Parallel Implementations of Dynamic Traffic Assignment Models and Algorithms for Dynamic Shortest Path Problems

by

Hai Jiang

B.S., Civil Engineering, Tsinghua University (2001)

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

Master of Science in Transportation

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author .....
Department of Civil and Environmental Engineering
Septermber 10, 2003

Certified by .....
Ismail Chabini
Associate Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Heidi M. Nepf
Chairperson, Department Committee on Graduate Studies

Parallel Implementations of Dynamic Traffic Assignment Models and Algorithms for Dynamic Shortest Path Problems

by

Hai Jiang

Submitted to the Department of Civil and Environmental Engineering
on Septermber 10, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Transportation

Abstract

This thesis aims at the development of faster Dynamic Traffic Assignment (DTA) models to meet the computational efficiency required by real world applications. A DTA model can be decomposed into several sub-models, of which the most time consuming ones are the dynamic network loading model and the user's route choice model.

We apply parallel computing technology to the dynamic network loading model to achieve faster implementations. To the best of our knowledge, this concerns the first parallel implementations of macroscopic DTA models. Two loading algorithms are studied: the iterative loading algorithm and the chronological loading algorithm. For the iterative loading algorithm, two parallelization strategies are implemented: decomposition by network topology and by time. For the chronological loading algorithm, the network topology decomposition strategy is implemented. Computational tests are carried out in a distributed-memory environment. Satisfactory speedups are achieved.

We design efficient shortest path algorithms to speedup the user's route choice model. We first present a framework for static shortest path algorithms, which prioritize nodes with optimal distance labels in the scan eligible list. Then we apply the framework in dynamic FIFO, strict FIFO, and static networks. Computational tests show significant speedups.

We proceed to present two other shortest path algorithms: Algorithm Delta and Algorithm Hierarchy. We also provide the evaluations of the algorithms.

Thesis Supervisor: Ismail Chabini
Title: Associate Professor of Civil and Environmental Engineering

Acknowledgments

I would like to thank Professor Chabini for his guidance, insights and encouragement throughout all stages of the research. I would like to thank his patience in explaining to me many concepts and ideas. He is not only a respectful advisor, but also a great friend of mine. In addition, I enjoy many of the discussions we had in the past two years about life.

I would like to thank my collegues in the ACTS group for their friendship and assistance.

Thanks to Ford Motor Company and the National Science Foundation (CAREER Award Grant CMS-9733948 and ETI Award Grant CMS-0085830) for supporting this research.

Thanks to my family for their love and confidence in my ability to succeed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Congestion in road transportation systems has reached unprecedented level, and costs tens of billions of dollars each year in productivity and extra fuel consumption in the U.S. alone. A recent study by Scharank and Lomax [1] investigated 68 urban areas in the U.S. and reported that the average annual delay per person has climbed from 11 hours in 1982 to 36 hours in 1999. The cost of these delays in these areas is estimated to be $78 billion per year, including $4.5 billion hours of delay and 6.8 billion gallons of fuel. This does not include other negative consequences of congestion such as accidents, air pollution, and higher operating cost for commercial vehicles. On the other hand, travel demand is expected to increase dramatically in the future. The vehicle miles travelled (VMT) are estimated to increase by 50% to reach about 4 trillion by 2020 [2]. Congestion has become an increasingly urgent issue that needs immediate response from the society.

Intelligent Transportation Systems (ITS) are being developed to alleviate congestion, increase the efficiency, and improve the safety of existing transportation facilities. It combines advanced technology in sensing, communication, information, and advanced mathematical methods with the conventional world of surface transportation infrastructure. Examples of ITS technologies include coordinated traffic management, electronic toll collection, and route guidance.

The success of ITS deployment depends on the availability of advanced Traffic Estimation and Prediction Systems (TrEPS) to predict network conditions and analyze network performance. Many ITS sub-systems, especially Advanced Traffic Management Systems (ATMS) and Advanced Traveller Information Systems (ATIS), are heavily dependent on the availability of timely and accurate wide-area estimates of future traffic conditions.

Dynamic Traffic Assignment (DTA) is a critical component of TrEPS. Its main function is to predict time-dependent link flows and link travel times for a given network and a given time-dependent origin-destination demand matrix. The origin-destination demand matrix is usually estimated by combining historical data and real time data gathered by network sensors. The details in OD matrix estimation beyond the scope of this thesis and can be found in [3]. The DTA problem is often solved using an iterative approach of "routing" - "network loading" - "feedback of travel times" to obtain a new route assignment [4, 5, 6, 7]. Figure 1-1 shows a framework for the DTA problem and depicts the iterative approach. The four boxes on the corners represent the four categories of variables. The arrows that connect the boxes represent the sub-problems contained in the DTA problem. Before we explain how this iterative approach works, we explain two sub-problems we focus on in this thesis.

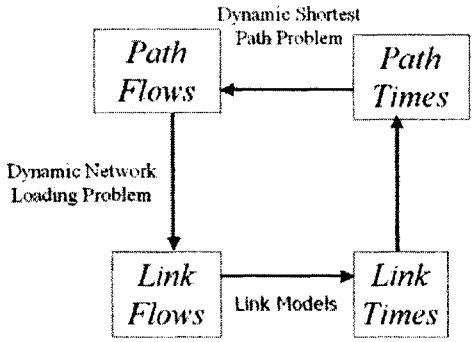

Figure 1-1: DTA framework

The subproblem that computes link flows from a set of route choices (path flow) is called the Dynamic Network Loading Problem (DNLP). It is at the heart of the DTA problem. Depending on how the flow is represented and how the flow propagates in the

network, there are two basic traffic flow models: microscopic models and macroscopic models. In microscopic models, each vehicle is considered as an object. The vehicles move along links by using driving logics that mimic real world driving. Examples of these logics are car-following models and lane-changing models. There are quite a few microscopic DTA models developed in the literature including INTRAS/FRESIM [8, 9, 10], NETSIM, and THOREAU. In macroscopic models, vehicles are aggregated into clusters [1]. Flows propagate along links using various macroscopic traffic flow logics, for example, the volume-delay function [5], the cell transmission model [11], and the hydro-dynamic theory [12]. FREFLO [13] and CONTRAM [14, 15] are examples of macroscopic DTA models. Between the two extreme models, there is a third hybrid model called mesoscopic DTA model. In this model, traffic is represented at the vehicle level, but the speed is obtained from macroscopic traffic flow logics. Examples of mesoscopic DTA models are: DynaMIT [6] and DynaSMART [7].

These models are applied depending on the scope of investigation and different levels of detail that are necessary in the modelling. For the simulation of large road networks, the family of macroscopic flow models is the common choice. Microscopic models are more often used for studying the traffic flow in smaller areas, but then in greater detail. Mesoscopic models is somewhere in between.

The second sub-problem is the Dynamic Shortest Path Problem (DSPP). It concerns the determination of the shortest paths (or fastest paths) between the origins and destinations in the network. In user optimal traffic assignment, the information about shortest paths is important to model user route choice behavior and provide route guidance. There are many variants of the shortest path problem. Depending on the number of origins and destinations, shortest path problems are categorized into one-to-all, all-to-one, one-to-one and many-to-all problems etc. Depending on whether the network satisfies the First-In-First-Out property, there are FIFO and non-FIFO problems. The solution of the one-to-all shortest path problem for all departure times via an iterative Dijkstra's algorithm is a celebrated result [16, 11, 17].

The iterative approach starts by assuming a set of initial path flows. The path

[1] In each cluster there can be 2.5 vehicles and 0.3 vehicles as well.

flows are then loaded through dynamic network loading in the network and link flows are obtained. Link travel times can then be computed using the link model, and lead to a set of path travel times. The OD demands are then assigned to paths according to the new path times. This is called one *DTA iteration.* If the set of new path flows is equal to the initially assumed one, we claim that consistency is reached and stop. Otherwise, we adapt the path flow and take it as the initial path flow for the next iteration. The process is repeated until the path flows reach consistency.

1.2 Research Problems and Solution Approaches

There is currently heightened interest in DTA, particularly in the development of approaches that can be deployed for large-scale real-time applications. He [5] developed a DTA software system, which has an underlying flow based macroscopic DTA model. On a workstation with one Pentium Xeon 2.0 GHz processor and 1 GB RAM, the model based on the iterative network loading procedure requires 22 minutes to predict traffic conditions for a 66-minute analysis period in the Amsterdam beltway network model, which contains 196 nodes, 310 arcs, 1000 O-D pairs. For a network model of Boston, which typically contains 7,000 nodes, 25,000 arcs, 1,000 origins and 1,000 destinations, it is then not able to be used for real time prediction.

In real world DTA applications, the size of the network is growing larger and the time window allowed is getting smaller. There is a significant desire to develop faster DTA models. A faster DTA model enables us to:

- manage traffic over a large network to achieve a global optimal state rather than a suboptimal one. For example, optimization within downtown Boston can improve the network performance in downtown; however, the optimization strategy applied in downtown may worsen the network performance in adjacent areas. The penalty we pay in the adjacent areas may exceed the benefit we gain in the downtown area. If we optimize in the greater Boston area instead of the downtown area, we can achieve a global optimum;

- evaluate multiple traffic management strategies in a time much quicker than real time. DTA can play a role as a simulation laboratory to evaluate traffic management strategies. The faster the DTA model, the more strategies can be evaluated in a given time window. Therefore, the actual applied strategy can be selected from a broader candidate set and we can achieve better network performance;

- predict traffic conditions faster so as to be more responsive to network changes. In real-time traffic management, DTA models are applied in a rolling horizon. An example is given in [18], which we briefly describe here. Suppose now it is 8:00am. A TrEPS system starts an execution cycle. It performs a network state estimation using data collected during the last 5 minutes. When the state of the network at 8:00 is available, the system starts predicting for a given horizon, say one hour, and computes a management strategy which is consistent with the prediction. At 8:07, the system finishes the computation, and is ready to implement the management strategy on the real network. This strategy will be in effect until a new strategy is generated. Immediately following that, the system starts a new execution cycle. Now, the state estimation is performed for the last 7 minutes. While the system was busy computing and implementing the new management strategy, the surveillance system continued to collect real-time information, and update the TrEPS system's knowledge of the current network conditions. The new network estimate is used as a basis for a new prediction and management strategy. The process continues rolling in a similar fashion during the whole day. A faster DTA model can shorten the computation time of execution cycles therefore improve the responsiveness of the TrEPS system.

Additionally, in transportation planning, DTA models are applied in a large geographical area and in conjunction with other models, for example, population generation model and activities generation model, where model dependency exists. A usual way to obtain consistent prediction is through systematic relaxation. An initial plan is generated and acts as input to these models. The models then predict the

future scenario, which includes population distribution, activities performed by the population, and so on. Under this future scenario, a corresponding new plan can be generated. If the new plan is the same as the initial plan, we say that consistency is reached among these models. Otherwise, some mechanism is applied to combine the new plan and the initial one. The combined plan then acts as the initial plan for the next run. This process continues until consistency is achieved. In such a context, multiple instances of the DTA problem needs to be solved to reach consistency. Consequently, a computing time that is acceptable for a single run may not be acceptable any more.

Within DTA models, the most time consuming steps are the network loading procedure and the user route choice procedure. The network loading procedure is carried out by the dynamic network loading algorithm. The modelling of the user choice behavior requires the computation of dynamic shortest paths. Thus in order to develop faster DTA models, one needs to decrease the run time of:

- the dynamic network loading algorithm;
- the dynamic shortest path algorithm; and
- the DTA algorithm.

In general there are two ways to improve the efficiency of algorithms: 1) design more efficient serial algorithms; 2) exploit parallel computing platforms by developing parallel solution algorithms.

For the dynamic shortest path problem, we design more efficient algorithms for a class of shortest path problems, which consists of finding shortest paths from one node to all other nodes for multiple departure times. The algorithm exploits results corresponding to previous departure times, instead of solving the problems independently for each departure time.

For the dynamic network loading problem, we design parallel loading algorithms to improve its efficiency. Parallelization is exploited along two dimensions: time and network topology. Each decomposition is implemented under two parallel systems: shared-memory system and distributed-memory system.

1.3 Thesis Contributions

This thesis presents various advancements beyond previous research. Specifically,

- we presented the first parallelization of macroscopic DTA models in the literature. Two loading algorithms are investigated on both shared-memory and distributed-memory parallel computing platforms;
- we presented a new framework for static shortest path algorithms which prioritize nodes with optimal distance labels. This framework is applied in dynamic FIFO and strict FIFO networks in the one-to-all shortest path problem for all departure times to develop efficient algorithms. It is also applied in static networks in one-to-all, one-to-one, many-to-all shortest path problems to develop efficient algorithms.
- we also presented two other interesting shortest path algorithms: Algorithm Delta and Algorithm Hierarchy. Through experimental evaluation, we found that the smaller the number of nodes in the network the more effective Algorithm Delta is. We gave the runtime complexity of Algorithm Hierarchy. Using a small example in dense networks, we illustrate the effectiveness of Algorithm Hierarchy.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents the parallel implementations of DTA models. It first introduces a macroscopic DTA model with two loading algorithms, followed by parallel computing concepts required to develop the parallel implementations. Then the parallel implementations of the iterative loading algorithm and the chronological loading algorithm are presented. Finally, numerical tests in the distributed-memory platform are studied.

Chapter 3 presents a framework for static shortest path algorithms which label-sets and prioritizes the nodes with optimal distance labels. It first proposes the framework and explains the relationship between the framework and existing algorithms.

Then the framework is applied to dynamic and static shortest path problems to develop more efficient algorithms. Computer implementations and numerical results are presented.

Chapter 4 presents two ideas in the area of shortest path problem. Algorithm Delta is introduced first, followed by its experimental evaluation. We then introduce the idea of Algorithm Hierarchy in the context of dense networks. After providing the algorithmic statement of Algorithm Hierarchy for static networks, we develop a new algorithm for one-to-all shortest path problems in dynamic FIFO networks for all departure times.

Chapter 5 summarizes the main conclusions of this thesis and suggests future research directions.

Chapter 2

Parallel Implementations of DTA Models

In the past fifteen years, demand for supercomputing resources have risen sharply. There is a need for tremendous computational capabilities in many areas including large-scale network analysis, weather prediction, human genome analysis, and aircraft design. These applications are characterized by a huge amount of computation (lots of equations and variables), large data sets, and fine granularity. Parallel machines are designed to meet the computation demand from these applications. Parallel computers have evolved from experimental contraptions in laboratories to the everyday tools of computational scientists who need the ultimate computing power in order to solve their problems.

In this chapter we apply parallel computing technology to macroscopic DTA modelling to develop faster implementations of these models. The models studied in this chapter are based on the sequential model described in [5]. To the best of our knowledge, the derivations in this chapter concern the first macroscopic DTA model parallelized in the literature.

A parallel machine is a computer that has multiple processors. It can be a single machine with multiple processors or a cluster of computers that can be used to accomplish operations in parallel, which is referred to as parallel processing. To have a parallel machine work, we need:

- a load partition algorithm to decompose the original problem into sub-problems. Note that we can also view a partition algorithm as one that decomposes the algorithm for the original problem into sub-algorithms;
- algorithms and associated data structures to solve the sub-problems; and
- a communication mechanism between the processors.

This chapter focuses on the above items and is organized as follows. In Section 2.1 we give a brief description on the serial DTA model studied in this chapter. In Section 2.2 we introduce two basic parallel computational models. Section 2.3 presents three common parallel computing paradigms. Section 2.4 surveys related work in parallel computing in the field of transportation. In Section 2.5 through 2.7, we present the parallel implementations of the DTA algorithms. Numerical results are presented in Section 2.8.

2.1 A Macroscopic Dynamic Traffic Assignment Model

Dynamic Traffic Assignment (DTA), although still in a state of flux, has evolved substantially since the seminal work of Merchant and Nemhauser [19, 20]. The task of a DTA model is: given a network and an original-destination demand matrix, find a set of reasonable link flows and link travel times. The word "reasonable" can take several meanings, but the most widely used ones are: user optimal (UO) and system optimal (SO). In UO DTA each traveller is assumed to arrive at his destination as early as possible; while in SO DTA the total travel time of all travellers are is minimized.

Recall the taxonomy of DTA models in Section 1.1. There are three types of DTA models: microscopic, mesoscopic and microscopic. In this thesis, we focus on macroscopic models, which are introduced in the following text.

In He [5] a modelling framework for the DTA problem is proposed. It contains four components: a user's behavior module, a link performance module, a network

loading module, and a path generation module. Based on this framework, a DTA model was formulated. Various solution algorithms are developed, including two dynamic network loading algorithms (the iterative network loading algorithm and the chronological network loading algorithm) and one DTA algorithm. This section summarizes the algorithms used in this thesis.

2.1.1 Notation

The physical traffic network is represented as a directed network $G = (N, A)$, where N is the set of nodes and A is the set of arcs. The index m denotes the type of users, the index r denotes an origin node, the index s denotes a destination node and the index p denotes a path between O-D pair (r, s). In the implementation, three types of users are modelled: type 1 users always follow the fixed routes; type 2 users follow routes with minimum perceived travel time; and type 3 users follow routes with minimum actual travel time.

Path variables:

$f_{mp}^{rs}(t)$: departure flow rates from origin r to destination s on path p at time t for type m user.

$g_{mp}^{rs}(t)$: new departure flow rates from origin r to destination s on path p at time t for type m user.

Link-Path variables:

$u_{ap}^{rs}(t)$: entrance flow rate at time t for link a on path p from origin r to destination s.

$v_{ap}^{rs}(t)$: exit flow rate at time t for link a on path p from origin r to destination s.

$U_{ap}^{rs}(t)$: cumulative entrance flow at time t for link a on path p from origin r to destination s.

$V_{ap}^{rs}(t)$: cumulative exit flow at time t for link a on path p from origin r to destination s.

$X_{ap}^{rs}(t)$: link flow at time t for link a on path p from origin r to destination s.

Link variables:

$\tau_a(t)$: travel time over link a for flows entering link a at time t.

$\tau_a^{(n)}(t)$: travel time over link a for flows entering link a at time t in iteration n in the iterative network loading algorithm.

$X_a(t)$: link flow at time t for link a.

Time variables:

Δ : minimum free flow link travel time over all links.

$\delta = \frac{\Delta}{M}$, where M is a positive integer.

Other variables:

N_1 : the number of loops in the DTA algorithm.

N_2 : the number of loops in the iterative network loading algorithm.

T : the number of demand intervals. Time is discretized into intervals of length δ. Let T_{dmd} denote the duration of the demand period, then $T = T_{dmd}/\delta$.

2.1.2 The DTA Solution Algorithm

An iterative process is used to solve the DTA Model. The process consists of assuming an initial network condition (time-dependent link travel times). The travel demands are then assigned to the network according to UO condition, which leads to an initial values of path flows f_{mp}^{rs}. The network loading module then loads the path flows to the network. This leads to a set of new time-dependent link travel times, and thus a set of new path flows g_{mp}^{rs}. This process is called a *DTA iteration*. The set of new path flows g_{mp}^{rs} are not necessarily equal to the set of path flows used in the previous network loading procedure. There are various ways to generate another set of path flows as the input for the next DTA iteration. One approach is to use the Method of Successive Average (MSA) [21] to combine the results from the current DTA iteration with the previous one. The DTA algorithm is outlined in Figure 2-1.

DTA algorithm

Step 0: (Initialization)

$N_1 \leftarrow$ maximum number of iterations;

Compute initial path flows $f_{mp}^{rs(0)}(k)$ from free-flow path travel times;

$n \leftarrow 0$.

Step 1: (Main loop)

1.1: Perform dynamic network loading procedure;

1.2: Compute $g_{mp}^{rs}(k)$ by `Route Choice Algorithm`;

1.3: Update path flows:

$f_{mp}^{rs(n+1)}(k) \leftarrow f_{mp}^{rs(n)}(k) + \alpha^{(n)}[g_{mp}^{rs}(k) - f_{mp}^{rs(n)}(k)], \alpha^{(n)} \leftarrow \frac{1}{n+1}, m = 2, 3.$

Step 2: (Stopping criterion)

If $n = N_1$, then stop;

Otherwise, $n \leftarrow n + 1$ and go to Step 1.

Figure 2-1: Statement of the DTA algorithm

2.1.3 The Iterative Dynamic Network Loading Algorithm

The network loading model takes the path flows as input and uses the link performance model to generate the resulting link-based network conditions such as time-dependent link volumes and link travel times.

If link travel times $\tau_a(k)$ are known and do not change with network conditions, a solution to the discrete model can be found by propagating flows along the paths from origins to destinations. However, if $\tau_a(k)$ is a function of network conditions, this method may not be valid. The reason is that, using a set of fixed link travel time $\tau_a(k)$ to propagate path flows will result in a new network condition, and hence a set of new link travel times $\tau_a^{new}(k)$. The method is invalid if $\tau_a(k)$ is not equal to $\tau_a^{new}(k)$.

However, an iterative heuristic method can be used to update $\tau_a(k)$ in each iteration. If after a certain number of iterations, the set of new link travel times $\tau_a^{new}(k)$

I-Load algorithm

Step 0: (Initialization)
$N_2 \leftarrow$ maximum number of iterations;
$\tau_a^{(0)}(k) \leftarrow$ free flow travel time;
$n \leftarrow 0$.
Step 1: (Main loop)
1.1: Move departure flows according to $\{\tau_a^{(n)}(k)\}$;
1.2: Compute the resulting total link volumes $\{X_a(t)\}$;
1.3: Compute the new link travel times $\{\tau_a^{new}(k)\}$;
1.4: Update link travel times:
$\tau_a^{(n+1)}(k) \leftarrow \tau_a^{(n)}(k) + \alpha^n(\tau_a^{new}(k) - \tau_a^{(n)}(k))$;
$\alpha^n \leftarrow \frac{1}{n+1}$.
Step 2: (Stopping criterion)
If $n = N_2$, then stop;
Otherwise, $n \leftarrow n + 1$ and go to Step 1.

Figure 2-2: Statement of the I-Load algorithm

converge to a set of link travel times $\tau_a^*(k)$, a solution is found. MSA is adopted in the averaging.

The iterative loading algorithm (I-Load) is based on the idea and is outlined in Figure 2-2. The I-Load algorithm is a heuristic. Computational examples in [5] show that I-Load algorithm generates similar path travel time pattern to C-Load algorithm which is introduced in the following section. However the path travel time from the I-Load algorithm is lower than that from the C-Load algorithm.

To analyze complexity of I-Load algorithm, we introduce some additional notations:

P : the set of all paths between all OD pairs.;

P_i : the i^{th} path in P;

P_i^A : the set of arcs path P_i passes through.

The runtime complexity of I-Load algorithm is:

$$O(N_2 T \sum_{i=1}^{|P|} |P_i^A|)$$

There are two possible parallelization strategies for the I-Load algorithm. One is to decompose the demand period T into several sub-periods. This corresponds to a time-based decomposition strategy. The other is to decompose $\sum_{i=1}^{|P|} |P_i^A|$. This corresponds to a network topology based decomposition strategy. The two strategies are presented in detail in Section 2.5.

