Tutorial for Networked Parallel Computing. ", The MIT Press, Cambridge, MA 02142.

[18] Mayiz B. Habbal, Haris N. Koutsopolous, Steven R. Lerman, "A Decomposition Algorithm for the All-Pairs Shortest Path Problem on Massively Parallel Computer Architectures", *Transportation Science*, Vol. 28, No.4,pp 292-308, Nov. 1994.

[19] Yiyi He, "A Flow based approach to the Dynamic Traffic Assignment Problem : Formulations, Algorithms and Computer Implementations" M.S. Thesis, Dept. of Civil and Environmental Engg., MIT.

[20] H. S. Mahmassani, T.-Y. Hu, S. Peeta, and A. Ziliaskopoulos. Development and testing of dynamic traffic assignment and simulation procedures for atis/atms applications. *Report DTFH61-90-R-00074-FG, U.S. DOT, Federal Highway Administration*, McLean, Virgina, 1994.

[21] Michelle Hribar, Valerie E. Taylor, "Reducing the Idle Time of Parallel Transportation Applications", *submitted to the International Parallel Processing Symposium.*

[22] Lacagnina,V.; Russo, R., " The combined distribution/assignment problem in transportation network planning: a parallel approach on hypercube architecture", *Applications of High-Performance Computing in Engineering* IV, p. 332, 279-87 (1995).

[23] D.E.Kaufman and R.L.Smith, "Minimum Travel Time Paths in Dynamic networks with Application to Intelligent Vehicle Highway Systems. ", *IVHS Journal*, 1993, Vol 1(1), pp. 1-11.

[24] Bil Lewis, Daniel J. Berg, "Threads Primer : A Guide to Multithreaded programming.", PRENTICEHALL, Upper Saddle River, NJ 07458.

[25] A. Orda and R.Rom, "Shortest-Path and Minimum Delay Algorithms in Networks with Time Dependent Edge-Length.", *Journal of the ACM*, Vol. 37, 1990, pp. 607-625.

[26] Pape, U. "Implementation and Efficiency of Moore-Algorithms for the shortest route problem .", *Mathematical Programming*, 7, 212-222, 1974.

[27] B.Ran, D.E.Boyce, and L.J.LeBlanc. "A new class of instantaneous dynamic user optimal traffic assignment models.", *Operations Research*, 41(1):192-202, 1993.

[28] Susann Ragsdale, "Parallel Programming", Intel McGrawhill Publications.

[29] Q. Yang, "A microscopic traffic simulator for evaluation of dynamic traffic management systems.", *Transportation Research*, 4(3), 1996, pp.113-129.

[30] Athanasios Ziliaskopoulous, Dimitris Kotzinos, and Hani Mahmassani, "Design and Implementation of Parallel time dependent least time path algorithms for Intelligent Transportation applications.", *Transportation Research Part C*, Vol. 5, No.2, 1997, pp. 95-107.