2.1.4 The Chronological Dynamic Network Loading Algorithm

Instead of solving the network loading problem iteratively, one can find a network loading solution in chronological order, that is, finding a solution within each time interval $(i\delta, (i+1)\delta]$ in increasing order of time index i until there is no demand to load and the network is empty. This algorithm is the chronological loading algorithm (C-Load). Let T_{em} denote the minimum number of Δ intervals required for the network to become and remain empty. The algorithm is outlined in Figure 2-3.

C-Load algorithm

Step 0: (Initialization)
Determine Δ by $\Delta \leftarrow min_a[\tau_a(0)]$;
M: the number of δ intervals within a Δ interval;
$i \leftarrow 0$.
Step 1: (Solve the equations within i^{th} Δ interval)
1.1: for ($a \in A$ and p passing through a) do
for $k = iM$ to $(i+1)M - 1$ do
Compute $V_{ap}^{rs}(k)$;
Compute $v_{ap}^{rs}(k)$;
1.2: for ($a \in A$ and p passing through a) do
for $k = iM$ to $(i+1)M - 1$ do
Compute $u_{ap}^{rs}(k)$;
Compute $U_{ap}^{rs}(k)$;
Compute $X_{ap}^{rs}(k)$;
Compute $\tau_a(k)$.
Step 2: (Stopping criterion)
If the network is empty and there is no demand to load, then stop;
Otherwise, $i \leftarrow i+1, T_{em} \leftarrow i\Delta$ and go to Step 1.

Figure 2-3: Statement of the C-Load algorithm

Let a_i^{PATH} denote the set of paths that pass through arc i. Whenever Step 1 is executed, calculations are performed over $O(\sum_{i=1}^{|A|} |a_i^{PATH}|)$ variables. Therefore the runtime complexity is:

$$O(T_{em} M \sum_{i=1}^{|A|} |a_i^{PATH}|)$$

A consolidation method was used to reduce the number of link-path flow variables. All link-path flow variables belonging to a link need to be processed at each δ interval. Since the number of link-path flow variables of a link equals the number of paths that pass through that link, it may be very large if many paths share this link. The idea of consolidation comes from the observation that it is not necessary to distinguish the path flows which have the same destination, follow the same subpath from a link to that destination and correspond to the same time at the current link. The LPV data structure is introduced to carry out the consolidation procedure. Interested readers can refer to [5] for the consolidation algorithm. Let a_i^{LPV} denote the set of LPV objects in link i. We have $|a_i^{LPV}| \leq |a_i^{PATH}|$. The complexity of this consolidated C-Load algorithm is:

$$O(T_{em} M \sum_{i=1}^{|A|} |a_i^{LPV}|)$$

One possible parallel strategy is to decompose $\sum_{i=1}^{|A|} |a_i^{LPV}|$. This corresponds to the network topology decomposition strategy. The parallel algorithms and implementations are in Section 2.7.

2.2 Parallel Computing Systems

The exact definition of a parallel machine is still open for debate: for any definition there will always be examples that are inappropriately included in the definition,

or excluded from it. We define a parallel machine as a computer that has multiple processors or a cluster of computers being used to accomplish operations in parallel.

There are many ways to categorize parallel computing systems. In existing parallel models, processors communicate in essentially two different ways: (1) through the reading from and writing to a shared global memory, or (2) by passing messages through links connecting processors that do not have a common memory. This leads to two different computing systems: shared-memory system and distributed-memory system.

Shared-memory System In this system, each processor has access to a single, shared-memory. Processors communicate by reading from and writing to the same physical location in the shared memory. A difficult aspect in programming on such systems is the so-called non-determinism: the result of a computation may differ depending on how fast each of the processes manages to write to the memory. Usually such non-determinism is undesired, because the result may differ from one run to the other. Hence the processes must coordinate memory access in some way. A schematic diagram of this model is shown in Figure 2-4. The development of code under the shared-memory model is usually done using multi-threaded techniques. The shared-memory implementations in this thesis were developed using POSIX threads [22, 23], which is based on the POSIX Standard [24].

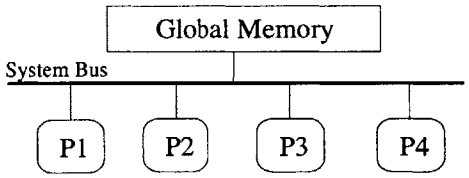

Figure 2-4: Parallel computing systems: shared-memory

Distributed-memory System For technical reasons, the number of processors in a shared memory system cannot grow without bounds. It is so difficult to go beyond 16 processors in a shared-memory system and a different approach must be sought if one is interested in computers with a larger number of processors. The solution is to

use a distributed memory system. In such a system, each processor has its own memory space. The processors exchange information by sending messages to each other (Figure 2-5) . Data transfer from the local memory of one process to the local memory of another process requires operations performed by both processes. In this kind of model, processors typically communicate through a communication software library. The Message Passing Interface (MPI) [25, 26] is an emerging standard specification for message-passing libraries. The distributed-memory implementation of this thesis is developed using MPICH (1.2.4) [27], which is a portable implementation of the full MPI specification for a wide variety of parallel computing environments. Note that MPICH can also run on shared-memory systems.

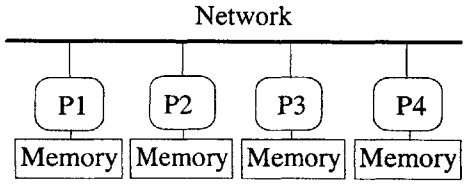

Figure 2-5: Parallel computing systems: distributed-memory

Besides the two extreme systems, hybrid systems also exist as well. In this case shared-memory systems are interconnected through networks. The cluster of processors that share memory can communicate with each other using the share-memory model; however, they need to pass messages to communicate with processors in other clusters (Figure 2-6).

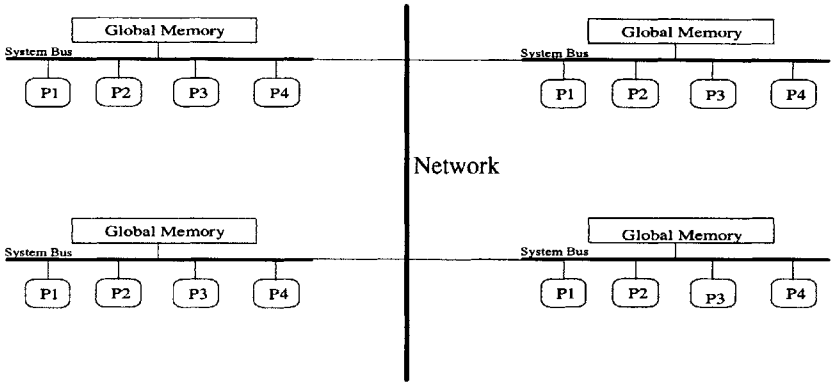

Figure 2-6: Parallel computing systems: hybrid

2.3 Parallel Programming Paradigms

In the beginning of this chapter, we note that in order to accomplish parallel processing, we need to decompose the original problem into several subproblems. Each process or thread works on one subproblem and contributes to the final result. There are many ways regarding how processes or threads work with each other, but there are three primary paradigms: pipeline, work crew, and master/slave.

These paradigms are valid for both shared-memory and distributed-memory models. For the ease of explanation, we assume a distributed-memory model. A shared-memory counterpart explanation can be obtained by substituting the term "thread" for "process" in the following text.

Pipeline In this paradigm, each process repeatedly performs the same operation on a sequence of data sets, passing the results to the next processor for the next step. This is also known as an "assembly line". Different processes often carry out different operations. For example, suppose there are a set of images to be examined for certain patterns. Process 1 scans one image; Process 2 searches the scanned image for the pattern; and Process 3 collects the search results from Process 2 into a report. Process 1 keeps scanning; Process 2 keeps searching; and Process 3 keeps collecting until the whole set of images are processed.

Work Crew In this paradigm, each process work on its own data. They may or may not all execute the same operation. A parallel decomposition of a loop usually falls into this model. For instance in matrix processing, a set of processes can be created to work on the columns of a large matrix. The matrix is divided into several column blocks and each is assigned to a process. The processes in the work crew carry out the same operation on their assigned columns. This is always called Single Instruction Multiple Data (SIMD) parallel processing, because all processes perform the same operation on different data.

The work crew model is not limited to SIMD parallelism. The processes may perform completely different operations on different data. This is often named Multiple

Instruction Multiple Data (MIMD) parallel processing.

The work crew model is used in the development of parallel algorithms in this thesis.

Master/Slave In the master/slave paradigm, there is one master process and many slave processes. The master process takes care of I/O, decomposes the problem into subproblems and assign them to the slave processes. The slave processes work in parallel and return their partial results to the master. The master/slave paradigm is used in the development of parallel algorithms in this thesis as well.

2.4 Related Work

Early use of parallel computing technology in the area of transportation includes the parallelization of fluid-dynamical models for traffic [28], parallel implementation of assignment models [29] and shortest path algorithms [30, 31, 32].

The parallel implementation of DTA models is a relatively new area in the field of transportation. Parallel DTA models can be found in [33, 34]. A recent development is the parallel implementation of TRANSIMS [4]. All these are either microscopic or mesoscopic DTA models. We briefly review the parallel implementation of TRANSIMS in Section 2.4.1 and the parallel implementation of Algorithm DOT in Section 2.4.2.

2.4.1 Parallel Implementation of TRANSIMS

The TRansportation ANalysis and SIMulation System, or TRANSIMS, is an integrated system of travel forecasting models developed in Los Alamos National Laboratory. It is a microscopic traffic simulator and uses the cellular automata (CA) for the representation of driving dynamics. The roads are divided into cells whose length is about 7.5 meters (the length a typical car occupies during traffic jam). At any moment each cell can be occupied by at most one vehicle. The movement of vehicles is represented by hopping from one cell to another. Different hopping distance during

one time step is designed for different vehicle speed, which is determined from driving behavior emulation.

The most significant advantage of CA is the facilitation of a parallelization based on network topology decomposition, because the state at time step t depends only on information from time step $t-1$, and only from neighboring cells. The information exchange along boundaries is carried out at the end of time step $t-1$. Each processor updates the vehicle position in its subnetwork for time step t, and the exchange along boundaries is carried out again.

The network is partitioned at the middle of links rather than nodes to reduce the complexity introduced by intersections. Each link is represented in both processors, but each processor is responsible for only one half of the link. In order to generate consistent driving behavior on boundary links, each processor sends the first five cells' information on its side to the processor on the other side. The length of five cells is defined as the interactive distance in the CA. By doing so, the other process has enough information on what is happening in the other half of the link to compute consistent driving behavior.

In the implementation described in [4], the master/slave paradigm is used. The master process decomposes the workload, spawns the slave processes, and controls the general scheduling of all slave processes. The network is partitioned using METIS [35]. During the course of computation, adaptive load balancing is applied from iteration to iteration. During run time the execution time of the simulation is collected and fed back to the partitioning algorithm in the next run.

The results show that with a common technology - 100 Mbit/sec switched Ethernet - one can run the 20 000-link EMME/2-network for Portland (Oregon) more than 40 times faster than real time on 16 dual 500 MHz Pentium computers.

2.4.2 Parallel Implementation of Transportation Related Algorithms

Chabini and Ganugapati [32] explored various ways to parallelize the computation of dynamic shortest paths. Algorithm DOT [36] solves the all-to-one dynamic fastest paths problem for all departure times with optimal worst case complexity. One way to speed up the computation of algorithm DOT is through parallel computing.

In [32] two parallelization strategies are developed: decomposition by destination and decomposition by network topology. The destination decomposition strategy decomposes the all-to-many dynamic shortest paths problem into several smaller all-to-many problems by dividing the set of destinations into subsets. The network topology decomposition strategy splits the network into subnetworks and assign subnetworks to processors. This strategy is at the algorithm level and involves the re-writing of the algorithm.

Two parallel libraries are used for each parallel strategy: Parallel Virtual Machine (PVM) and Solaris Multithreading. Numerical results are obtained using large-size dynamic networks and two parallel computing systems: a distributed network of Unix workstations and a SUN shared-memory machine containing eight processors. The results show that a speedup of about 6 is achieved in shared memory systems using 6 processors. The shared-memory system appears to be the most appropriate type of parallel computing systems for the computation of dynamic shortest paths for real-time ITS applications.

2.5 Parallel Implementations of the I-Load Algorithm: Network Topology Decomposition

In this section we present the details of the network topology decomposition strategy for the iterative loading algorithm. Network topology decomposition is usually done at the link level, which means a link is the basic element in the partitioning. However, the **MoveFlow()** procedure in the I-Load algorithm is essentially path based, therefore a

compromised network topology decomposition is adopted, where we split the OD pairs into subsets and assign them to different processors. This decomposition strategy is implemented in both the shared-memory and the distributed-memory environments.

2.5.1 Load Partition Algorithm

In addition to the notation introduced in Section 2.1.1, we introduce the following notation for the parallel implementations:

NP	:	the total number of processes;
G_i	:	the set of OD pairs assigned to process or thread i;
G_i^P	:	the set of paths in G_i;
G_i^A	:	the set of links covered by $OD_i \in G_i$. If OD_i^A and OD_j^A $(i \neq j)$ share a common link, that link should be counted twice;
OD	:	the set of all OD pairs;
OD_i	:	OD pair i; and
OD_i^A	:	the set of links covered by all the paths between OD_i. If two paths share a common link, that link should be counted twice.

Network-topology based decomposition means that the geographical region is decomposed into several sub-networks of similar size, and each processor of the parallel computer carries out the network loading procedure for one of these sub-networks. The term "similar size" means that the computational effort of each sub-network is similar.

For microscopic DTA models, the computational effort required for each vehicle is the same. If we assume a uniform spacial distribution of vehicles in the network, a realistic measurement for size is the total length of all streets associated with the sub-network [4].

For the I-Load network loading procedure, flows are represented as fractions and sent along paths by the **MoveFlow()** procedure. The procedure is briefly illustrated in Figure 2-7. Node r is the origin node, and node s is the destination node. The amount of departure flow is $f_p^{rs}(t)\delta$. For each departure time t_0, we calculate the

travel time $t^{(r,1)}(t_0)$ along link $(r,1)$. We add $f_p^{rs}(t)\delta$ to $X^{rs}_{(s,1)p}(t')$ for t' such that $t_0 \leq t' < t_0 + t^{(r,1)}(t_0)$. We take $t_0 + t^{(r,1)}(t_0)$ as the entering time for link $(1,2)$, calculate the travel time $t_t^{(1,2)}$ along link $(1,2)$. We add $f_p^{rs}(t)\delta$ to $X^{rs}_{(1,2)p}(t')$ for t' such that $t_0 + t^{(r,1)}(t_0) \leq t' < t_0 + t^{(r,1)}(t_0) + t^{(1,2)}(t_0 + t^{(r,1)}(t_0))$. The previous step is repeated until we reach the last link (n,s).

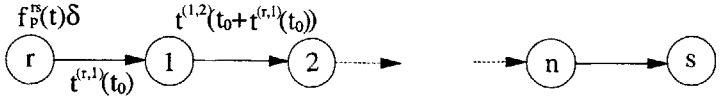

Figure 2-7: Example for **MoveFlow()**

It is seen that the computational effort for each link along a path is the same. Therefore a measure of size for the macroscopic DTA model based on the I-Load algorithm is the total number of links in different origin-destination paths. Note that if path 1 and path 2 share a common link, we need to count that link twice.

METIS [35] is a software package available in the public domain to partition graphs. The algorithm in METIS is based on multilevel graph partition. The graph is first coarsened and partitioned. Then it is uncoarsened and at every uncoarsening step an exchange algorithm is used. The coarsening can be done via random matching. The first set of links in the network are selected, such that no two links are incident to each other. Then the end nodes of these selected links collapse into one. It is easier to find an optimal partitioning when the number of nodes is sufficiently small. In the process of uncoarsening, METIS systematically tries whether exchanges of nodes at the boundaries lead to better solutions.

Our approach to network decomposition is through origin-destination decomposition. The reason is that the **MoveFlow()** algorithm is path based. While the paths between OD pairs in the network are very likely to share links, the subnetwork generated by METIS will probably cut through the middle of a path. This is incompatible with the **MoveFlow()** algorithm. We first count the number of links in the paths between each OD pair (If a link is shared by two paths, it should be counted twice.). Then we try to group the OD pairs into several OD pair groups, such that each group contains almost the same number of links.

We first sort all OD pairs $\{OD_i\}$ by the number of arcs covered by the paths between OD pair OD_i in increasing order as $OD'_1, OD'_2, \cdots, OD'_{|OD|}$ ($|OD'^A_1| \leq |OD'^A_2| \leq \cdots \leq |OD'^A_{|OD|}|$). Then we compute the average number of arcs denoted as avg for each group. The load partition starts by first assigning OD pair $OD'_{|OD|}$ to group 1, that is, $G_1 \leftarrow G_1 \cup \{OD'_{|OD|}\}$. If $|G^A_1| \geq avg$ we stop the assignment for group 1; otherwise, we add OD pair OD'_1 to group 1, that is, $G_1 \leftarrow G_1 \cup \{OD'_1\}$. If $|G^A_1| \geq avg$, we stop the assignment for group 1; otherwise, we continue to add OD pairs $OD'_3, OD'_4, \cdots$, to group 1 until $|G^A_1| \geq avg$. Suppose that the last OD pair added to group 1 is OD'_k. Then the load partition algorithm assigns OD pair $OD'_{|OD|-1}$ to group 2, that is $G_2 \leftarrow G_2 \cup \{OD'_{|OD|-1}\}$. If $|G^A_2| \geq avg$, we stop the assignment for group 2; otherwise, we continue to add OD pairs $OD'_{k+1}, OD'_{k+2}, \cdots$, until $|G^A_2| \geq avg$. Then the load partition algorithm assigns OD pairs to group 3, 4, $\cdots$, NP. The partitioning algorithm is shown in Figure 2-8.

```
LOAD PARTITION ALGORITHM ()
1   sort all OD pairs by |OD^A_k| in increasing order: OD'_1, OD'_2, ..., OD'_|OD| ;
2   G_k ← ∅ (k = 1, 2, ..., NP) ;
3   avg ← (Σ_{k=1}^{|OD|} |OD^A_k|) / NP ;
4   i ← |OD|, j ← 0, np ← 1 ;
5   while i > j do
6     G_np ← G_np ∪ {OD'_i} ;
7     i ← i − 1 ;
8     while j < i AND |G^A_np| < avg do
9       j ← j + 1 ;
10      G_np ← G_np ∪ {OD'_j} ;
11    endwhile
12    np ← np + 1 ;
13  endwhile
```

Figure 2-8: Load partition algorithm for I-Load algorithm for network decomposition strategy

We compute the average number of arcs for each group in Line 3 in Figure 2-8 and use the average throughout the whole algorithm. A potential problem of this is that $|G^A_k|$ might be substantially less than the average when k is large. An improvement can be made to address this problem as follows. Instead of using the average computed in Line 3 throughout the algorithm, we update the average whenever we finish the

assignment of a group, that is, add the statement

$$avg \leftarrow \frac{\sum_{k=j+1}^{k=i} |OD_k^A|}{NP - np}$$

between Line 11 and 12.

The origin-destination decomposition has no impact on the distributed-memory implementation; however it may affect the shared-memory implementation. Because if the two paths that pass through a link are assigned to two threads in the shared-memory implementation, there is a chance that there would be memory write conflicts. In the distributed-memory implementation, this is not an issue.

2.5.2 Shared-memory Implementation

In the shared-memory implementation, the network and the link-path variables are stored in the global memory, where each thread can have access. The master/slave paradigm is used in the shared-memory implementation. The master thread (main) reads the input file and prepares for the parallelism. The master thread algorithm is in Figure 2-9.

```
MASTER THREAD ALGORITHM ()
1   read the network G(N, A) and time-dependent OD demands ;
2   group OD's into subsets G_i using LOAD PARTITION ALGORITHM ;
3   for i ← 1 to NP do
4     create slave thread i ;
5   endfor
6   wait for all slave threads to join ;
7   output and stop.
```

Figure 2-9: Master thread algorithm for I-Load algorithm for network decomposition strategy

The load partition algorithm stated in Figure 2-8 provides a load partition with the aim to balance the workload caused by the **MoveFlow()** procedure. However, it may not provide a good load partition for workload induced by other procedures, for example, the procedure to compute link total volumes and travel times. In a shared-memory implementation, we can adapt different load partition alternatives

for different procedures. Moreover, if the load partition algorithm is well designed, it improves the efficiency of the corresponding parallelized procedure without paying any penalty. In view of this, we used a better load partition for the computation of total link volumes, new link travel times, and the update of link travel times. We assume that A, which is the set of arcs in G, has been partitioned into NP mutually exclusive and collectively exhaustive sets: $A_1, A_2, \cdots, A_{NP}$, such that $\lfloor \frac{|A|}{NP} \rfloor \leq |A_i| \leq \lceil \frac{|A|}{NP} \rceil$.

The slave thread algorithm is shown in Figure 2-10. The DTA algorithm and the I-Load algorithm are integrated in the thread algorithm. The function **Barrier** blocks the caller until all threads have called it; the call returns at any thread only after all threads have entered the call.

```
SLAVE THREAD ALGORITHM ()
1   i ← GETTHREADID ;
2   compute initial path flows f_mp^{rs(0)}, ∀(r,s) ∈ G_i ;
3   for n_1 ← 0 to N_1 do
4     compute initial link free flow travel times τ_a^0(k), ∀a ∈ A_i ;
5     for n_2 ← 0 to N_2 do
6       move departure flows f_mp^{rs(0)}, ∀(r,s) ∈ G_i according to τ_a^{n_2}(k) ;
7       BARRIER ;
8       for a ∈ A_i do
9         compute the total link volumes X_a^{rs} ;
10        compute the new link travel times τ_a^{new}(k) ;
11        update link travel times to τ_a^{n_2+1}(k) ;
12      endfor
13    endfor
14    BARRIER ;
15    compute g_mp^{rs}(k), ∀(r,s) ∈ G_i by ROUTE CHOICE ALGORITHM ;
16    update path flows to f_mp^{rs(n_1+1)}, ∀(r,s) ∈ G_i ;
17  endfor
```

Figure 2-10: Slave thread algorithm for I-Load algorithm for network decomposition strategy

Now we analyze the advantages and disadvantages of the load partition algorithm shown in Figure 2-8 to METIS for the shared-memory implementation. The example network is shown in Figure 2-11. There are 2 OD pairs 1-6 (path 1-3-4-5-6) and 2-7 (path 2-3-4-6-7). Because METIS is not able to recognize the paths in the network, it partitions the network as shown in Figure 2-12. Thread 1 and Thread 2 work on different sets of links, therefore there will be no memory access conflict. However,

Thread 2 will idle until Thread 1 has moved flow across the boundary. The two threads are not synchronized.

The Load Partition Algorithm shown in Figure 2-8 will partition the network as shown in Figure 2-12. We duplicate Node 3, 4, and 5 to show that both threads will work on link (3,4) and (4,5). Memory write conflicts may arise in this case; however, both threads are able to start working at the beginning independently of each other. The problem of synchronization is solved.

One can see that each approach has its advantages and disadvantages. Which approach has a better performance remains to be investigated. Since Network Partition Algorithm shown in Figure 2-8 involves less restructuring of the original code, we adopt it in our development of the parallel code.

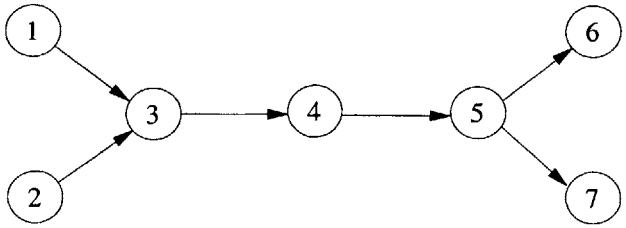

Figure 2-11: An example network with two OD pairs

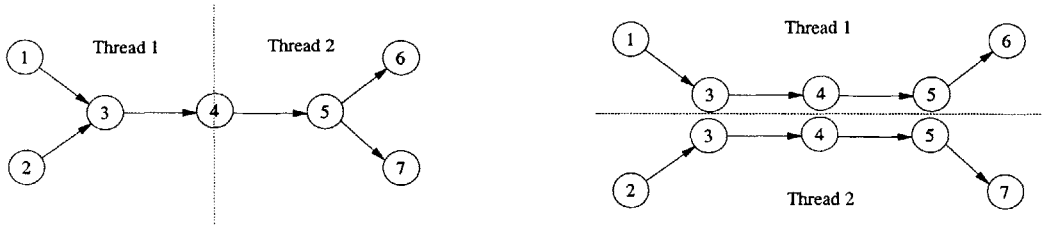

Figure 2-12: How METIS (left) and the Network Partition Algorithm (right) partition the network

2.5.3 Distributed-memory Implementation

In the distributed-memory implementation, each process has its own copy of the network. We use the work crew paradigm rather than the usual master/slave paradigm

for the following reason. In the distributed-memory implementation, we need to explicitly pass messages between processes to update their copies of the network. If we have a master process which handles the I/O, decomposes the original problem, assembles the partial network information and distribute to the slave processes, we have to note that it actually stays idling when the slave processes are working. Another fact is that the larger the number of processes, the larger the communication time because more slave processes need to be updated. If we need NP processes in the master/slave paradigm (1 master process and $NP-1$ slave processes), we only need $NP-1$ processes in the work crew paradigm, which reduces the communication burden because less computation nodes are involved in collective communications, say, broadcast. The algorithm is shown in Figure 2-13. Note that we do not explicitly implement a barrier to keep the processes synchronized, because the **MPI_Allreduce()** function will implicitly have all processes wait until all processes obtain the collective message. **MPI_Allreduce()** performs a global reduce operation (such as sum, max, logical AND, etc.) across all the processes. Details on **MPI_Allreduce()** can be found in [26].

```
PROCESS ALGORITHM ()
1  i ← GETPROCESSID ;
2  read the network G(N, A) and time-dependent OD demands ;
3  run NETWORK PARTITION ALGORITHM. G_i is the subset of ODs for process i ;
4  compute initial path flow f_mp^{rs(0)}(t), ∀(r,s) ∈ G_i, ∀m, ∀p ∈ K_rs ;
5  for n_1 ← 1 to N_1 do
6    for n_2 ← 1 to N_2 do
7      MOVEFLOW for ∀(r,s) ∈ G_i ;
8      use MPI_ALLREDUCE() to receive X_ap^rs(t), ∀(r,s) ∈ G, ∀p ∈ K_rs, ∀a ∈ p^a ;
9      obtain link flow totals X_a(t), ∀a ∈ G_i^a ;
10     compute the new link travel times τ_a^new(k), ∀a ∈ G_i^a ;
11     update link travel times τ_a^{(n_2+1)}(k), ∀a ∈ G_i^a ;
12   endfor
13   compute g_mp^rs(k), ∀(r,s) ∈ G_i, ∀m, ∀p ∈ K_rs by ROUTE CHOICE ALGORITHM ;
14   compute the new path flow f_mp^{rs(n_1+1)}(k), ∀(r,s) ∈ G_i, ∀m, ∀p ∈ K_rs ;
15 endfor
```

Figure 2-13: Process algorithm for I-Load algorithm for network decomposition strategy

To reduce the communication time, we applied two techniques. The communica-

tion time T_{comm} required by sending a message of the size S can be expressed as:

$$T_{comm} = T_{latency} + \frac{S}{b_{network}}$$

where $T_{latency}$ is the time needed to initiate the communication. Usually it is independent of the size of the message. $b_{network}$ is the bandwidth between the sender and the receiver. $b_{network}$ is determined by the minimum of the bandwidth of the network and the bandwidth of the network interface card (NIC) of the computation nodes.

We take two measures to decrease T_{comm}: 1) increase $b_{network}$. We used Myrinet instead of the common 100Mbit/sec ethernet to construct the communication network. Myrinet offers full duplex 2 gigabit/second connection between computation nodes; 2) pack small messages into a single message, therefore reduce the number of messages we need to initiate.

These two measures are adopted in all the distributed implementations throughout this thesis.

2.6 Parallel Implementations of the I-Load Algorithm: Time-Based Decomposition

From the statement of I-Load algorithm, one can see that prior to each loading iteration, the time-dependent link travel times are known. Therefore within each iteration, instead of loading path flows from interval 1 to interval T sequentially, we could load path flows in interval $[1, \frac{T}{NP})$, path flows in interval $[\frac{T}{NP}, \frac{2T}{NP})$, $\cdots$, and path flows in interval $[\frac{(NP-1)T}{NP}, T]$ in parallel. The time-based decomposition implementation is based on this idea.

2.6.1 Load Partition Algorithm

Before we describe the load partition algorithm, we introduce some notation:

T^i_{start} : the start time interval of the demand period assigned to process i;

T^i_{end} : the end time interval of the demand period assigned to process i; and

T^i : the demand interval $[T^i_{start}, T^i_{end}]$.

The advantage of time-based decomposition is the ease of load balancing. We just need to decompose the demand period T into equal periods. The load partition algorithm is shown in Figure 2-14. The algorithm needs to be run locally on each processor and it will compute the demand period assigned to that processor.

```
LOAD PARTITION ALGORITHM ()
1   i ← GETPROCESSID ;
2   avg ← T/NP + 1 ;
3   T^i_start ← i × avg ;
4   T^i_end ← (i + 1) × avg − 1 ;
5   if T^i_end > T then
6     T^i_end ← T ;
7   endif
```

Figure 2-14: Load partition algorithm for I-Load algorithm for time-based decomposition strategy

2.6.2 Distributed-memory Implementation

Again the work crew paradigm is used in this model. The process algorithm is shown in Figure 2-15.

PROCESS ALGORITHM ()

1 $i \leftarrow$ GETPROCESSID ;
2 read the network $G(N, A)$ and time-dependent OD demands ;
3 LOAD PARTITION ALGORITHM ;
4 compute initial path flow $f_{mp}^{rs(0)}(t), \forall t \in T^i$;
5 **for** $n_1 \leftarrow 1$ to N_1 **do**
6 **for** $n_2 \leftarrow 1$ to N_2 **do**
7 MOVEFLOW for $\forall t \in T^i$;
8 use MPI_ALLREDUCE() to receive the link flows $X_{ap}^{rs}(t), \forall t, \forall rs, \forall a, \forall p$;
9 compute link flow totals $X_a(t), \forall t, \forall a \in G$;
10 compute the new link travel times $\tau_a^{new}(k), \forall t, \forall a \in G$;
11 update link travel times $\tau_a^{(n_2+1)}(t), \forall t, \forall a \in G$;
12 **endfor**
13 compute $g_{mp}^{rs}(t), \forall t \in T^i, \forall rs, \forall m, \forall p$ by ROUTE CHOICE ALGORITHM ;
14 compute the new path flow $f_{mp}^{rs(n_1+1)}(t), \forall t \in T^i, \forall rs, \forall m, \forall p$;
15 **endfor**

Figure 2-15: Process algorithm for I-Load algorithm for time-based decomposition strategy

2.6.3 Shared-memory Implementation

Although we did not implement the shared-memory implementation, we provide the algorithms in this section. The shared-memory implementation is quite similar to the distributed-memory implementation. The master/slave paradigm is used, where the master thread (main) takes care of I/O and starts the slave threads. Slave threads move flow for their own demand period and joins the master thread.

The difference between the distributed-memory and the shared-memory implementations is that in the distributed-memory implementation, we need to use `MPI_Allreduce`

to explicitly form a full picture of the network for each process; however, in the shared-memory implementation, the global memory stores the network and can let all threads to have access to. The extra cost we pay is the possible memory write conflict.

The master thread algorithm is shown in Figure 2-16. The slave thread algorithm is shown in Figure 2-17.

```
MASTER THREAD ALGORITHM ()
1   read the network G(N, A) and the time-dependent OD demands ;
2   for i ← 1 to NP do
3     create slave thread i ;
4   endfor
5   wait for all slave threads to join ;
6   stop and output the result.
```

Figure 2-16: Master thread algorithm for I-Load algorithm for time decomposition strategy

```
SLAVE THREAD ALGORITHM ()
1   i ← GETTHREADID ;
2   LOAD PARTITION ALGORITHM ;
3   compute initial path flow f_mp^rs(0)(t), ∀t ∈ T^i ;
4   for n_1 ← 1 to N_1 do
5     for n_2 ← 1 to N_2 do
6       MOVEFLOW for ∀t ∈ T^i ;
7       compute link flow totals X_a(t), ∀t, ∀a ∈ G ;
8       compute the new link travel times τ_a^new(k), ∀t, ∀a ∈ G ;
9       update link travel times τ_a^(n_2+1)(t), ∀t, ∀a ∈ G ;
10    endfor
11    compute g_mp^rs(t), ∀t ∈ T^i, ∀rs, ∀m, ∀p by ROUTE CHOICE ALGORITHM ;
12    compute the new path flow f_mp^rs(n_1+1)(t), ∀t ∈ T^i, ∀rs, ∀m, ∀p ;
13  endfor
14  join the master thread.
```

Figure 2-17: Slave thread algorithm for I-Load algorithm for time decomposition strategy

2.7 Parallel Implementations of the C-Load Algorithm: Network Topology Decomposition

The C-Load algorithm is a link based algorithm: each link has its own set of variables, which makes it naturally a good example for network topology decomposition. We divide the network into subnetworks by cutting at the nodes of the arcs as shown in Figure 2-18. Figure 2-19 presents the partition at a more detailed level, where we can see how the linkages between LPV objects are cut.

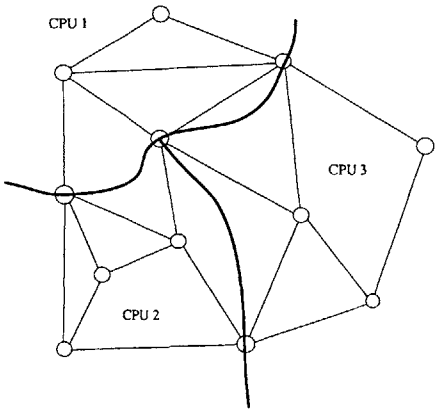

Figure 2-18: Example: network decomposition

In distributed memory implementation, each process only has a subset of the network. The links on the boundaries are stored on the processors on both sides of the boundary. After each Δ interval, we need to synchronize the variables for the boundary links (This statement is not precise. We use it here just to illustrate the idea. In Section 2.7.2 we detail how the communication need is identified and carried out.).

In shared memory implementation, the network is stored in the global memory, where each thread have access. The flow propagates along a path by the manipulation on the History lists along the path. History List is a circular queue, which stores the historic values of entrance flow rates. Each element of the list stores one past value $u_{ap}^{rs}(j)$, and contains a pointer to the next element in the list. Two pointers *tail* and *head* point to the tail and the head of the circular queue list, respectively. Special

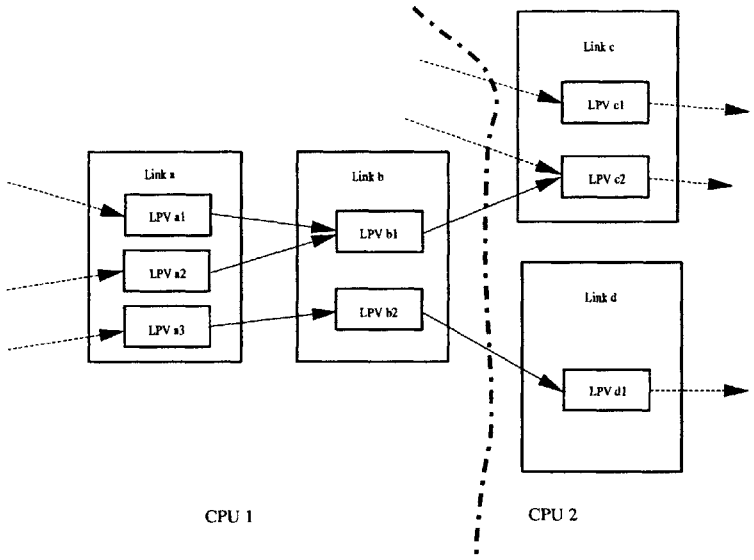

Figure 2-19: Network partition showing at LPV level

measure is needed to protect the History list to avoid any reading and writing conflicts.

2.7.1 Load Partition Algorithm

We define the diameter of a network as the maximum of the shortest path distances between any two nodes when assuming all link lengths are 1. In a distributed environment, due to the slow processor to processor communication speed, it is usually desirable to partition the network into subnetworks such that: 1) the diameter of each subnetwork is small; 2) the total length of the boundaries of the subnetworks is small; and 3) the workload (in other word, weight) in each subnetwork is similar.

We explain the above considerations as follows. Define link density as the number of links in a network divided by the area of the network. We assume that the link density conforms to a spatial uniform distribution. As we have said earlier, in the chronological loading algorithm the workload associated with a network is proportional to the number of links in the network. Therefore 1) and 3) indicate that the subnetworks are similar in shape.

Since the total amount of communication is proportional to the number of links across the boundaries, 2) is designed to reduce the amount of communication. Among

all planar shapes with the same area the circle has the shortest perimeter [37]. Therefore 2) indicates that in the best case, the subnetworks should be in a circular shape, or look like a regular polygon.

Take the network in Figure 2-20 as an example. The side of the square is 1. We want to decompose the network into 4 subnetworks. Note that for simplicity we do not show the links in the network; however, we know that the link density is uniform. In Figure 2-21 we partition the network into 4 squares of equal size. In Figure 2-22 we partition the network into 4 stripes of equal size. It is observed that the total length of the boundaries between subnetworks for Figure 2-22 is 3; while that for Figure 2-21 is 2. This is because the diameter of the subnetworks in Figure 2-21 is smaller than that in Figure 2-22.

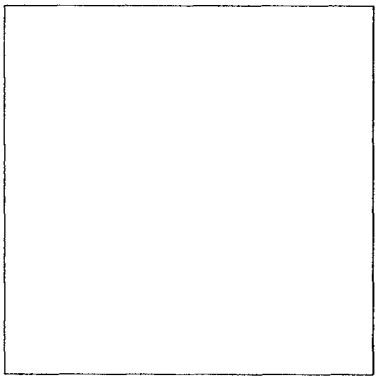

Figure 2-20: The original network

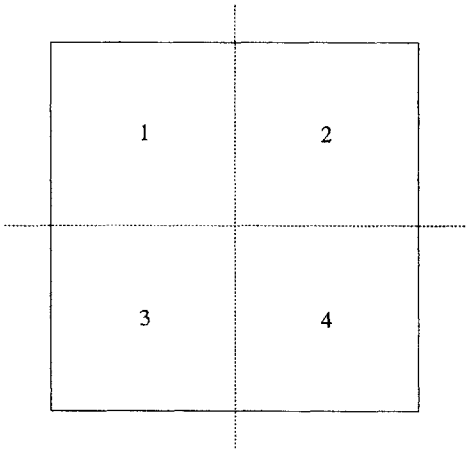

Figure 2-21: Partition method 1

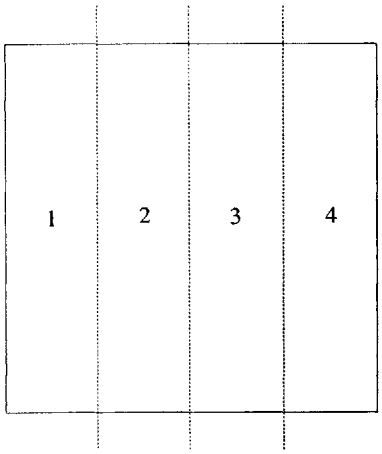

Figure 2-22: Partition method 2

Existing popular partition software such as METIS is applicable only in undirected graphs, that is, graphs in which for each link (u, v) there is also an link (v, u)) [38]. Apparently this is not the case in transportation networks. As a heuristic, we propose a link-based breadth-first search algorithm to partition directed weighted graphs.

The advantage of the ordinary breadth-first search algorithm introduced in [39] is that it makes sure that the diameter of the graph as small as possible starting from the source node. The algorithm searches all the nodes in a network that are reachable from the source. All nodes have two states: *marked* (reachable from the source) or *unmarked* (not yet determined). An arc is called an *admissible* arc if its tail node is marked and its head node is unmarked. Initially only the source node is marked. The algorithm fans out the admissible arcs of the source node and mark the head nodes of the admissible arcs. It then subsequently select a marked node and fans out its admissible arcs until no marked nodes have admissible arcs. In breadth-first search the marked nodes are stored in a queue and nodes are always selected from the front and added to the rear. What is obtained from the ordinary breadth-first search algorithm is a tree covering all nodes (We assume that the graph is strongly connected, that is, it contains at least one directed path from every node to every other node.).

In order to cover all the links in the network, we modify the ordinary breadth-first search algorithm as follows:

```
MODIFIED BREADTH-FIRST SEARCH ((u, v), w, A_sub)
1   mark link (u,v);
2   A_sub ← ∅ ;
3   LIST ← {(u, v)} ;
4   while LIST ≠ ∅ and W_Asub ≤ w do
5     if LIST ≠ ∅ then
6       select a link (i,j) from the front of LIST
7     else
8       select an unmarked link (i,j) in A ;
9     endif
10    A_sub ← A_sub ∪ {(i, j)} ;
11    if (i, j) is adjacent to an unmarked link (i', j') then
12      mark link (i',j');
13      add link (i',j') to the rear of LIST;
14    else
15      delete link (i,j) from LIST;
16    endif
17  endwhile
```

Figure 2-23: The modified breadth-first search algorithm

- initially, a designated link is marked;
- mark those unmarked links that are adjacent to the source link;
- a marked link is selected and mark those unmarked links adjacent to this link.

Another consideration in load partition is load balancing. Let $W_{(i,j)}$ denote the workload associated with link (i, j), that is, the number of LPV objects in link (i, j). Let $W_{A_{sub}}$ denote the workload associated with subnetwork A_{sub}, then we have $W_{A_{sub}} = \sum_{(i,j)\in A_{sub}} W_{(i,j)}$. The modified breadth-first search algorithm returns once the weight of the marked subnetwork exceeds a designated value, denoted as w.

The link-based breadth-first search algorithm is shown in Figure 2-23. In line 11 of the algorithm, when we say link (i', j') is adjacent to link (i, j), we mean that these two arcs share at least one node. Since this is a directed graph, (i', j') can be an arc connecting node i and j but in the opposite direction of link (i, j), in which case link (i', j') and (i, j) share two nodes. Readers can skip statements between line 5 to 9 at this time and they are explained later in this section.

The modified breadth-first search algorithm is called in the network partition algorithm shown in Figure 2-24. First the average workload in is calculated. Then

we perform $np-1$ modified breadth-first searches to get the subgraphs. $A[c]$ denote the set of links in subnetwork c assigned to processor c.

```
LOAD PARTITION (G(N, A), np, A[0, ··· , np − 1])
1   avg ← W_A / np ;
2   c ← 0;
3   unmark all links in A ;
4   for c ← 0, ··· , np − 1 do
5     select an unmarked link (u,v) ;
6     MODIFIED BREADTH-FIRST SEARCH((u, v), avg, A[c]) ;
7   endfor
8   select an unmarked link (u,v) ;
9   MODIFIED BREADTH-FIRST SEARCH((u, v), W_A − Σ_{0≤c≤np−2} W_{A[c]}, A[np − 1]) ;
```

Figure 2-24: The modified breadth-first search algorithm

Line 5 to 9 in algorithm Modified Breadth-First Search is important. Although the original network is assumed to have strong connectivity, the unmarked links may not be strongly connected once the set of links for the first subgraph is determined by the load partition algorithm. In order to visit all the arcs in the network, at Step 8 when $LIST$ is empty and there are unmarked arcs, we select an unmarked arc.

2.7.2 Distributed-memory Implementation

The distributed-memory implementation of the C-Load algorithm involves modifications in data structures to support the implementation of the loading algorithm.

Modifications in Data Structures

The sequential program is written in C++, which is an object-oriented programming language. There are several major classes: Class Link, Class LPV, and Class OD pair. For the parallel implementation, some of these classes need more attributes and more methods.

Links are represented in Class Link with the following attributes: link identification number, link type, tail node and head node, and pointers to Link-Path flow Variables (LPV) etc.

We supplement Class Link with the following attributes and methods to facilitate the load partition process:

- groupID: the subnetwork identification number;
- FindUpstreamGroupID(): find the subnetwork identification numbers of the upstream links;
- GetTotalHistorySize(): find the size of the *History* lists of all the LPV objects in the link. This is used to prepare the upstream to downstream communication; and
- GetNumLPV(): find the number of LPV objects in the link.

Class LPV is used to represent link-path flow variables. Each link can have more than one LPV objects. Given the network loading procedure is performed at the level of LPV objects, we supplement the following attributes to Class LPV:

- linkID: the link that the LPV object belongs to; and
- HistorySize: the size of the *History* list.

Communication

Once the network is partitioned, each processor only has full knowledge about the subnetwork stored in its own memory. However, the links on the borders need information from their upstream links when computing the values of variables: U_{ap}^{rs}, V_{ap}^{rs}, u_{ap}^{rs}, v_{ap}^{rs} in the network loading procedure.

In microscopic simulation, the vehicles are transferred from upstream links to downstream links along the boundary; while in macroscopic simulation, it is the value of the flow variables that is transferred. We illustrate this idea in Figure 2-25, which corresponds to the partition shown in Figure 2-19. Link a and link b belong to CPU 1. Link c and link d belong to CPU 2. The LPV objects in link x are numbered as LPV $x_1, x_2, \cdots, x_n$. CPU 1 has a copy of link c and link d. CPU 2 has a copy of link b. At the beginning of each Δ interval, the upstream processor CPU 1 sends the

History lists in link b to the downstream processor CPU 2. Then CPU 1 and CPU 2 perform network loading for this interval concurrently. At the end of the interval, the downstream processor CPU 2 sends the number of items to be deleted from the History lists of LPV b_1 and LPV b_2 to the upstream processor CPU 1. CPU 1 then deletes the corresponding items in LPV b_1 and LPV b_2. By now, the work for this Δ interval is finished and both processors are ready for the next Δ interval.

The method Link::FindUpstreamGroupID() returns the subnetwork identification numbers of the upstream links. Function UpstreamToDownstream() and DownstreamToUpstream() implement the communication.

After each network loading procedure, a new set of path flows is to be computed on all processes. Therefore Function SyncLinkTravelTimes() is called prior to such calculations to ensure each process has the latest link travel times.

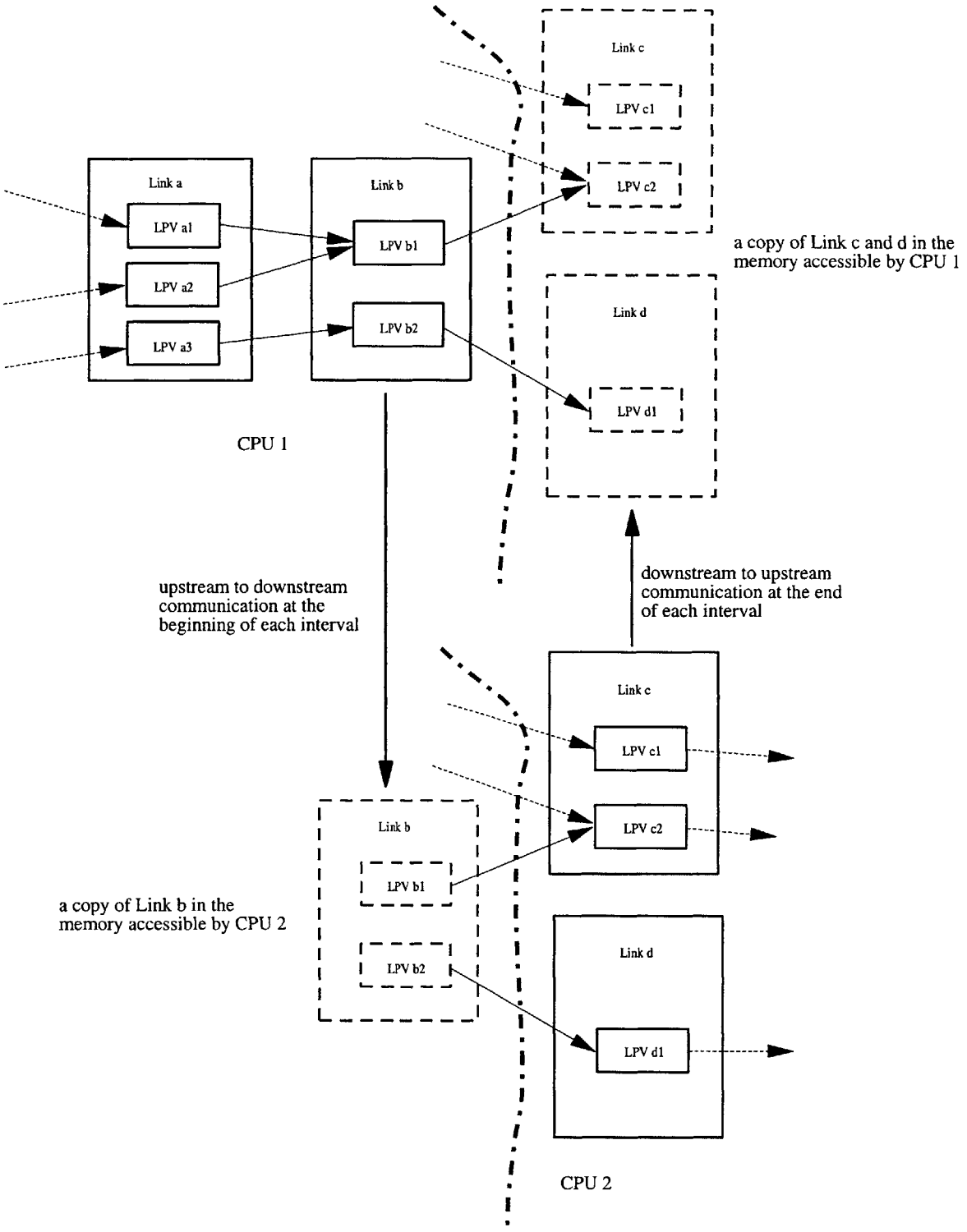

Figure 2-25: Communication in the distributed-memory implementation of C-Load algorithm by network decomposition

Process Algorithm

The work crew model is used in the distributed implementation of the chronological loading algorithm. The statements of process algorithm is in Figure 2-26

```
PROCESS ALGORITHM ()
1   rank ← GETPROCESSID ;
2   read G(N,A) and time-dependent OD demands ;
3   Δ ← min_a[τ_a(0)] ;
4   M ← number of δ intervals within a Δ interval ;
5   LOAD PARTITION(G(N, A), np, A[0, ··· , np − 1]) ;
6   Compute initial path flows ;
7   for j ← 1 to N_1 do
8     i ← 0 ;
9     while (the network is not empty) AND (i < T_dmd) do
10      UPSTREAMTODOWNSTREAM() ;
11      for a ∈ A[rank] do
12        for k ← iM to (i + 1)M − 1 do
13          Compute V_ap^rs(k) ;
14          Compute v_ap^rs(k) ;
15        endfor
16      endfor
17      DOWNSTREAMTOUPSTREAM() ;
18      for a ∈ A[rank] do
19        for k ← iM to (i + 1)M − 1 do
20          Compute u_ap^rs(k) ;
21          Compute U_ap^rs(k) ;
22          Compute X_ap^rs(k) ;
23          Compute τ_a(k) ;
24        endfor
25      endfor
26      i ← i + 1 ;
27    endwhile
28    SYNCLINKTRAVELTIMES
29  endfor
```

Figure 2-26: Process algorithm for C-Load algorithm for network decomposition strategy

2.7.3 Shared-memory Implementation

In the shared-memory implementation, we need to take measures to avoid access conflict. In Figure 2-19 LPV b_1 is upstream of LPV c_2. If CPU 1 processes LPV b_1 when CPU 2 is processing LPV c_2, there may be access conflict in the History list of LPV b_1: LPV b_1 calls AddToHead() method to add items to the head of its History list; while LPV c_2 calls Getu() method to read the History list of LPV b_1 and DeleteFromList() method to remove items from the tail of the History list of LPV b_1. In the implementation, we associate a *mutex* with each LPV object. For the three methods in Class LPV: AddToHead(), Getu(), and DeleteFromList() before accessing the History list of an LPV object, the mutex is locked; after accessing the History list, the mutex is then unlocked. By such means, the access conflict is resolved.

The master/slave paradigm is used in the shared memory implementation. The main thread (main) reads the network file and OD demands file. Then it starts NP slave threads. The slave threads work on the network loading problem and join the master thread. Finally, the master thread writes the output. The master thread algorithm is in Figure 2-27 and the slave thread algorithm is in Figure 2-28. The notation introduced in the previous section is observed.

```
MASTER THREAD ALGORITHM ()
1  read the network G(N, A) and time-dependent OD demands ;
2  CONSOLIDATE ;
3  LOAD PARTITION ALGORITHM ;
4  for n ← 1 to N_1 do
5     start NP slave threads ;
6     wait for all slave threads to join ;
7     compute g^{rs}_{mp}(k) ∀rs, ∀m, ∀p, ∀k ∈ T by ROUTE CHOICE ALGORITHM ;
8     compute new path flows f^{rs(n+1)}_{mp}(k), ∀m, ∀p, ∀k ∈ T ;
9  endfor
```

Figure 2-27: Master thread algorithm for C-Load algorithm for network decomposition strategy

```
SLAVE THREAD ALGORITHM()
1  rank ← GETTHREADID() ;
2  i ← 0 ;
3  while network is not empty do
4    for a ∈ A[rank] do
5      for k ← iM to (i + 1)M − 1 do
6        Compute V_ap^rs(k) ;
7        Compute v_ap^rs(k) ;
8      endfor
9    endfor
10   DOWNSTREAMTOUPSTREAM() ;
11   for a ∈ A[rank] do
12     for k ← iM to (i + 1)M − 1 do
13       Compute u_ap^rs(k) ;
14       Compute U_ap^rs(k) ;
15       Compute X_ap^rs(k) ;
16       Compute τ_a(k) ;
17     endfor
18   endfor
19   i ← i + 1 ;
20 endwhile
```

Figure 2-28: Slave thread algorithm for I-Load algorithm for time decomposition strategy

2.8 Experimental Setup and Numerical Results

In this section we report on the experimental setup and numerical results of the parallel algorithms developed in this chapter. Recall that our goal in this chapter is to develop faster DTA models. Hence, the first objective in this section is to compare the running times of the parallel algorithms to the corresponding serial algorithms. Besides that, we are also interested in how the running times of the parallel algorithms vary as a function of the number of processors and the effect of different analysis period on the running times of the parallel algorithms.

Besides the running times of the parallel algorithm, we also report the curves of speedup and of relative burden, as a function of the number of processors. Let $T(P)$ denote the running time obtained by using p processors. The speedup is defined as $S(p) = T(1)/T(p)$. The speedup measure does not generally allow for asymptotic performance predictions based on a small number of processors. The relative burden is described in Chabini and Gendron [40] to measure the deviation from the ideal

improvement in time from the execution of the algorithm on one processor normalized by the running time on one processor to its execution on p processors. The expression of relative burden is $B(p) = \frac{T(p)}{T(1)} - \frac{1}{p} = \frac{1}{S(p)} - \frac{1}{p}$. We have $S(p) = \frac{p}{1+B(p)p}$, and when p is large $S(p) \approx \frac{1}{B(p)}$. In the following discussion relative burden is referred as burden.

2.8.1 Test Network

The testing network is the Amsterdam A10 beltway, which is shown in Figure 2-29. It consists of two 32-km freeway loops which intersect with five major freeways and have 20 interchanges of various sizes (75 ramp intersections). The network serves local and regional traffic and acts as a hub for traffic entering and exiting north Netherland. There are 196 nodes and 310 links in the network. The number of OD pairs is about 1000. The time-dependent OD trips were originally estimated by Ashok [41] between 20 centroids based on speeds and counts measured at 65 sensor stations. Most OD pairs in the A10 beltway have two routes; therefore the total number of paths is about 1,500.

2.8.2 Test Platform

Due to the unavailability of shared-memory machines, computational tests are carried out for distributed-memory machines only. Each computational node has a dual-Intel Xeon 2.4 gigahertz-processor with 1-gigabyte memory. The computation nodes are connected through Myrinet [42]. Myrinet is a high-performance packet-communication and switching technology that is widely used to interconnect clusters of workstations, PCs, servers, or single-board computers. Characteristics that distinguish Myrinet from other networks include:

- Full-duplex 2 Gigabit/second data rate links, switch ports, and interface ports.
- Flow control, error control, and "heartbeat" continuity monitoring on every link.

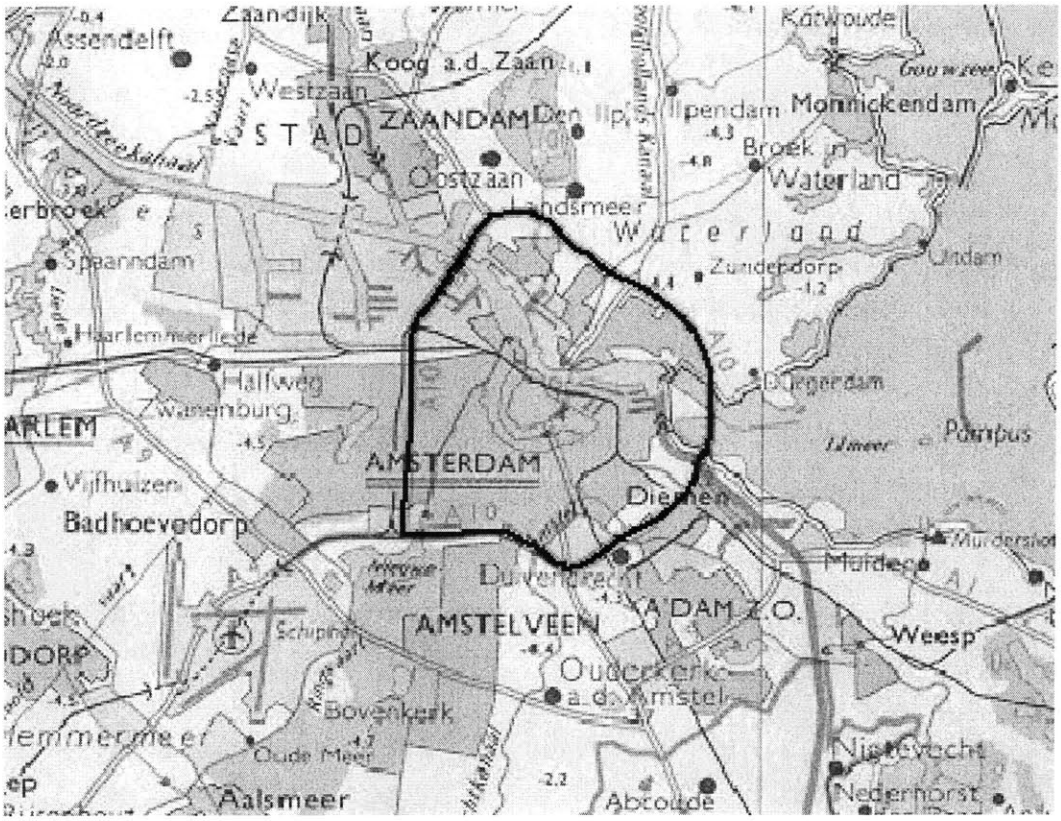

Figure 2-29: Amsterdam A10 Beltway

- Low-latency, cut-through switches, with monitoring for high-availability applications.

- Switch networks that can scale to tens of thousands of hosts, and that can also provide alternative communication paths between hosts.

Myrinet is an American National Standard – ANSI/VITA 26-1998. The link and routing specifications can be found in [43].

2.8.3 Numerical Results for the I-Load Based Parallel DTA Model

We first provides in Table 2.1 the running times of the serial algorithm for the I-Load based DTA model as a function of the duration of the maximum demand period (or analysis period) T. For a given network and a given origin-destination demand

matrix, the duration of the maximum demand period acts as a measure of the problem size, because the numbers of variables is a linear function of T.

T	300	1000	2000	3000
Run Time	235.1	774.5	1479.9	2247.8

Table 2.1: The running times of the serial algorithm for the I-Load based DTA model as a function of the duration of the maximum demand period. Running times reported are in seconds

The inputs to the parallel DTA algorithm based on I-Load are: the number of DTA loops N_1, the number of I-Load loops N_2, the number of δ intervals within each Δ interval M, and the maximum demand period T. In the testing we have $N_1 = 10, N_2 = 5, M = 5$. T varies from 300 seconds to 3000 seconds. Note that NP, the number of processors to run on, is supplied to MPI.

Network Topology Decomposition

Following are statistics of results obtained from the load partition algorithm shown in Figure 2-8. The Amsterdam network contains 1,134 OD pairs and

$$\sum_{k=1}^{1134} |OD_k^A| = 27,300.$$

When we decompose the OD pairs into 2 groups, the average number of links per group is 13,650. The partition result shows that the first group has 13,683 links,the second group has 13,617 links. When we decompose the OD pairs into 4 groups, the average number of links per group is 6,825. The partition result shows the numbers of links each group has are 6,833, 6,860, 6,840, and 6,767, respectively. When we decompose the OD pairs into 10 groups, the average number of links per group is 2,730. The partition result shows the numbers of links each group has are 2,735, 2,738, 2,730, 2,739, 2,768, 2,734, 2,765, 2,758, 2,771, and 2,562, respectively. One can note that the load partition algorithm did balance the load well. We also observe that when $NP = 10$, the last group contains 2,562 links, which deviates from the average quite a bit. This can be improved using the improvement we stated in Section 2.5.1.

Table 2.2 summarizes the running times of the parallel algorithm as a function of both the maximum demand period and the number of processors. We also report the runtime of the parallel algorithm for $NP = 1$. The running time of the parallel algorithm when $NP = 1$ reflects the extra computation effort associated with the parallel implementation.

NP	T=300	T=1000	T=2000	T=3000
1	235.5	779.7	1482.8	2240.3
2	139.0	423.4	805.7	1228.8
4	93.9	266.4	504.9	760.9
6	74.7	199.1	382.5	570.1
8	63.0	156.3	291.3	464.5
10	61.6	149.5	287.1	426.0

Table 2.2: The running times of the parallel algorithm for the I-Load based DTA model as a function of both the duration of the maximum demand period and the number of processors. The decomposition strategy applied is network-based. Running times reported are in seconds

When $NP = 1$ the parallel algorithm has similar running time to the serial code. However, the running times of the parallel algorithm is less than those of the serial code when $NP \geq 2$. We plot the speedup curves and burden curves in Figure 2-30. When $T = 300$, the speedup is not quite significant as for a small problem size. This is a common phenomenon for parallel algorithms. For curves corresponding to $T = 1000, 2000$ and 3000, one can note significant speedups. It indicates that the optimal analysis period (The optimal analysis period is the one that can lead to a maximum asymptotic speedup.) may fall into the range 1000 to 3000 seconds. Most real time DTA models work in a rolling horizon [6], this suggests that the optimal prediction period would be 1000 to 3000 seconds. For $T = 1000, 2000$ and 3000, $B(p)$ is around 0.1, therefore the asymptotic speedup is approximately 10 for those values of T.

Time-Based Decomposition

The load partition algorithm shown in Figure 2-14 partitions the demand period into well balanced sub-periods. We show the decomposition results obtained for $T = 1000$

seconds. When $NP = 2$, the two sub-periods are of length 500 seconds each. When $NP = 4$, the four sub-periods are of length 250 seconds each. When $NP = 6$, the first five sub-periods of six sub-periods are of length 167 seconds each; and the sixth sub-period are of length 165 seconds. When $NP = 10$, the 10 sub-periods are of length 100 seconds. The partition algorithm based on time decomposition provides a more balanced load compared to that based on network decomposition. Hence we expect to have a better performance in the time-based decomposition strategy.

Table 2.3 shows the running times of the parallel algorithm as a function of both the maximum demand period and the number of processors. When comparing to the results in Table 2.3, one can note that the running times for the time-based parallel algorithm are less than those for the network-based parallel algorithm for $T \leq 2000$ seconds. However, when $T = 3000$ seconds, the running times for the time-based parallel algorithm are more than those for the network-based parallel algorithm. This is caused by the larger communication need in the time-based parallel algorithm. In the network-based parallel algorithm, each process is responsible for a subset of OD pairs. It only sends the data related to those subset of OD pairs for all network intervals to other processes. However, in the time-based parallel algorithm, each process is responsible for all OD pairs. It sends the data related to all OD pairs for all network intervals to other processes. An improvement can be made to the time-based algorithm by only sending the data related to all OD pairs for a subset of network intervals to other processes.

NP	T=300	T=1000	T=2000	T=3000
1	237.0	775.0	1480.3	2235.8
2	134.5	415.2	815.7	1180.6
4	81.9	227.7	440.6	7865.9
6	66.1	165.9	319.0	5823.0
8	56.4	134.1	255.3	4396.0
10	52.9	117.5	220.4	3190.9

Table 2.3: The running times of the parallel algorithm for the I-Load based DTA model as a function of both the duration of the maximum demand period and the number of processors. The decomposition strategy applied is time-based. Running times reported are in seconds

Figure 2-31 shows the speedup curves and burden curves of the time-based decomposition strategy. The duration of analysis period varies from 300 seconds to 3000 seconds. In the burden curves plot, we did not include the case when $T = 3000$ because its burden is always greater than 1 and even reached 6 when $p = 2$. Figure 2-31 and Figure 2-30 exhibit similar trends. The best performance is reached when T is 2000 seconds. We can also observe that the time decomposition has a better performance than the network decomposition when $T = 2000$. The asymptotic speedup is 1/0.04=25. However, the time-based decomposition suffers when T is large ($T = 3000$ sec) due to a larger communication effort.

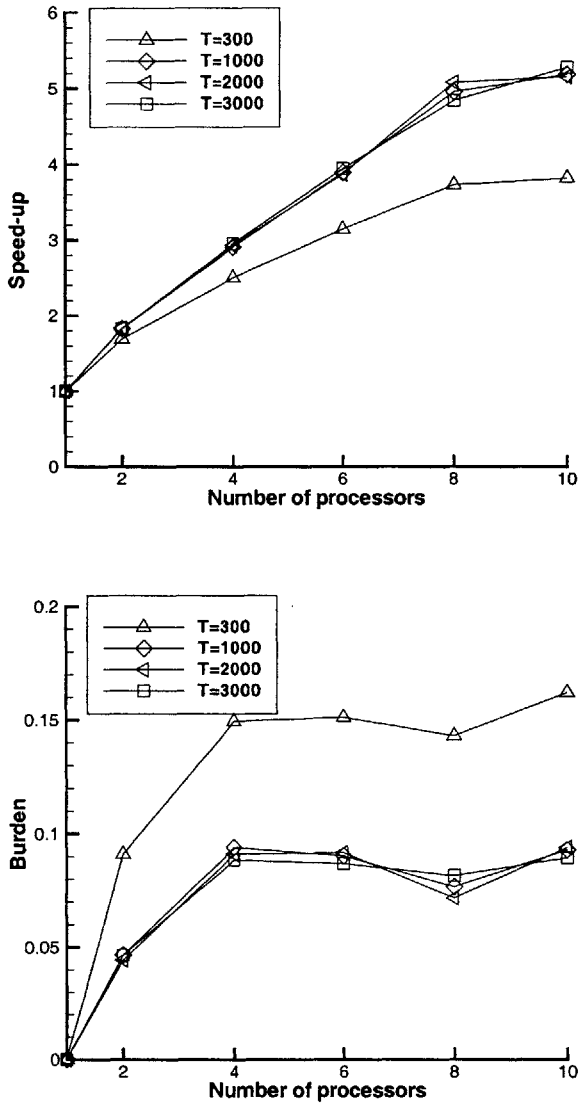

Figure 2-30: Speed-up curves and burden curves of the network decomposition MPI implementation for I-Load as a function of the number of processors. The 4 curves correspond to 4 maximum demand periods: T=300 seconds, T=1000 seconds, T=2000 seconds and T=3000 seconds.

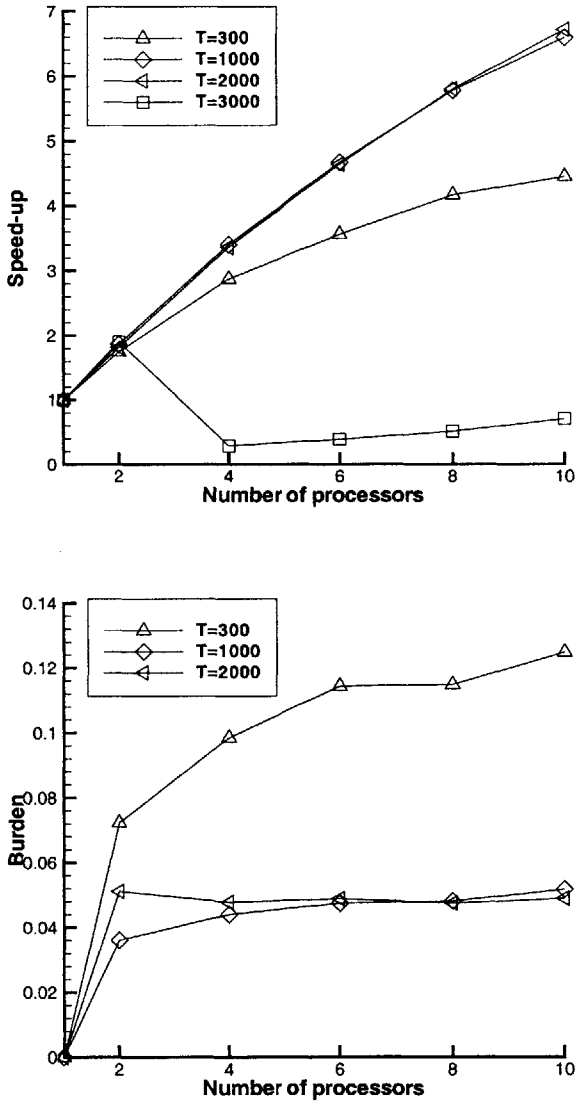

Figure 2-31: Speed-up curves and burden curves of the time decomposition MPI implementation for I-Load as a function of the number of processors. The 4 curves correspond to 4 maximum demand periods: T=300 seconds, T=1000 seconds, T=2000 seconds and T=3000 seconds. The burden curve for T=3000 is not shown because it is greater than 1

2.8.4 Numerical Results for the C-Load Based Parallel DTA Model

We first provides in Table 2.4 the running times of the serial algorithm for the C-Load based DTA model as a function of the duration of the maximum demand period (or analysis period) T. For a given network and a given origin-destination demand matrix, the duration of the maximum demand period acts as a measure of the problem size, because the numbers of variables is a linear function of T.

T	300	1000	2000	3000
Run Time	56.5	126.9	224.1	322.5

Table 2.4: The running times of the serial algorithm for the C-Load based DTA model as a function of the duration of the maximum demand period. Running times reported are in seconds

The inputs to the parallel DTA algorithm based on C-Load are: the number of DTA loops N_1, the number of δ intervals within each Δ interval M, the maximum demand period T, and the number of processors to run on p. In the testing, we fix $N_1 = 10$ and $M = 5$.

Table 2.5 summarizes the running times of the parallel algorithm as a function of both the maximum demand period and the number of processors.

NP	T =300	T =1000	T =2000	T =3000
1	62.8	133.9	238.7	341.6
2	50.6	111.0	197.8	285.6
4	45.7	101.2	182.3	265.3
6	45.7	100.1	179.2	257.5
8	49.4	108.5	198.1	293.1
10	110.8	244.5	489.3	804.1

Table 2.5: The running times of the parallel algorithm for the I-Load based DTA model as a function of both the duration of the maximum demand period and the number of processors. The decomposition strategy applied is network-based. Running times reported are in seconds

Figure 2-32 shows the speedup curves and burden curves of the network topology decomposition strategy. We note that the curves for different T have very similar

trend. The speedup increases slowly and reaches the peak when $NP = 6$. The maximum speedup obtained for the Amsterdam A10 beltway is below 1.3. The speedup decreases sharply when np goes beyond 6.

The test result does not show good performances of the distributed-memory implementation. We believe this is due to two reasons. The first one is the already very efficient implementation in the sequential program. Various measures at the algorithm and data structure level were taken to achieve computational efficiency, which are described in [5]. Due to the ring topology of the beltway, the consolidation procedure substantially reduces the number of LPV objects, therefore reduces the number of variables. The exit flows are also calculated using $V_{ap}^{rs}(k) = V_{ap}^{rs}(k-1) + \sum_{j \in \{j:(k-1)\delta < j\delta + \tau_a(j) \leq k\delta\}}$, together with the circular queue data structure, the calculation of the exit flows are fast. The second reason is that the Amsterdam A10 beltway is a relatively small network, which only contains 196 nodes and 310 links. The efficient implementation and the small network make the computation time of the sequential code fast. Let $T(1)$ denote the computation time of the sequential code. The computation time of the parallel code using p processors can be written as $T(p) = \frac{T(1)}{p} + T'$, where T' represents the communication time etc. associated with parallel processing. The speedup can then be expresses as:

$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{\frac{T(1)}{p} + T'}$$

The efficient sequential algorithm and the small test network makes $T(1) < T'$ or makes the two comparable, therefore the speedup is not sensitive to p. However, when the number of processors exceeds a certain threshold, in this case 8, T' increases due to the larger number of processors involved in the communication, so we observe a decrease in the speedup curve.

The burden curve shows that for the Amsterdam A10 beltway, the asymptotic speedup is 0. This is understandable following the above analysis. We would expect better performance in a much larger network.

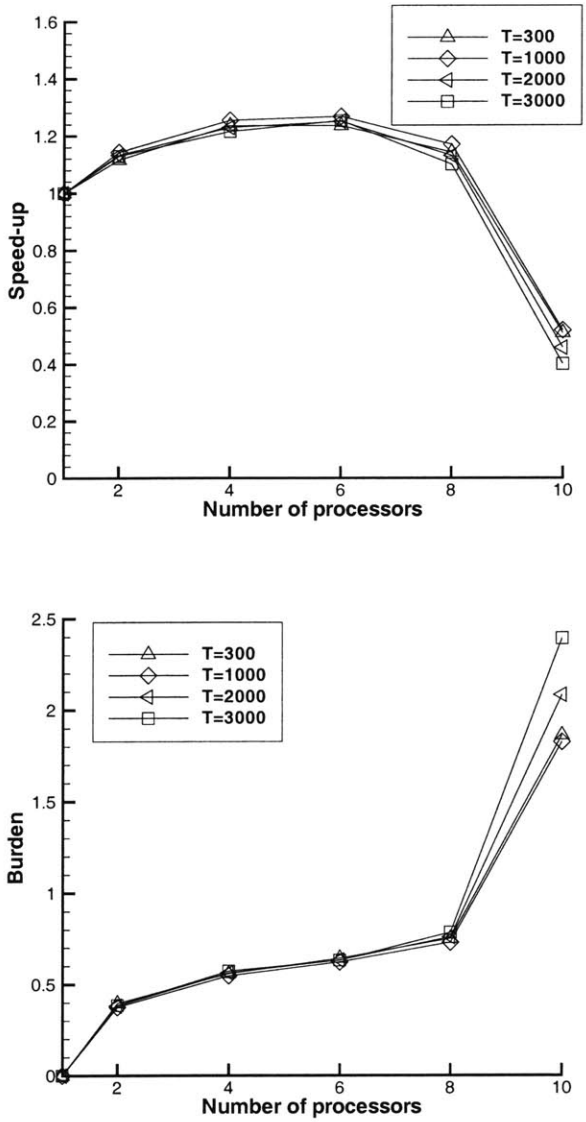

Figure 2-32: Speed-up curves and burden curves of the network decomposition MPI implementation for C-Load as a function of the number of processors. The 4 curves correspond to 4 maximum demand periods: T=300 seconds, T=1000 seconds, T=2000 seconds and T=3000 seconds.

2.8.5 Limitations in the Experimental Tests

The experimental tests is conducted on the Amsterdam A10 Beltway in the distributed-memory environment. We studied the running times of the parallel algorithms as a function of the number of processors and the duration of the analysis period. There are several limitations associated with these tests and shall be addressed in future research:

1. The Amsterdam network contains 196 nodes and 310 links, which is rather small in terms of the number of nodes and the number of arcs when compared to a network model of the great Boston area, which typically contains 7000 nodes and 25,000 links. The serial codes developed in [5] are already quite fast in such a network. Take the C-Load algorithm based DTA model, results in Table 2.4 shows the real-time run-time ratio is about 10, which is quite satisfactory. Given such a context, where the size of the problem is small and the efficiency of the serial code is satisfactory, it is desirable to perform numerical tests on a larger network to study the running times of the parallel algorithms.

2. As is shown in Figure 2-29, the Amsterdam network is a beltway, which has a ring topology. However, typical transportation networks have a grid topology. It means that the numerical results may not provide enough information to evaluate the performance of the algorithms for general transportation networks. Let n denote the number of nodes in the network and m denote the number of arcs in the network. Typical transportation networks has $m/n \approx 4$, because for each node there are four outgoing links and four incoming links. While for the Amsterdam network, $m/n = 1.58$, which means that on average there are altogether 3 links (including both outgoing links and incoming links) incident to each node. This limits the number of possible paths between any origin-destination pair. Therefore it would be of help to test the algorithms in a grid network.

3. In the numerical tests, we analyzed the running times as a function of the duration of the maximum demand period and the number of processors. It is

desirable to do some sensitivity analysis of the algorithms with respect to the number of OD pairs and the average number of enumerated paths between OD pairs.

2.9 Conclusions and Future Work

In this chapter, we did the first parallelization for macroscopic DTA models in the literature. Two loading algorithms are studied: the iterative loading algorithm and the chronological loading algorithm. Two parallelization strategies are explored where applicable: network topology based and time-based.

Tests under the distributed-memory environment are carried out. In the parallelization based on the iterative loading algorithm, it shows that both the network topology decomposition strategy and the time decomposition strategy has an optimal maximum demand period $T = 2000$.

In the parallelization based on the chronological loading algorithm, the speedup is not significant due to the efficiency sequential algorithm and the size of the network. It would be high desirable to apply the program in a larger network.

Future research in this chapter shall be directed at studying the following issues:

1. An immediate future research work is to test the parallel implementations in a shared-memory environment. By doing that we shall know the performances of the shared-memory implementations. In shared-memory machines, the communication between processors is faster and thus may lead to better performances. However, shared-memory machines are usually costly and the number of processors in one machine is limited due to technology limitations.

2. The serial DTA models used in this thesis have several limitations. First, incidents are not considered in the dynamic network loading model. Incidents are not unusual in traffic networks and can have a significant impact on traffic conditions. A proper incident model should be added to the dynamic network loading model. Second, in the DTA models used in this thesis, a volume delay function link model is used, which is a fairly simple model. It does not take into consideration the distri-

bution of traffic along links and this poses a potential threat to the accuracy of the loading results. A more realistic link performance model shall be used in future.

3. Load partition is a crucial step in parallel computing. The original problem is decomposed into sub-problems by the load partition algorithm. The more uniform the sub-problems are, the better the parallel implementation will be. Although the partition algorithms we developed shows good performance, there may exist other partition algorithms that lead to better performance.

4. In this thesis, we looked at distributed-memory implementations and shared-memory implementations. One can also consider a hybrid implementation combining the two. For example, if we have 4 SMP machines with 2 processors each. We could decompose the network into 4 subnetworks and assign each subnetwork to one SMP machine. Each of these subnetworks can then be implemented using a multi-threaded implementation within the assigned SMP machine.

Chapter 3

A Framework for Static Shortest Path Algorithms with Applications

3.1 Introduction

The computation of shortest paths is an important task in transportation applications. For instance the shortest path problem lies at the heart of the route guidance in Intelligent Transportation Systems, especially in Dynamic Traffic Assignment (DTA) problems. In such applications, there is usually a need to solve a large number of shortest path problems in dynamic networks. To meet the real-time operational requirement of ITS applications, efficient solution algorithms are desired.

Consider a network model of the Greater Boston area, which typically contains 7,000 nodes, 25,000 arcs, 1,000 origin nodes and 1,000 destination nodes. Suppose that one wants to provide route guidance to drivers in the 2-hour morning peak period. If we discretize time at a 15-second interval, we have 480 departure times. One current serial computer with a 2 GHz processor solves a single one-to-all shortest path problem in such a network in the order of 10^{-3} seconds. The time to compute shortest paths in one DTA iteration, which amounts to calculating 1,000 trees for 480 possible departure time intervals, then requires $10^{-3} \times 1000 \times 480 = 480$ seconds. In order to obtain a satisfactory approximation of DTA solution, it typically requires at least 10 DTA iterations. Therefore the total time that would be needed to compute

shortest paths would be $480 \times 10 = 4800$ seconds $= 1.3$ hours. This is a close figure to the length of the analysis period, which prevents using this model in real-time applications.

There are several variants of the shortest path problems that arise in common transportation applications. First, depending on the number of origins and destinations, there are 1-to-1, 1-to-all, all-to-1, and many-to-all etc. shortest path problems. Second, depending on whether link travel times are static or time-dependent, there are static shortest path problems and dynamic shortest path problems. Third, depending on whether the First-In-First-Out (FIFO) property is satisfied, there are FIFO shortest path problems and non-FIFO shortest path problems. Fourth, depending on whether link travel times are stochastic or not, there are deterministic shortest path problems and stochastic shortest path problems.

In this chapter we first focus on the problem of computing 1-to-all shortest paths in dynamic FIFO networks for all departure times $1, 2, \cdots, T_0$. Transportation networks usually satisfy the First-In-First-Out property. Consider dynamic networks, which are defined as networks in which link travel times are time-dependent, the FIFO property for an arc holds if and only if an individual leaving the source node cannot arrive at the end node earlier by departing later. If all arcs satisfy FIFO property for all departure times, the network is a FIFO network. Furthermore if an individual leaving the source node later can only arrive at the end node later, we say the strict FIFO property for that arc holds. If all arcs satisfy strict FIFO property for all departure times, the network is a strict FIFO network.

The 1-to-all shortest path problem in dynamic FIFO and strict FIFO networks can be decomposed into a series of static shortest path problems [36]. When typical classical static shortest path algorithms are applied to solve dynamic shortest path problems, they solves the T_0 1-to-all shortest path problems independently of each other. It is then interesting to develop static shortest path algorithms that can benefit from results from other departure times

We present a new framework which exploits information from the results of previous 1-to-all problems to solve the current 1-to-all problem. The idea of the framework

is to prioritize (label set) nodes with optimal distance labels (earliest arrival times) in the scan eligible list of label-correcting algorithms. To achieve this, a key question to answer is how to determine the optimality of node labels. In dynamic strict FIFO networks, we exploit the fact that $a_i(t-1)$, the minimum arrival time for departure time $t-1$ at the origin, is a lower bound on $e_i(t)$, the possible arrival time for departure time t at the origin. If $e_i(t) = a_i(t-1)+1$, we label node i as optimal and prioritize it. This idea extends to FIFO networks as well, that is, node i is optimal if $e_i(t) = a_i(t-1)$.

This framework is also applicable to static shortest path problems. In the 1-to-all static shortest path problem, through introducing the *findmin* operation, we construct an optimality condition to determine whether a node is optimal. Such technique also makes label-correcting algorithms more efficient to solve 1-to-1 and 1-to-many shortest path problems. In the many-to-all shortest path problem, classical shortest path algorithms simply solve a series of independent 1-to-all shortest path problems for all origins. The potential to enhance the computation time for this problem has been observed in [44], where an efficient method is proposed for recalculating the shortest path tree when the origin node is changed. If one replaces the arc costs with their reduced costs and applies the algorithm we proposed to solve for origin s_1 and then for s_2, it automatically explores the shortest path tree it has already computed for s_1. When it label-sets a node, it consecutively label-sets all the nodes in the subtree rooted at that node in the shortest path tree found for s_1. If $s_2 = s_1$, it will compute the shortest path tree starting at s_2 in linear time, while for classical shortest path algorithms, the computation effort for s_2 will be the same as that for s_1.

The rest of this chapter is organized as follows. In Section 3.2 we present a new framework for static shortest path algorithms. We then provide applications of this framework in Section 3.3 through 3.5. Finally, in Section 3.6, we provide computer implementations and numerical results for the application of the algorithm in dynamic networks.

3.2 The New Framework

Let $G = (N, A)$ be a graph, where N is the set of nodes and A is the set of arcs. Let the number of nodes in G be $n = |N|$, and the number of arcs be $m = |A|$. We associate with each arc (i, j) a non-negative travel time d_{ij}. The minimum travel time from the origin node s to a node i is denoted by d_i (d_i is also referred as distance label of node i in the following text). For a given node i, let $A(i) = \{j : (i, j) \in A\}$ and $B(i) = \{j : (j, i) \in A\}$. We also refer to $A(i)$ as the forward star of node i and $B(i)$ as the backward star of node i. s is the origin node. The length of a path $(i_1, i_2, \cdots, i_k)$ is the sum of the length of its arcs. A shortest path between node i_1 and i_k is one that has the minimum length among all paths linking i_1 and i_k.

A generic static shortest path algorithm maintains a scan eligible list SE and does the following:

algorithm: generic Static Shortest Path (SSP)

```
1   d_s ← 0; d_j ← ∞ for each j ∈ N \ {s} ;
2   SE ← {s}; ;
3   while SE ≠ ∅ do
4      remove a node i from SE ;
5      for allj ∈ A(i) do
6         if d_j > d_i + d_ij then
7            d_j ← d_i + d_ij ;
8            SE ← SE ∪ {j} ;
9         endif
10     endfor
11  endwhile
```

Different algorithms are distinguished by the method of selecting the node to exit the scan eligible list SE at each iteration. In one major class, the label-setting algorithms (for example: Dijkstra's algorithm [45, 46]), the node exiting SE is a node whose label is the minimum among all the nodes in SE. Algorithms that do not follow this node selection criterion are called label-correcting algorithms. In label-correcting algorithms, the selection of the node to be removed from SE is faster than that in label-setting methods, at the expense of potential multiple entrances of nodes in SE. An example of label-correcting algorithm is the Bellman-Ford algorithm [47, 48] that uses a queue to maintain the scan eligible list SE. For the ease of discussion, in the

following text, we refer to the Bellman-Ford algorithm with a queue implementation of the scan eligible list as the Bellman-Ford algorithm.

In Dijkstra's algorithm, each node selected from SE is optimal because we assume d_ij is non-negative; while in label-correcting algorithms, this is usually not true. We are interested in finding ways to determine nodes that have optimal distance labels at each iteration. In addition to the scan eligible list SE, we create a Priority Scan Eligible list PSE. Nodes in SE that have optimal distance labels are moved into PSE. PSE enjoys preemptive priority queue over SE, which means SE is not processed until PSE is empty and the processing of SE is interrupted in the event that the PSE becomes non-empty.

The framework with priority enabled for static shortest path algorithms is described in Figure 3-1.

algorithm: SSP algorithm with priority enabled

```
1   d_s ← 0; d_j ← ∞ for each j ∈ N \ {s} ;
2   SE ← {s}; PSE ← ∅ ;
3   while SE ∪ PSE ≠ ∅ do
4     if PSE ≠ ∅ then
5       remove a node i from PSE ;
6     else
7       remove a node i from SE ;
8     endif
9     for j ∈ A(i) do
10      if d_j > d_i + d_ij then
11        d_j ← d_i + d_ij ;
12        SE ← SE ∪ {j} ;
13        for all k ∈ SE do
14          if k is determined to be optimal then
15            PSE ← PSE ∪ {k} ;
16            SE ← SE \ {k} ;
17          endif
18        endfor
19      endif
20    endfor
21  endwhile
```

Figure 3-1: A framework for static shortest path algorithms with priority enabled

The differences between this new framework and that of a generic shortest path

algorithm lie in two aspects. First, in the beginning of the main loop, appropriate modifications are made to implement the preemptive priority queue PSE. Second, in the end of the main loop, some mechanisms should be applied to replenish the priority scan eligible list PSE.

Algorithms developed under this framework should be distinguished from the threshold algorithm [49]. Both algorithms maintain two scan eligible list; however, there is one fundamental difference: in threshold algorithm, nodes are replenished to the two lists by the threshold value and the optimality of their distance labels is unknown. In the framework above, all nodes replenished to PSE are guaranteed to have optimal distance labels.

The label-setting algorithm and label-correcting algorithms can be well interpreted under this general framework. When a node is selected from SE and its forward star is explored, we term it as one *iteration* in the algorithms. If exactly one node that has the minimum distance label in SE is added to PSE at each iteration, then we obtain Dijkstra's algorithm. If SE is implemented as a binary heap, the effort to replenish PSE is $O(log(n))$ in each iteration. Each node is visited exactly once and the algorithm stops after n iterations. If no nodes are ever added to PSE, we obtain label-correcting algorithms. In this case, the effort to replenish PSE is zero with a potential drawback that a node may be visited more than once [50].

We observe that there is a trade-off to be done between the effort to replenish PSE and the number of iterations for the algorithm to stop. We pose the question of whether there are any better ways to replenish PSE so as to balance the effort to replenish PSE and the number of iterations the algorithm requires to stop? We associate a threshold value μ_j for node j such that when $d_j \leq \mu_j$, we can claim that d_j is optimal. Note that both μ_j and d_j may change during the iterations, with d_j non-increasing and μ_j non-decreasing. In the following discussion we refer to this condition as the *optimality condition.*

The key question then is how to calculate μ_j. We propose different efficient ways to determine μ_j in different networks.

3.3 Application in Dynamic Strict FIFO Networks

Let $d_{ij}(t)$ denote the travel time along arc (i,j) when we enter arc (i,j) at time t. The minimum travel time from the source node s to a node i departing the source at time t is denoted by $d_i(t)$, and the minimum arrival time at node i when departing the source at time t by $a_i(t)$. Let $e_i(t)$ be a feasible arrival time to node i, which means $e_i(t)$ is an upper bound on the minimum arrival time to node i when departing the origin node at time t. Note that $a_i(t) = t + d_i(t)$.

In discrete time networks the strict FIFO property of an arc (i,j) holds true if and only if the inequality $d_{ij}(t+1) \geq d_{ij}(t)$ holds true for all values of t. If all arcs satisfy the strict FIFO property for all departure times, the network is a strict FIFO network. In discrete time strict FIFO networks, $a_i(t)$ is an increasing function of t for each i, that is, $a_i(t_1) < a_i(t_2)$, if $t_1 < t_2$.

Proposition 1 *In dynamic strict FIFO networks, if* $e_i(t+1) = a_i(t)+1$, *then* $e_i(t+1)$ *is optimal, that is,* $a_i(t+1) = e_i(t+1)$.

The results of Proposition 1 follows directly from the definition of strict FIFO networks. A usefulness of this result is that we can define the threshold for node j for departure time t as $\mu_j(t) = a_j(t-1) + 1$. To make the expression $a_i(t-1)$ valid for $t = 1$ we assume that $a_i(0)$ is 0 for all i. The pseudocode of the adaption of the framework described in the previous section to determine shortest path tree for departure time t assuming that an optimal solution for departure time $t-1$ is known is given in Figure 3-2.

3.4 Application in Dynamic FIFO Networks

In discrete time networks the FIFO property of an arc (i,j) holds true if and only if the inequality $d_{ij}(t+1) \geq d_{ij}(t) - 1$ holds true for all values of t. If all arcs satisfy the FIFO property for all departure times, the network is a FIFO network. In discrete time FIFO networks, $a_i(t)$ is a non-decreasing function of t for each i, that is, $a_i(t_1) \leq a_i(t_2)$, if $t_1 < t_2$.

algorithm: label-correcting with priority enabled in strict FIFO networks

```
1   a_i(0) ← 0, ∀i ∈ N ;
2   for t ← 1 to T_0 do
3     a_s(t) ← t ;
4     a_i(t) ← ∞, ∀i ∈ N \ {s} ;
5     SE ← {s} ; PSE ← ∅ ;
6     while SE ∪ PSE ≠ ∅ do
7       if PSE ≠ ∅ then
8         remove a node i from PSE ;
9       else
10        remove a node i from SE ;
11      endif
12      for j ∈ A(i) do
13        if a_j(t) > a_i(t) + d_ij(a_i(t)) then
14          a_j(t) ← a_i(t) + d_ij(a_i(t)) ;
15          if a_j(t) = a_j(t − 1) + 1 then
16            PSE ← PSE ∪ {j} ;
17            if j ∈ SE then
18              SE ← SE \ {j} ;
19            endif
20          else
21            if j ∉ SE then
22              SE ← SE ∪ {j} ;
23            endif
24          endif
25        endif
26      endfor
27    endwhile
28  endfor
```

Figure 3-2: Label-correcting with priority enabled in strict FIFO networks

Proposition 2 *In dynamic FIFO networks, if* $e_i(t+1) = a_i(t)$, *then* $e_i(t+1)$ *is optimal, that is,* $a_i(t+1) = e_i(t+1)$.

This comes directly from the definition of FIFO networks. Based on Proposition 2, we can define the threshold for node j for departure time t as $\mu_j(t) = a_j(t-1)$.

In FIFO networks we could change the condition "if $a_j(t) = a_j(t-1) + 1$" to "if $a_j(t) = a_j(t-1)$" in Line 15 in Figure 3-2 to make it suitable for FIFO networks. While this will lead to more efficient algorithms compared to a repetitive application of a static shortest path algorithm, a more efficient adaptation can be developed.

Denote the shortest path tree computed for time t as T_t.

Proposition 3 *Suppose that we compute $T_1, T_2, \cdots, T_{T_0}$ in increasing order of the departure times. For $i \in N$, if $e_i(t+1) = a_i(t)$, there exists a shortest path tree T_{t+1} such that the subtree rooted at node i in T_{t+1} is the same as that in T_t.*

To prove this is equivalent to prove that for any node j in the subtree rooted at node i in T_t, $a_j(t+1) = a_j(t)$. Given that $e_i(t+1) = a_i(t)$, if we follow the path from i to j in T_t, we have $e_j(t+1) = a_j(t)$. Thus $a_j(t+1) \leq e_j(t+1) = a_j(t)$. On the other hand, from the FIFO property we have $a_j(t+1) \geq a_j(t)$. Therefore $a_j(t+1) = a_j(t)$.

Proposition 3 implies that if $e_i(t+1) = a_i(t)$, we need not explore the forward star of node i. This is quite powerful in reducing the number of arcs scanned and we could design more efficient algorithms for FIFO networks than for strict FIFO networks.

There are various ways to implement the idea of not exploring the forward star of node i. One way is to compute shortest path trees in decreasing order of time. We initialize $a_i(t-1)$ to the value of $a_i(t)$ when computing the 1-to-all shortest path problem for departure time $t-1$. The advantages of working in decreasing order of time and initializing $a_i(t-1)$ to $a_i(t)$ are twofold: 1) $a_i(t)$ does provide an upper bound on $a_i(t-1)$, which is required by the algorithm. The algorithm improves the upper bounds until the upper bounds equal the optimal values; 2) after the initialization of $a_i(t-1)$ to $a_i(t)$, if $a_i(t-1)$ never decreases in computing T_{t-1}, it will never be added to SE and its forward star will never be explored. The non-exploration of its forward star does not jeopardize the optimality of the minimum arrival times for nodes in the subtree routed at i in T_{t-1}, because the minimum arrival times of those nodes for departure time $t-1$ are the same as the minimum arrival times of those nodes for departure time t, which are the initial values of the minimum arrival times of those nodes for departure time $t-1$.

The overall algorithm starts by computing a shortest path tree for $t = T_0$ using any shortest path algorithms and obtain $a_i(T_0), \forall i \in N$. The pseudocode of this algorithm is in Figure 3-3.

algorithm: label-correcting with priority enabled in FIFO networks

```
1   compute a_i(T_0), for ∀i ∈ N ;
2   for t ← T_0 − 1 to 1 do
3     a_s(t) ← t; a_i(t) ← a_i(t + 1), ∀i ∈ N \ {s} ;
4     SE ← {s} ;
5     while SE ≠ ∅ do
6       remove a node i from SE ;
7       for j ∈ A(i) do
8         if a_j(t) > a_i(t) + d_ij(a_i(t)) then
9           a_j(t) ← a_i(t) + d_ij(a_i(t)) ;
10          SE ← SE ∪ {j} ;
11        endif
12      endfor
13    endwhile
14  endfor
```

Figure 3-3: Label-correcting with priority enabled in FIFO networks

We did not use PSE explicitly in the above algorithm as we need not explore the forward star of those nodes. One should also note that the arrival times of those nodes are no longer guaranteed to be optimal.

The reason why we need to work in decreasing order of time is that it is impossible to avoid exploring the forward star of node i if $a_i(t+1) = a_i(t)$ and if we work in increasing order of time. We cannot initialize $a_i(t+1)$ to $a_i(t)$ when computing T_{t+1}, because the initial value of $a_i(t+1)$ should always be an upper bound on the optimal value of $a_i(t+1)$. Suppose that we initialize $a_i(t+1)$ to $f_i(t+1)$, where $f_i(t+1)$ is an upper bound on the optimal value of $a_i(t+1)$ and $f_i(t+1) > a_i(t)$. In such a context, if we detect that at some time $a_i(t+1) = a_i(t)$ for some node i, we still need to explore the forward star of node i; otherwise, the optimality of $a_j(t)$, for all $j \in A(i)$, are not guaranteed.

Proposition 4 *The algorithm in Figure 3-3 solves the one-to-all shortest path problems for all departure times in $O(mT_0)$.*

Proof: It is shown in [51] that a dynamic network can be viewed as a static network by using the time-expanded network representation. We provide a brief description as

follows. The static network is formed by expanding the original dynamic network in the time dimension, and making a separate copy of all nodes for every integer value of time $t \in \{1, 2, \cdots, T_0\}$. Every node in the time-expanded network represents a time-node pair consisting of a time $t \in \{1, 2, \cdots, T_0\}$ and a node $i \in N$, where the nodes at the highest level of time are taken to represent not only time interval T_0, but all times greater than or equal to T_0. Every link in a time-expanded network is a directed link from a node-time pair (i, t) to another node-time pair $(j, min\{T_0, t + d_{ij}(t)\})$.

Time-expanded networks have the following properties, which are identified in [51].

1. Along the time dimension, they are acyclic if arc travel times are positive.

2. Every path on the original dynamic network corresponds to a path on the time-expanded network with the same travel time. Visiting a node i in the original dynamic network at time t corresponds to visiting node-time pair (i, t) in the corresponding time-expanded network.

3. A shortest path problem in a dynamic network can be solved by applying a static shortest path algorithm to its equivalent representation as time-expanded network.

A consequence of properties 2 and 3 above is that dynamic shortest path problems can be solved by applying static shortest path algorithms to the time-expanded representation of a dynamic network. In the time-expanded network, there are nT_0 nodes and mT_0 links, a direct repetitive application of label-correcting algorithm with a queue implementation of the scan eligible list leads to a total runtime complexity of $O(T_0 \times mT_0 \times nT_0) = O(mnT_0^3)$.

However, for the algorithm shown in Figure 3-3, the forward star of each node-time pair in the time-expanded network is visited at most once, that is, the links in the time-expanded network are visited at most once. Therefore the runtime complexity of the algorithm is $O(mT_0)$. ■

Curious readers may ask why this idea of not exploring the forward star of node i if $a_j(t+1) = a_j(t) + 1$ is not applicable in strict FIFO networks. This is because for a

node $i \in N$, even if $a_i(t+1) = a_i(t)+1$, this would not provide any information about T_{t+1}. We only explored the subtree rooted at node i for departure time $a_i(t)$; and we have not yet explored the subtree routed at node i for departure time $a_i(t) + 1$.

3.5 Application in Static Shortest Path Problems

As is stated before, the many-to-all shortest path problem can be decomposed into 1-to-all shortest path problems. Our goal is to find the linkage between these 1-to-all shortest path problems, therefore it is instructive to first focus on how this framework will fit in the 1-to-all problem. The application in many-to-all is presented subsequently.

3.5.1 1-to-all Shortest Path Problems

We define the caliber of node i is defined as $C(i) = min_{j \in B(i)} d_{ij}$. We also introduce a *findmin* operation over SE. Denote the set of nodes in the scan eligible list at iteration k as SE^k, then the *findmin* operation returns the minimum distance label SE^k_{min} of the nodes in SE^k.

Proposition 5 *SE^k_{min} is a non-decreasing function of k, that is, $SE^k_{min} \leq SE^l_{min}$, if $k \leq l$.*

Because the arc travel times are non-negative, the distance label of any node j added to SE after iteration k must satisfy the inequality $d_j \geq SE^k_{min}$. Therefore the minimum in SE will never decrease.

Proposition 6 *d_j is optimal is optimal if $d_j \leq SE^l_{min} + C(j)$, $j \in SE^l$.*

We only need to prove that once node j satisfies the above condition, it will never be added to SE, which means that its distance label d_j will remain unchanged afterwards. Assume at iteration $m(m > l)$ node j is added to SE, which means $\exists k \in N$ such that $d_k + d_{kj} < d_j$. Then we have $d_j > d_k + d_{kj} \geq SE^m_{min} + C(j) \geq SE^l_{min} + C(j)$. This contradicts the assumption that $d_j \leq SE^l_{min} + C(j)$.

Proposition 7 *d_j is optimal if $d_j \leq SE^k_{min} + C(j)$, $j \in SE^l$, $k \leq l$.*

From Proposition 5, we can derive that $d_j \leq SE^k_{min} + C(j) \leq SE^l_{min} + C(j)$. From Proposition 6, we know node j is optimal.

Given Proposition 7, we need not find the minimum distance label at each iteration. Because the *findmin* operation is expensive, the minimum distance label found in previous iterations can be used as a lower bound to the current minimum distance label in the scan eligible list.

Define μ^k_j as the threshold for node j at iteration k: $\mu^k_j = SE^l_{min} + C(j)$, l is the latest iteration in which a *findmin* operation was carried out. At iteration k any node j in SE that satisfies $d_j \leq \mu^k_j$ is optimal. The PSE list is replenished by the following mechanism: 1) if node j is already in SE, remove it from SE and add it to PSE; 2) if that node is not in SE, add it to PSE directly.

We note that if the *findmin* operation is carried out at each iteration, we are doing something similar to Dijkstra's algorithm, particularly Dial's implementation of Dijkstra's algorithm. We may pay too much to accomplish the *findmin* operation in reducing the number of iterations needed by the algorithm. Instead, we adopt the following strategy which is shown in Figure 3-4. SE and PSE are implemented as a queue. Suppose at iteration k, we did a *findmin* operation, which means we scan SE and find the minimum distance label. We use two pointers P_1(black) and P_2(gray) to indicate the front and the end of SE. Then we scan SE for a second time, remove those nodes that satisfy the optimality condition ($d_j \leq \mu^k_j$) in SE and add them to PSE. If the node to be removed is the front of SE, we update P_1 such that it will not point to removed nodes in SE. After that we process nodes in PSE and SE according to the rule we presented in the general framework. When processing a node, say i, we look at its forward star. If its child j is to be updated, which means $d_i + C(j) \leq d_j$, we add node j to PSE or SE depending on whether it satisfies the optimality condition. This process continues and at a certain iteration, we will encounter the situation when $PSE = \emptyset$ and P_1 passes P_2. Now, we will perform a second *findmin* operation and make P_2 point to the end of $SE(P_2 \leftarrow rear[SE])$.

The pseudocode for the algorithm is shown in Figure 3-5.

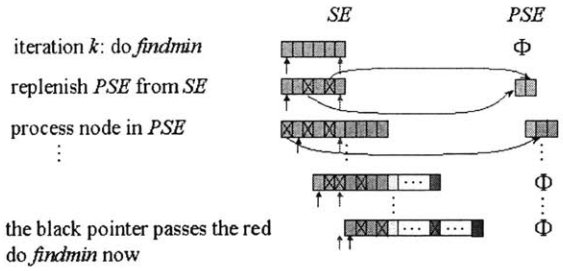

Figure 3-4: How the *findmin* operation is carried out in 1-to-all SSP problems.

algorithm SSP with findmin operation

```
1   d_s ← 0; d_j ← ∞ for each j ∈ N \ {s} ;
2   SE ← {s}; PSE ← ∅ ;
3   CurMin ← 0 ;
4   while SE ∪ PSE ≠ ∅ do
5     if PSE ≠ ∅ then
6       remove the front node i of PSE ;
7     else
8       remove the front node i of SE and update P_1 ;
9     endif
10    for j ∈ A(i) do
11      if d_j > d_i + d_ij then
12        d_j ← d_i + d_ij ;
13        if d_j ≤ CurMin + C(j) then
14          PSE ← PSE ∪ {j} ;
15          if j ∈ SE then
16            SE ← SE \ {j} ;
17          endif
18        else
19          if j ∉ SE then
20            SE ← SE ∪ {j} ;
21          endif
22        endif
23      endif
24    endfor
25    if (P_1 passes P_2) AND (PSE = ∅) then
26      P_2 ← rear[SE] ;
27      CurMin ← FINDMIN(SE) ;
28      for all j ∈ SE do
29        if d_j ≤ CurMin + C(j) then
30          SE ← SE \ {j} ;
31          PSE ← PSE ∪ {j} ;
32        endif
33      endfor
34    endif
35  endwhile
```


Figure 3-5: A static shortest path algorithm with findmin operation

The above algorithm also makes label-correcting algorithms more efficient in solving 1-to-1 and 1-to-many shortest path problems. Label-correcting algorithms solve a 1-to-all problem in order to solve a 1-to-1/1-to-many problem. By introducing the *findmin* operation and using the property of caliber, we construct an optimality condition $d_j \leq \mu_j$. Label-correcting algorithms with the *findmin* operation can stop once the destination node(s) verify(ies) this optimality condition.

This algorithm should be distinguished from the threshold algorithm [52]. They share the attribute that they both maintain two lists for the candidate nodes. However, there is a fundamental difference. In the algorithm we present nodes in PSE have optimal distance labels; while in threshold algorithm, neither list contains exclusively nodes with optimal distance labels.

3.5.2 An Example

To assist the reader in better understanding the algorithm presented in this chapter, we provide a small example. The example network is shown in Figure 3-6 accompanied by Table 3.1 which summarizes the solution steps. In the following paragraph we will walk through the steps the algorithm would take in solving the shortest paths from source node 0.

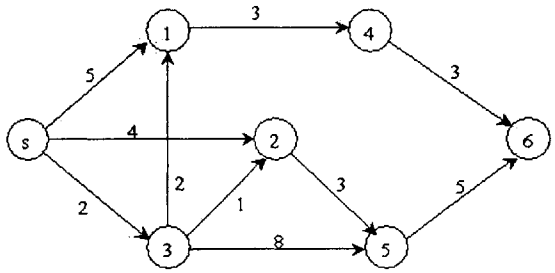

Figure 3-6: A sample network used to demonstrate the fundamentals of the algorithm.

The first column shows the iteration number. The second column shows the current SE_{min}. The third and fourth column show the content of SE and PSE respectively. The last 6 columns show the changes to the distance labels of the 6 nodes in the network. The number in parenthesis is the caliber of each node. For

iteration	SE_{min}	SE						PSE	1(2)	2(1)	3(2)	4(3)	5(3)	6(3)
1	0	s						Φ	-	-	-	-	-	-
2	0		1	2				3	5	4	2	-	-	-
findmin	0		1	2	5			Φ	4	3	2	-	10	-
3	3		1^+	2^+	5			1, 2	4	3	2	7	10	-
4	3		1^+	2^+	5	4		2	4	3	2	7	10	-
5	3		1^+	2^+	5^+	4		5	4	3	2	7	6	-
	3		1^+	2^+	5^+	4	6	Φ	4	3	2	7	6	11
findmin	3		1^+	2^+	5^+	4	6	Φ	4	3	2	7	6	11
6	7		1^+	2^+	5^+	4^+	6	4	4	3	2	7	6	11
7	7		1^+	2^+	5^+	4^+	6^+	6	4	3	2	7	6	11

Table 3.1: Steps to solve the shortest path problem for the network in Figure 3-6 with source node 0.

SE, two pointers P_1 and P_2 are maintained as described in the previous paragraphs. The black arrow indicates the location of P_1 and the gray arrow indicates that of P_2. The circles and ellipses show the operation carried out at each iteration. The algorithm visits the nodes in the sequence: s, 3, 1, 2, 5, 4, 6. However, if we apply Bellman-Ford algorithm to this example, we will visit the nodes as following: s, 1, 2, 3, 4, 5, 1, 2, 6, 4, 5, 6.

The *findmin* operation is also applicable to FIFO networks. Let $SE^k(t)$ denote the set of nodes in the scan eligible list at iteration k when computing T_t, then the *findmin* operation returns the minimum value in $SE^k(t)$, which is denoted as $SE^k_{min}(t)$. We can define a static caliber of node i as $C(i) = min_{j \in B(i), 1 \le t \le T_0} d_{ji}(t)$. We can say that $a_j(t)$ is optimal if $a_j(t) \le SE^k_{min}(t) + C(j)$. Alternatively, we can also define a dynamic caliber of node i as $C(i,t) = min_{j \in B(i)} d_{ji}(t)$, then $a_j(t)$ is optimal if $a_j(t) \le SE^k_{min}(t) + C(j, SE^k_{min}(t))$.

3.5.3 Many-to-all Shortest Path Problems

In a static network G_1, the many-to-all shortest path problem is defined as finding the shortest paths from origin nodes $s_1, s_2, \cdots, s_p$ to all other nodes in the network.

Suppose we have computed the shortest path tree T_{s_1} for origin s_1.Define the reduced cost of arc (i,j) with respect to origin s as $\overline{d}_{ij}^{\,s} = d_{ij} - (d^s_j - d^s_i)$. We replace d_{ij} with its reduced cost $\overline{d}_{ij}^{\,s_1}$ in the network and obtain G_2. A shortest path solution in G_1 is a shortest path solution in G_2 and vice versa [53].

Now we want to compute a shortest path tree T_{s_2}, which originates from s_2. The running times of certain classical shortest path algorithms are invariant to the change of cost, that is, if we apply classical shortest path algorithms, the run time would be in the same order of that for s_1 no matter we use G_1 or G_2. However, if we prioritize nodes with optimal distance labels in G_2, we could substantially outperform classical algorithms.

Proposition 8 *If node $i \in PSE$ is processed, the nodes in the subtree routed at i in T_{s_1} will be consecutively label-set and added to PSE.*

Proof: if $(i,j) \in T_{s_1}$, $\overline{d}_{ij}^{\,s_1} = 0$. Therefore $C(i) = 0$ for all $i \in G_2 \setminus \{s_1\}$. Suppose i is j's predecessor in T_{s_1}, $d_j^{s_2} = d_i^{s_2} \leq CurMin + 0$. Node j satisfies the condition to enter PSE. Therefore node j, the descendent of i in T_{s_1}, will be added to PSE. This process is applicable for all nodes in the subtree rooted at i. ■

This algorithm is more efficient in solving many-to-all shortest path problems than a repetitive application of a shortest path algorithm. As a special case, if $s_1 = s_2$, the algorithm solves the shortest path tree rooted at s_2 in linear time, which is $\theta(n+m)$. While for certain classical shortest paths algorithms, the computation effort to solve the shortest path tree rooted at s_2 is in the same order as that to solve the shortest path tree rooted at s_1. For example, for a binary-heap implementation of Dijkstra's algorithm, it is $O((m+n)log(n))$. For a queue-implementation of label-correcting algorithm, it is $O(mn)$.

3.6 Computer Implementations and Numerical Results

The algorithms for dynamic shortest path algorithms presented in this chapter have been implemented and tested. The computer implementation were written in C/C++ [54, 55]. The tests were performed on a Pentium III 733 megahertz computer with 320 megabytes of RAM. In this section, we describe the objective of the tests, the test networks used, and the computational results obtained.

3.6.1 Dynamic FIFO/strict FIFO Networks

The objective of the experimental study is to analyze the running times of Algorithm LCP for dynamic shortest path problems in FIFO/strict FIFO networks, as a function of the size of the network, the number of nodes, the number of arcs, and the percentage of dynamic arcs.

Two types of random networks are used in the tests: networks with a random topology (Type I) and grid networks (Type II). Type I random networks are generated using a pseudo random network generator. The input to the network generator are: the number of nodes n , the number of arcs m, the number of time intervals T, the range of link travel times $1, \cdots, C$, the percentage of dynamic arcs α, and a parameter estimating the level of link dynamics β. The topology of the network is generated in two stages. First a cycle containing all nodes is generated to ensure strong connectivity. Then the remaining links are added randomly. Type II random networks are generated using a grid network generator. The input to the generator are: the number of columns and the number of rows instead of the number of nodes and arcs. Each node will have a directed link to its adjacent nodes. We set T sufficiently large, such that all arcs are dynamic for departure time $1, 2, \cdots, 100$. α indicates the percentage of arcs that are dynamic in the network. β describes the probability that a particular dynamic arc changes its travel time in the next time interval. We assume that β is the same for all links. In order to avoid the situation that link travel times go to either a very large integer or zero, we assume that the probability that travel time increases and decreases is the same. Therefore $0 \leq \beta \leq 0.5$. $d_{ij}(t)$ is generated as follows:

$$d_{ij}(t+1) = \begin{cases} d_{ij}(t)+1 & \text{w.p. } \beta, \\ d_{ij}(t) & \text{w.p. } 1-2\beta, \\ max\{d_{ij}(t)-1, 0\} & \text{w.p. } \beta. \end{cases} \tag{3.1}$$

We also assume that the changes of the travel time of a link is either +1 or -1 in

FIFO networks. In discrete time dynamic FIFO networks, the the travel time of a link can decrease at most 1 from one interval to the other; however, it can increase more than +1. The assumption is used to make sure that the travel times of arcs will not increase to infinity (or a very large integer). The $d_{ij}(t)$'s generated satisfy the FIFO property because for each link (i,j), we have $d_{ij}(t) \geq d_{ij}(t) - 1$ for all t.

In strict FIFO networks ($0 \leq \beta \leq 1$) $d_{ij}(t)$'s are generated as follows:

$$d_{ij}(t+1) = \begin{cases} d_{ij}(t) + 1 & \text{w.p. } \beta, \\ d_{ij}(t) & \text{w.p. } 1 - \beta. \end{cases} \tag{3.2}$$

The $d_{ij}(t)$'s generated satisfy the strict FIFO property because for each link (i,j), we have $d_{ij}(t) \geq d_{ij}(t)$ for all t.

In the experiment tests, we fix $\beta = 0.1$. All running times are reported in seconds and are averaged over 5 trials of each algorithm.

The running time reported are the total running time for solving the one-to-all dynamic shortest path problem for departure time 0 through 100. For ease of discussion, we shall refer to the class of algorithms developed in this chapter as Algorithm LCP (Label-Correcting with Priority). The running time of LCP is compared to the successive application of a queue implementation of Bellman-Ford algorithm, which is the same as using a queue to implement the scan eligible list in a label-correcting algorithm (we refer this algorithm to Algorithm BF). We also report the total number of iterations for both algorithms in the parentheses.

Tables 3.2 - 3.4 show the running times of LCP and BF in FIFO random networks (Type I), as a function of the size of the network, the number of nodes, the number of arcs, and the percentage of dynamic arcs. Algorithm LCP runs about 2 to 4 times faster than Algorithm BF, and the ratio of the running time of Algorithm BF to that of Algorithm LCP (this is also referred to as *speedup* in later discussion) is an increasing function of α. This is because the larger α is, the more links are dynamic, which means their travel time can decrease. This leads to the result that more nodes

satisfy the optimality condition.

Table 3.5 shows the running times of LCP and BF in FIFO grid networks, as a function of the size of the network and the percentage of dynamic arcs. Algorithm LCP runs more than 7 times faster than Algorithm BF when $\alpha = 100\%$ in 60×60 grid networks. This can be attributed to two sources: 1) Algorithm BF performs worse in grid networks than in random networks (Type I); 2) Algorithm LCP is insensitive to the network topology.

Tables 3.6 - 3.8 show the running times of LCP and BF in strict FIFO random networks (Type I), as a function of the size of the network, the number of nodes, the number of arcs, and the percentage of dynamic arcs. Table 3.6 shows the effect of α on the run time of LCP and BF. It is observed that Algorithm LCP runs approximately 2 times faster than Algorithm BF when $\alpha < 50\%$, and the speedup is a decreasing function of α. The best case of Algorithm LCP is when $\alpha = 0\%$. In such a situation, all nodes satisfy the optimality condition. Table 3.7 and Table 3.8 shows the effect of network density and of α on the run time of LCP and BF. One can note that the denser the network, the better the speedup. This is because in dense networks, there are more paths from the origin to the destinations. Therefore there is a larger change for the destinations to verify the optimality condition.

Table 3.9 shows the running times of LCP and BF in strict FIFO grid networks, as a function of the size of the network and the percentage of dynamic arcs. The maximum speed up is achieved when $\alpha = 0\%$. The speedup is a decreasing function of α and the speedup is not significant. This can be explained as follows. In grid networks, each node has at most four arcs connected to the four adjacent nodes (the nodes on its top, bottom, left and right). A direct consequence is that in grid networks, the shortest paths contain more links than in Type I random networks. The more links on the shortest path, the higher the probability that the travel time along the path will increase in the next interval. Therefore the probability that the destination node satisfies the optimality condition decreases.

3.6.2 Many-to-all Static Shortest Path Problems

The objective of the experiment study is to analyze the performance of Algorithm LCP for static shortest path problems in random networks (Type I) and grid networks, as a function of the size of the network, the number of nodes, and the number of arcs.

Two types of random networks are tested and the network topology is generated in the same way as stated in the previous section. The link costs are integers between 1 and 100.

First Algorithm BF is applied to the network G_1 to solve a 1-to-all static shortest path from node 0. Then the link reduced costs network G_2 is calculated respect to node 0, that is, $\overline{d}_{ij}^0 = d_{ij} + d_i^0 - d_j^0$. Algorithm LCP is applied to solve the 1-to-all static shortest path problem from node 0 in G_2. Clearly, each node will be visited exactly once in this scenario, and this is the best case of Algorithm LCP. In the tests, a source node is randomly picked to serve as the origin node in a new instance of a one-to-all shortest path problem. Algorithm BF and Algorithm LCP are applied to solve this problem in G_1 and G_2 respectively. The running time may depend on the origin. We then select five source nodes randomly and report the average running time and average number of iterations for both algorithms.

Table 3.10 shows the results when we vary the size of the network (Type I random network). One can observe that the performance of Algorithm LCP highly depends on the number of children in the subtree routed at the source new node in T_0. If the number of descendants of the new source in T_0 is large, the savings obtained from Algorithm LCP will be greater. When node 0 is again selected as the source node, we observe that in the best case, a speedup of 1.2 is achieved. One should note that for Type 1 random network, each node is visited about 1.5 to 2 times on average; therefore a speedup of 1.2 is satisfactory. While there exist cases in which Algorithm LCP runs slower than Algorithm BF; however, on average, Algorithm LCP requires less iterations and less time.

Table 3.11 and Table 3.12 investigates the sensitivity of the two algorithms regarding to network density in Type 1 random network. In Table 3.11 we fix $m = 10,000$

and vary n. In Table 3.12 we fix $n = 100$ and vary m. The ratio of the average number of iterations of Algorithm BF to Algorithm LCP remains almost unchanged and fluctuates between 1 and 1.3. Therefore Algorithm LCP is not sensitive to network density.

Table 3.13 summaries the results in random grid networks. Similar trends are observed as compared to random networks. The average performance of Algorithm LCP is better than Algorithm BF.

3.7 Conclusions and Future Work

In this chapter we proposed a new framework for static shortest path algorithms. This framework prioritizes nodes with optimal distance labels. Existing algorithms are well interpreted under this framework. The algorithms developed under this framework are hybrid ones between label-setting algorithms and label-correcting algorithms. We applied this framework in three situations arose in dynamic strict FIFO networks, dynamic FIFO networks, and static networks. Different mechanisms to replenish the priority scan eligible list PSE are studied, which resulted into new and efficient specialized algorithms for various instances of the shortest path problem.

Interesting future research questions in this area include:

1. Throughout the implementations of the algorithms in this chapter, the scan eligible list (SE) and the priority scan eligible list (PSE) are implemented as a queue. However, SE and PSE can be implemented using other data structures, for example, a dequeue.

2. In Section 3.5 we designed a method to determine the frequency of the $findmin$ operation. However, numerical tests did not show significant savings achieved using LCP. We found that PSE is empty most of the time between two consecutive $findmin$ operations, which means that it is hard to replenish PSE between two consecutive $findmin$ operations. The reason is explained in the following text.

Suppose at iteration k we performed a $findmin$ operation. We re-scan SE^k to move those nodes that satisfy the condition $d_j \leq SE^k_{min} + C(j)$ from SE^k to PSE. For the remaining nodes in SE^k, we have $d_j > SE^k_{min} + C(j)$. If j is added to PSE in later iterations, it means that there must exist a node $i \in B(j)$ that decreases d_j to d'_j, that is,

$$d'_j = d_i + d_{ij} \leq SE^k_{min} + C(j) \tag{3.3}$$

However, on the other hand, we have $d_i \geq SE^k_{min}$ and $d_{ij} \geq C(j)$, therefore

$$d_i + d_{ij} \geq SE^k_{min} + C(j) \tag{3.4}$$

From Inequality 3.3 and 3.4, we know that if node j is replenished to PSE between two consecutive $findmin$ operations, we require that $d(i) = SE^k_{min}$ and $d_{ij} = C(j)$. This condition is not easy to satisfy, therefore we seldom observe nodes being replenished to PSE between two consecutive $findmin$ operations.

We may increase the frequency of the $findmin$ operation to increase the number of nodes being added to PSE; however, we also have to pay the penalty associated with more frequent $findmin$ operations.

Another way to increase the number of nodes being added to PSE between to consecutive $findmin$ operations is to increase the value of the right hand side of Inequality 3.3. We can re-define $C(i) = min_{j \in B(i), d(j) \text{ is not optimal}} d_{ij}$. Then the value of $C(i)$ is a non-decreasing function of the iteration counter k.

3. We proposed a way to generate the time-dependent link travel times in FIFO and strict FIFO networks. However, there can be other ways to generate the time-dependent link travel times. We shall focus on the generation of discrete time dynamic FIFO networks in our discussion.

   The solid line in Figure 3-7 illustrates the profile of the generated link travel times for link (i, j) following Equation 3.1. The values of $d_{ij}(t)$ oscillate around the initial value of $d_{ij}(t)$, which is $d_{ij}(0) = d_0$.

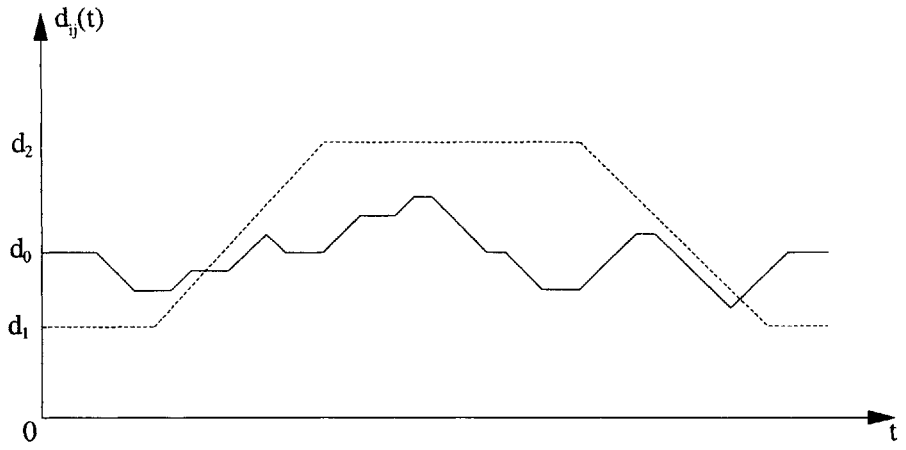

Figure 3-7: Two possible profiles for link travel times in FIFO networks

We could have generated the link travel times that look like the dashed line in Figure 3-7. The dotted curve mimic a peak in the travel times, which is similar to that in traffic networks in midday. In the morning, the traffic is light and the travel times are small, then the travel times start to increase to d_2 in midday, finally the travel times decrease to d_1. There can be more than one peaks to model the morning peaks and afternoon peaks.

4. We also want to point out that for the nodes in the priority scan eligible list, we can apply parallel computing technology to scan their forward stars in parallel.

FIFO dynamic networks (Type I) with m=9n						
α	n=	500	1000	2000	3000	4000
5%	LCP	0.0475	0.1053	0.2370	0.3586	0.5587
		(49789)	(99752)	(202001)	(287613)	(386237)
	BF	0.1087	0.2076	0.5375	0.7462	1.2634
		(115613)	(201614)	(495223)	(634839)	(952129)
25%	LCP	0.0472	0.0980	0.2135	0.3562	0.5273
		(44155)	(90868)	(178797)	(273238)	(368823)
	BF	0.1063	0.2093	0.4520	0.8157	1.5507
		(109288)	(208229)	(411841)	(690247)	(961197)
50%	LCP	0.0441	0.0817	0.1896	0.3017	0.4290
		(41984)	(79584)	(152093)	(239633)	(326746)
	BF	0.0836	0.1845	0.4999	0.7961	1.4499
		(92317)	(185740)	(454324)	(679292)	(920280)
75%	LCP	0.0352	0.0788	0.1594	0.2807	0.3387
		(37439)	(77446)	(134751)	(225583)	(263472)
	BF	0.0962	0.2106	0.5511	0.7513	1.6596
		(108855)	(209538)	(511783)	(640718)	(1058385)
100%	LCP	0.0326	0.0702	0.1599	0.2469	0.3850
		(34136)	(67921)	(127525)	(195453)	(268427)
	BF	0.1021	0.2123	0.4615	0.7808	1.4388
		(110121)	(213736)	(421321)	(669080)	(903882)

Table 3.2: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in FIFO random networks (Type I) as a function of network size. The numbers in parentheses are the total number of iterations

FIFO dynamic networks (Type I) with m=10000					
	n=	1000	2000	3000	4000
5%	LCP	0.1531	0.1771	0.2138	0.2411
		(100227)	(199793)	(297536)	(392942)
	BF	0.2698	0.3197	0.4193	0.4140
		(220422)	(358398)	(533926)	(674387)
25%	LCP	0.1186	0.1650	0.2134	0.2046
		(90761)	(173736)	(263203)	(314540)
	BF	0.3394	0.3310	0.3960	0.4037
		(254252)	(351318)	(519597)	(688542)
50%	LCP	0.1253	0.1381	0.1604	0.1663
		(82692)	(149400)	(221347)	(259910)
	BF	0.3221	0.3237	0.3627	0.4299
		(247662)	(393559)	(553715)	(708865)
50%	LCP	0.1018	0.1310	0.1178	0.1502
		(75527)	(143020)	(175840)	(224038)
	BF	0.2579	0.3057	0.3822	0.4059
		(218597)	(353768)	(597485)	(692671)
50%	LCP	0.0935	0.1310	0.1233	0.1471
		(70761)	(123981)	(170128)	(211778)
	BF	0.3068	0.3362	0.3239	0.3942
		(237788)	(377146)	(510355)	(664127)

Table 3.3: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in FIFO random networks (Type I) as a function of number of nodes. The numbers in parentheses are the total number of iterations

FIFO dynamic networks (Type I) with n=100						
	m=	500	1000	2000	3000	4000
5%	LCP	0.0062	0.0092	0.0159	0.0242	0.0301
		(10114)	(9883)	(10065)	(10099)	(10152)
	BF	0.0106	0.0172	0.0324	0.0475	0.0573
		(17170)	(18837)	(21009)	(17017)	(18024)
25%	LCP	0.0062	0.0129	0.0158	0.0230	0.0307
		(9789)	(9054)	(9765)	(9768)	(9782)
	BF	0.0087	0.0165	0.0298	0.0443	0.0589
		(14908)	(17514)	(18652)	(19715)	(20282)
50%	LCP	0.0052	0.0084	0.0140	0.0206	0.0291
		(8233)	(8936)	(8769)	(9031)	(9530)
	BF	0.0087	0.0147	0.0265	0.0352	0.0707
		(15685)	(16285)	(16878)	(15459)	(22988)
75%	LCP	0.0095	0.0075	0.0137	0.0194	0.0264
		(8516)	(8017)	(8373)	(8253)	(8630)
	BF	0.0076	0.0127	0.0268	0.0361	0.0471
		(13919)	(14609)	(17578)	(16281)	(16701)
100%	LCP	0.0043	0.0075	0.0135	0.0194	0.0249
		(6956)	(7719)	(7957)	(8192)	(8144)
	BF	0.0091	0.0139	0.0303	0.0384	0.0697
		(17414)	(16264)	(16892)	(17760)	(22572)

Table 3.4: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in FIFO random networks (Type I) as a function of number of arcs. The numbers in parentheses are the total number of iterations

FIFO grid networks				
	$n \times n$	40×40	50×50	60×60
5%	LCP	0.0685 (139257)	0.1437 (212233)	0.2146 (338247)
	BF	0.1402 (373962)	0.2561 (559007)	0.5100 (1152387)
25%	LCP	0.0480 (88697)	0.1318 (180839)	0.1571 (239548)
	BF	0.1504 (416899)	0.3100 (722010)	0.4003 (1038187)
50%	LCP	0.0375 (73799)	0.0693 (112259)	0.1029 (162266)
	BF	0.1601 (443951)	0.3947 (944321)	0.4903 (1095775)
75%	LCP	0.0273 (54077)	0.0594 (105416)	0.0922 (134885)
	BF	0.1675 (480410)	0.3955 (931281)	0.5301 (1152428)
75%	LCP	0.0337 (56298)	0.0396 (71736)	0.0773 (120189)
	BF	0.1767 (370084)	0.3303 (743653)	0.5460 (1221745)

Table 3.5: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in FIFO grid networks as a function of network size. The numbers in parentheses are the total number of iterations

strict FIFO dynamic networks (Type I) with m=9n						
	n=	500	1000	2000	3000	4000
0%	LCP	0.0537	0.1131	0.2424	0.4004	0.5698
		(50811)	(102180)	(204156)	(306480)	(408264)
	BF	0.0789	0.2215	0.4458	0.7721	1.0219
		(81911)	(220180)	(419756)	(697910)	(834664)
5%	LCP	0.0540	0.1159	0.2442	0.4036	0.5842
		(50902)	(106136)	(204625)	(309884)	(417456)
	BF	0.0887	0.1973	0.4489	0.7834	0.9805
		(93348)	(198297)	(419084)	(695080)	(803669)
25%	LCP	0.0544	0.1164	0.2505	0.4395	0.6640
		(51531)	(108458)	(207548)	(331908)	(474193)
	BF	0.0776	0.1995	0.4488	0.7731	1.1409
		(79735)	(198214)	(416897)	(689550)	(869485)
50%	LCP	0.0556	0.1197	0.2596	0.4519	0.6366
		(54204)	(110987)	(220389)	(356447)	(462320)
	BF	0.0817	0.1958	0.4494	0.6476	1.0833
		(84867)	(193706)	(397749)	(567242)	(863905)
50%	LCP	0.0584	0.1232	0.2823	0.4796	0.6856
		(56599)	(115432)	(243662)	(383614)	(490462)
	BF	0.0634	0.1384	0.3066	0.5327	0.7497
		(64589)	(134507)	(284519)	(460774)	(584976)

Table 3.6: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in strict FIFO networks (Type I) as a function of network size. The numbers in parentheses are the total number of iterations

strict FIFO dynamic networks (Type I) with m=10000					
	n=	1000	2000	3000	4000
0%	LCP	0.1210	0.1693	0.2245	0.2980
		(102017)	(203745)	(304992)	(405978)
	BF	0.2203	0.2779	0.3156	0.3507
		(203717)	(378245)	(504192)	(603778)
5%	LCP	0.1227	0.1761	0.2399	0.3020
		(104838)	(208759)	(310021)	(412725)
	BF	0.2362	0.2617	0.4155	0.4004
		(215502)	(354085)	(604464)	(657641)
25%	LCP	0.1270	0.1796	0.2553	0.3197
		(108133)	(214710)	(343990)	(445848)
	BF	0.2230	0.2552	0.3123	0.4091
		(206548)	(345684)	(468855)	(693154)
50%	LCP	0.1248	0.1908	0.2897	0.3842
		(107341)	(234547)	(409372)	(519576)
	BF	0.2022	0.2264	0.3090	0.3463
		(186446)	(304845)	(487260)	(580808)
100%	LCP	0.1369	0.2097	0.3199	0.4009
		(117569)	(262841)	(424170)	(619572)
	BF	0.1664	0.1881	0.2700	0.2407
		(150297)	(247762)	(412812)	(466600)

Table 3.7: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in strict FIFO networks (Type I) as a function of number of nodes. The numbers in parentheses are the total number of iterations

strict FIFO networks (Type I) with n=100						
	m=	500	1000	2000	3000	4000
0%	LCP	0.0067	0.0103	0.0180	0.0240	0.0309
		(10151)	(10163)	(10185)	(10152)	(10169)
	BF	0.0089	0.0154	0.0306	0.0349	0.0519
		(15251)	(16463)	(19392)	(15352)	(17473)
5%	LCP	0.0066	0.0105	0.0173	0.0250	0.0311
		(10140)	(10196)	(10174)	(10176)	(10157)
	BF	0.0077	0.0182	0.0275	0.0395	0.0516
		(12954)	(19607)	(17424)	(17323)	(17484)
25%	LCP	0.0070	0.0104	0.0184	0.0245	0.0333
		(10765)	(10262)	(10563)	(10200)	(10677)
	BF	0.0085	0.0185	0.0291	0.0368	0.0544
		(14231)	(19268)	(16657)	(15843)	(18602)
50%	LCP	0.0070	0.0105	0.0177	0.0249	0.0316
		(10345)	(10243)	(10402)	(10384)	(10182)
	BF	0.0098	0.0148	0.0262	0.0384	0.0466
		(16989)	(15955)	(16216)	(16716)	(15510)
50%	LCP	0.0072	0.0108	0.0191	0.0252	0.0335
		(11115)	(10727)	(11294)	(10460)	(10458)
	BF	0.0078	0.0131	0.0266	0.0321	0.0395
		(13151)	(13431)	(16116)	(13418)	(12862)

Table 3.8: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in strict FIFO networks (Type I) as a function of number of arcs. The numbers in parentheses are the total number of iterations

strict FIFO grid networks				
	$n \times n$	40×40	50×50	60×60
0%	LCP	0.0878	0.1441	0.2209
		(165883)	(257561)	(376000)
	BF	0.2090	0.2712	0.5581
		(594183)	(763661)	(1616000)
5%	LCP	0.0983	0.1466	0.2214
		(196258)	(258823)	(409436)
	BF	0.1782	0.3074	0.3737
		(470395)	(866312)	(1077044)
25%	LCP	0.0818	0.1615	0.3694
		(188121)	(334628)	(986061)
	BF	0.1570	0.2562	0.4551
		(474805)	(750309)	(1381843)
50%	LCP	0.1065	0.2784	0.4594
		(291311)	(857060)	(1354481)
	BF	0.1147	0.2528	0.4024
		(342883)	(798691)	(1217600)
100%	LCP	0.0963	0.1984	0.3304
		(279282)	(585266)	(884954)
	BF	0.0619	0.1019	0.1648
		(174149)	(264495)	(379088)

Table 3.9: Total running times (in sec) and total numbers of iterations required by LCP and BF to solve the one-to-all shortest path problems for departure times $1, 2, \cdots, 100$ in strict FIFO grid networks as a function of network size. The numbers in parentheses are the total number of iterations

source node		0	33	38	123	153	272	average
n=500	BF	0.000207	0.000235	0.000223	0.000188	0.000229	0.000217	0.0002184
		(773)	(812)	(847)	(686)	(867)	(814)	(805.2)
	LCP	0.000176	0.000196	0.000247	0.00017	0.000217	0.000192	0.0002044
		(500)	(541)	(866)	(500)	(780)	(699)	(677.2)
source node		0	130	397	725	798	955	average
n=1000	BF	0.000416	0.000526	0.000525	0.000456	0.000469	0.000467	0.0004886
		(1534)	(1951)	(1907)	(1547)	(1584)	(1614)	(1720.6)
	LCP	0.000364	0.000456	0.000407	0.000408	0.000491	0.000441	0.0004406
		(1000)	(1633)	(1373)	(1429)	(1740)	(1560)	(1547)
source node		0	182	549	1609	1884	1904	average
n=2000	BF	0.001061	0.001042	0.000909	0.000971	0.001045	0.000941	0.0009816
		(3299)	(3163)	(2944)	(3179)	(3289)	(2976)	(3110.2)
	LCP	0.000755	0.00086	0.000854	0.000946	0.000959	0.000925	0.0009088
		(2000)	(2859)	(2954)	(3272)	(3012)	(3152)	(3049.8)
source node		0	671	1430	2218	2332	2683	average
n=3000	BF	0.002034	0.001845	0.001659	0.001609	0.001926	0.001715	0.0017508
		(6588)	(5835)	(4968)	(4808)	(5495)	(5090)	(5239.2)
	LCP	0.001596	0.002029	0.001483	0.001511	0.001812	0.001627	0.0016924
		(3000)	(5757)	(4910)	(4695)	(5340)	(4570)	(5054.4)
source node		0	144	840	1297	1641	2179	average
n=4000	BF	0.002664	0.002554	0.002799	0.002804	0.00253	0.002858	0.002709
		(6916)	(6363)	(7249)	(7099)	(6474)	(7558)	(6948.6)
	LCP	0.002616	0.003247	0.003583	0.002827	0.002834	0.002859	0.00307
		(4000)	(6525)	(7201)	(6804)	(6355)	(5747)	(6526.4)
source node		0	924	1224	2947	3426	3646	average
n=5000	BF	0.003931	0.003801	0.004091	0.004229	0.004162	0.004774	0.0042114
		(8419)	(8855)	(8692)	(8483)	(8552)	(9791)	(8874.6)
	LCP	0.005004	0.00433	0.004824	0.005498	0.004758	0.005815	0.005045
		(5000)	(5000)	(8644)	(8271)	(8332)	(9831)	(8015.6)

Table 3.10: Running times (in sec) and number of iterations of Algorithm BF and LCP for 1-to-all static shortest path problems in random networks (Type I) as a function of network size (m=3n). The numbers in parentheses are the number of iterations

source node		0	108	303	319	331	339	average
n=500	BF	0.001267	0.001356	0.001218	0.001252	0.001253	0.001342	0.0012842
		(1086)	(1196)	(1054)	(1077)	(1085)	(1136)	(1109.6)
	LCP	0.00056	0.001072	0.00093	0.001053	0.000832	0.001074	0.0009922
		(500)	(1035)	(904)	(1015)	(804)	(1050)	(961.6)
source node		0	263	350	426	834	923	average
n=500	BF	0.001482	0.001312	0.001506	0.001378	0.001401	0.001453	0.00141
		(2169)	(1971)	(2322)	(2025)	(2019)	(2107)	(2088.8)
	LCP	0.000734	0.00075	0.001294	0.001216	0.00094	0.001153	0.0010706
		(1000)	(1026)	(2086)	(1955)	(1537)	(1913)	(1703.4)
source node		0	209	420	1278	1494	1885	average
n=1000	BF	0.001915	0.001844	0.00158	0.001723	0.001894	0.001651	0.0017384
		(4050)	(4237)	(3446)	(3934)	(4372)	(3759)	(3949.6)
	LCP	0.001042	0.00133	0.001299	0.001687	0.001699	0.001687	0.0015404
		(2000)	(3331)	(3288)	(3782)	(4288)	(3498)	(3637.4)
source node		0	234	687	1298	2130	2437	average
n=2000	BF	0.00172	0.001902	0.001725	0.001961	0.002034	0.001759	0.0018762
		(4579)	(4599)	(4694)	(5507)	(5931)	(4621)	(5070.4)
	LCP	0.001847	0.001364	0.001303	0.001804	0.001767	0.001621	0.0015718
		(3000)	(3000)	(3000)	(5482)	(4790)	(4750)	(4204.4)
source node		0	1218	1629	1798	2468	3146	average
n=3000	BF	0.001812	0.001895	0.002215	0.002344	0.002103	0.001943	0.0021
		(5610)	(5709)	(6375)	(6219)	(6687)	(6225)	(6243)
	LCP	0.002593	0.002042	0.002125	0.003212	0.003185	0.002862	0.0026852
		(4000)	(5524)	(6092)	(6602)	(6513)	(6222)	(6190.6)
source node		0	538	1133	2398	3158	4601	average
n=4000	BF	0.002452	0.002437	0.002098	0.003114	0.002885	0.002264	0.0025596
		(7444)	(8104)	(7171)	(7228)	(8076)	(7313)	(7578.4)
	LCP	0.004861	0.003842	0.003247	0.004141	0.0036	0.004176	0.0038012
		(5000)	(7819)	(6836)	(8801)	(8605)	(5000)	(7412.2)

Table 3.11: Running times (in sec) and number of iterations of Algorithm BF and LCP for 1-to-all static shortest path problems in random networks (Type I) as a function of number of nodes (m=10,000). The numbers in parentheses are the number of iterations

source node		0	7	16	74	89	95	average
m=500	BF	0.058	0.055	0.056	0.063	0.059	0.054	0.057
		(153)	(163)	(153)	(180)	(160)	(141)	(159.4)
	LCP	0.046	0.051	0.049	0.061	0.064	0.052	0.055
		(100)	(129)	(125)	(152)	(164)	(127)	(139.4)
source node		0	9	42	54	62	97	average
m=1000	BF	0.100	0.100	0.116	0.086	0.091	0.108	0.100
		(169)	(169)	(199)	(147)	(155)	(179)	(169.8)
	LCP	0.070	0.077	0.107	0.071	0.074	0.088	0.083
		(100)	(128)	(177)	(111)	(111)	(143)	(134.0)
source node		0	28	53	71	85	88	average
m=2000	BF	0.206	0.181	0.179	0.207	0.192	0.211	0.194
		(204)	(180)	(174)	(200)	(189)	(210)	(190.6)
	LCP	0.112	0.182	0.155	0.173	0.184	0.206	0.180
		(100)	(176)	(152)	(168)	(175)	(200)	(174.2)
source node		0	3	4	47	72	95	average
m=3000	BF	0.263	0.287	0.253	0.279	0.297	0.273	0.278
		(171)	(182)	(164)	(183)	(201)	(181)	(182.2)
	LCP	0.154	0.197	0.221	0.245	0.228	0.174	0.213
		(100)	(131)	(149)	(166)	(156)	(114)	(143.2)
source node		0	45	49	53	53	91	average
m=4000	BF	0.357	0.367	0.359	0.341	0.335	0.348	0.350
		(189)	(186)	(186)	(170)	(170)	(172)	(176.8)
	LCP	0.195	0.306	0.197	0.308	0.304	0.293	0.282
		(100)	(163)	(102)	(164)	(164)	(153)	(149.2)
source node		0	11	52	64	69	92	average
m=5000	BF	0.481	0.482	0.569	0.437	0.438	0.418	0.469
		(185)	(197)	(191)	(180)	(176)	(171)	(183.0)
	LCP	0.247	0.263	0.387	0.358	0.319	0.367	0.339
		(100)	(110)	(167)	(156)	(137)	(157)	(145.4)

Table 3.12: Running times (in 10^{-3} sec) and number of iterations of Algorithm BF and LCP for 1-to-all static shortest path problems in random networks (Type I) as a function of number of arcs (n=100). The numbers in parentheses are the number of iterations

source node		0	292	590	596	777	1366	average
40x40	BF	1.358	1.121	1.395	1.313	1.061	1.141	1.206
		(4413)	(3412)	(4507)	(4161)	(3133)	(3596)	(3761.8)
	LCP	0.593	1.012	1.217	1.198	0.927	1.137	1.098
		(1600)	(3054)	(3816)	(3933)	(2922)	(3662)	(3477.4)
source node		0	167	185	1230	1637	2478	average
50x50	BF	2.675	2.423	2.367	2.127	2.063	2.156	2.227
		(8626)	(7765)	(7673)	(6390)	(5997)	(6396)	(6844.2)
	LCP	0.993	1.675	1.832	1.797	2.012	2.063	1.876
		(2500)	(4847)	(5631)	(5469)	(5811)	(6322)	(5616.0)
source node		0	251	283	662	1743	3546	average
60x60	BF	3.269	3.926	3.947	4.488	3.344	5.683	4.278
		(9425)	(12030)	(12521)	(14328)	(9831)	(19079)	(13557.8)
	LCP	2.031	2.642	3.775	2.347	2.771	5.733	3.454
		(3600)	(6237)	(11105)	(5406)	(7173)	(18041)	(9592.4)
source node		0	1519	2875	3197	4143	4446	average
70x70	BF	8.696	5.594	6.076	5.138	5.634	5.807	5.650
		(28869)	(16429)	(18615)	(14830)	(16221)	(17374)	(16693.8)
	LCP	3.425	6.181	6.222	5.214	5.377	7.362	6.071
		(4900)	(16342)	(16811)	(13743)	(14595)	(20812)	(16460.6)

Table 3.13: Running times (in 10^{-3} sec) and number of iterations of Algorithm BF and LCP for 1-to-all static shortest path problems in random grid networks as a function of network size. The numbers in parentheses are the number of iterations

Chapter 4

Additional Ideas on Shortest Path Algorithms - Algorithm Delta and Algorithm Hierarchy

The previous chapter presents a description of an improvement in the area of shortest path algorithms, consisting of an algorithmic framework, its applications, and numerical tests to solve shortest paths in dynamics and static networks. In this chapter, we outline two additional ideas in the field of shortest path algorithms.

The first idea is motivated by the observation that in certain types of dynamic FIFO networks, shortest path trees for all departure times typically remain unchanged. This is typically the case if the number of arcs with time-dependent travel times (we later refer to such arcs as dynamic arcs) is a small portion of all arcs, and/or the frequency of changes in travel times in dynamic arcs is small. For the ease of discussion, we term arcs with time-dependent travel times as dynamic arcs. For a dynamic arc, we use link dynamics to describe the frequency of the changes in travel times. If the link travel time of a dynamic arcs changes frequently, we say the link dynamics are large. If the link travel time rarely changes, we say the link dynamics are small. We propose a way to compute the time interval during which a shortest path tree is always valid, thus avoiding the computation of shortest path trees during such intervals. The resulting algorithm is called Algorithm Delta and is outlined in

Section 4.1.

The second idea is motivated by an observation in shortest path problems in static networks. In a static network with n nodes and m arcs, the number of arcs in a shortest path tree is $n-1$, which means that $m-n+1$ arcs can be removed from the original network without affecting the correct computation of the shortest path tree. The idea involves the partitioning of the network into two layers. The shortest path tree is first computed in the lower layer and then the arcs in the higher layer are checked for the optimality condition. Algorithm Hierarchy is developed under this idea. Recall that in Chapter 3 static shortest path algorithms can be applied in dynamic FIFO networks to solve shortest path problems. Algorithm Hierarchy is then applicable in dynamic FIFO networks. Details of Algorithm Hierarchy is presented in Section 4.2.

4.1 Algorithm Delta

4.1.1 Problem Definition

We focus on the one to all minimum time path problem for all departure times in dynamic networks. When considering dynamic networks in practice, one can note the following characteristic. Not all the links are dynamic, that is, not all the links have time-dependent travel times, and for those do, there can be a significant amount of time between two consecutive changes. Therefore in a dynamic network, there may exist multiple periods of departure time from the origin, during which the network remains static. We term such an interval as a stationary period of duration Δ. In such a situation, the shortest path tree will remain unchanged and we need not spend time to recalculate it , because it has already been computed at the beginning of such a stationary period.

4.1.2 The Algorithm

The statements of the algorithm are shown in Figure 4-1. We denote the shortest path tree from source node at departure time t as T_t. The duration of the stationary period since departure time t is denoted as $\Delta(t)$. T is the maximum departure time.

```
ALGORITHM DELTA (N, A, s)
1  t ← 0 ;
2  while t ≤ T do
3     compute T_t ;
4     compute Δ(t) ;
5     t ← t + Δ(t) ;
6  endwhile
```

Figure 4-1: Statement of Algorithm Delta

The remaining question is how to calculate the duration of the stationary period $\Delta(t)$. For each node i, we examine the outgoing arcs. Starting from time 0, we record the time when the travel time along any of the outgoing arcs changes as $t_i(1), t_i(2), \cdots, t_i(k)$. $a_i(t)$ is the minimum arrival time at node i departing the source at time t. $a_i(t)$ must fall between an interval $[t_i(w), t_i(w+1))$. Let $\Delta_i(t) = t_i(w+1) - a_i(t)$ and $\Delta(t) = min_{i \in N} \Delta_i(t)$.

The shortest path tree computed at departure time t will remain as the shortest path tree for an interval of duration $\Delta(t)$. Because of the way $\Delta(t)$ is calculated, for departure times in interval $(t+1, t+\Delta(t)]$, the dynamic network is actually static. It comes with no surprise that the shortest path tree remains unchanged. A formal proof of the correctness of Algorithm Delta can be found in [56]. The computation of shortest path trees can be accomplished by any shortest path algorithms, for example, Algorithm LCP developed in Chapter 3.

We also note that $\Delta(t)$ has the following property.

Proposition 9 *If* $\Delta(t) > 1, \Delta(t+1) = \Delta(t) - 1$, *for* $t \in \{1, 2, \cdots, T_0 - 1\}$.

Proof: When $\Delta(t) > 1$, we have $\Delta(t+1) = min_{i \in N} \Delta_i(t+1) = min_{i \in N}(t_i(w+1) - a_i(t+1))$. Because $\Delta(t) > 1$, by the definition of $\Delta(t)$ we have $a_i(t+1) = a_i(t) + 1$. Therefore, $\Delta(t+1) = min_{i \in N}(t_i(w+1) - a_i(t) - 1) = min_{i \in N}(t_i(w+1) - a_i(t)) - 1 = \Delta(t) - 1$. ■

In the presentation of Algorithm Delta, we assumed that we work in increasing order of time, that is, we compute the shortest path trees in the order $T_1, T_2, \cdots, T_T$. Actually, we could have computed the shortest path trees in decreasing order of time. In such a condition, we have to re-define $\Delta_i(t)$ as $a_i(t) - t_i(w)$.

4.1.3 Experiment Evaluation

The objective of this section is to investigate the duration of Δ and perform some preliminary sensitivity analysis with respect to the size of the network, the percentage of dynamic arcs, and link dynamics. The effectiveness of Algorithm Delta highly depends on the duration of $\Delta(t)$. The larger the $\Delta(t)$, the better the performance.

The test networks are random networks (Type I) with $m = 3n$. As is stated in Section 3.6, two parameters α and β are used to describe the dynamics of the network. Figure 4-2 through Figure 4-10 show the curves of the Δ. If there is a jump in the curves, a new shortest path tree need to be computed. For example, in the second figure in Figure 4-3, two shortest path trees should be computed for departure time 0 and 24 respectively. One can note that the magnitude of Δ is very sensitive to the size of the network. When we increase n from 10 to 1,000, Δ decreases dramatically. If $n = 1,000$ even when the percentage of dynamic arcs α is 5% and $\beta = 0.005$, the magnitude of Δ is almost 1.

We also note that the product of α and β determines the appearances of the curves. The figures in Figure 4-3 have similar patterns to the figures in Figure 4-5 when n is the same, because both of them has $\alpha \times \beta = 0.0005$. The figures in Figure 4-6, and the figures in Figure 4-8 enjoy similar patterns when n is the same, because they all have $\alpha \times \beta = 0.001$. This does make sense as the total number of arc travel time changes is proportional to the product of α and β and the total number of arc travel time changes directly determines the magnitude and trend of Δ. The experiment tests show that Algorithm Delta can be quite efficient in networks where the total number of arc travel times is small.

Another observation is that the negative slope of the curves are always -1, which is consistent with Proposition 9.

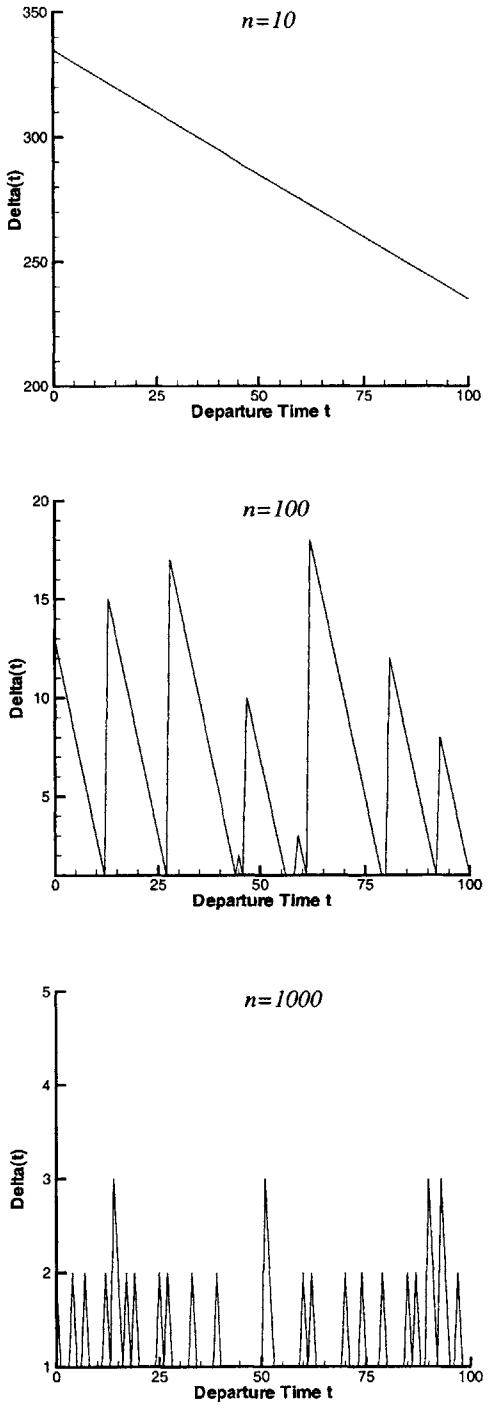

Figure 4-2: Δ as a function of the size of the network. $\alpha = 0.05, \beta = 0.005, m = 3n$

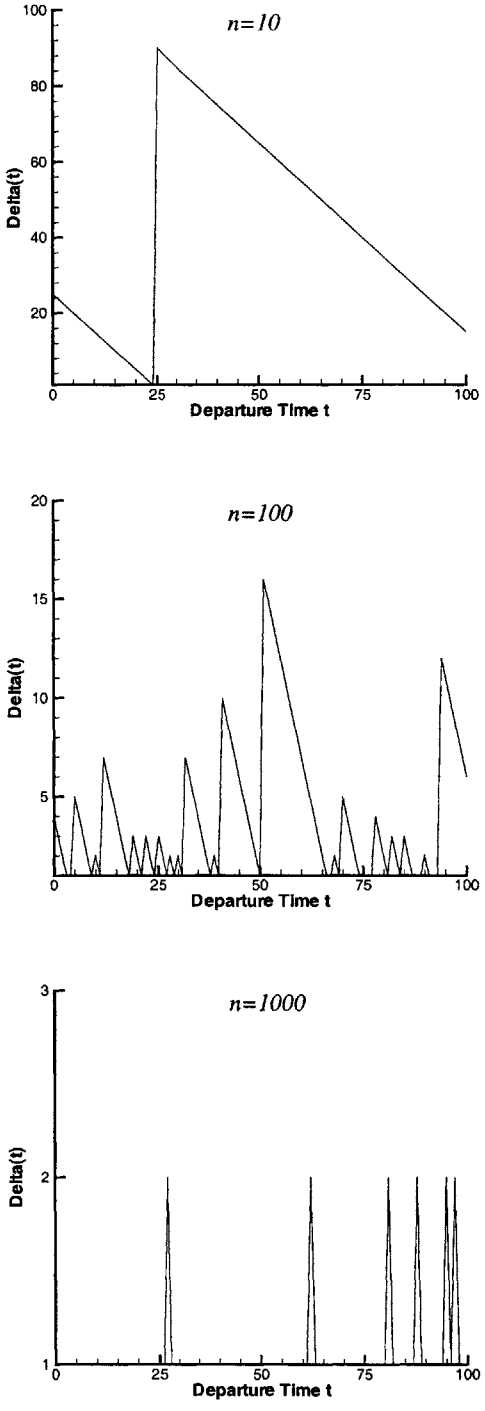

Figure 4-3: Δ as a function of the size of the network. $\alpha = 0.05, \beta = 0.01, m = 3n$

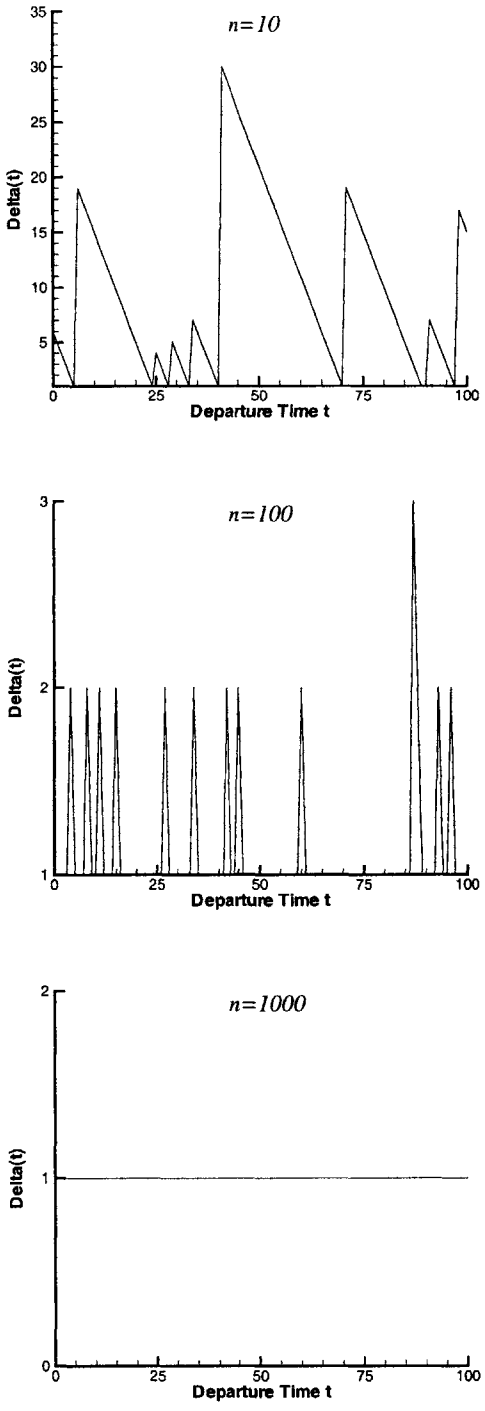

Figure 4-4: Δ as a function of the size of the network. $\alpha = 0.05, \beta = 0.05, m = 3n$

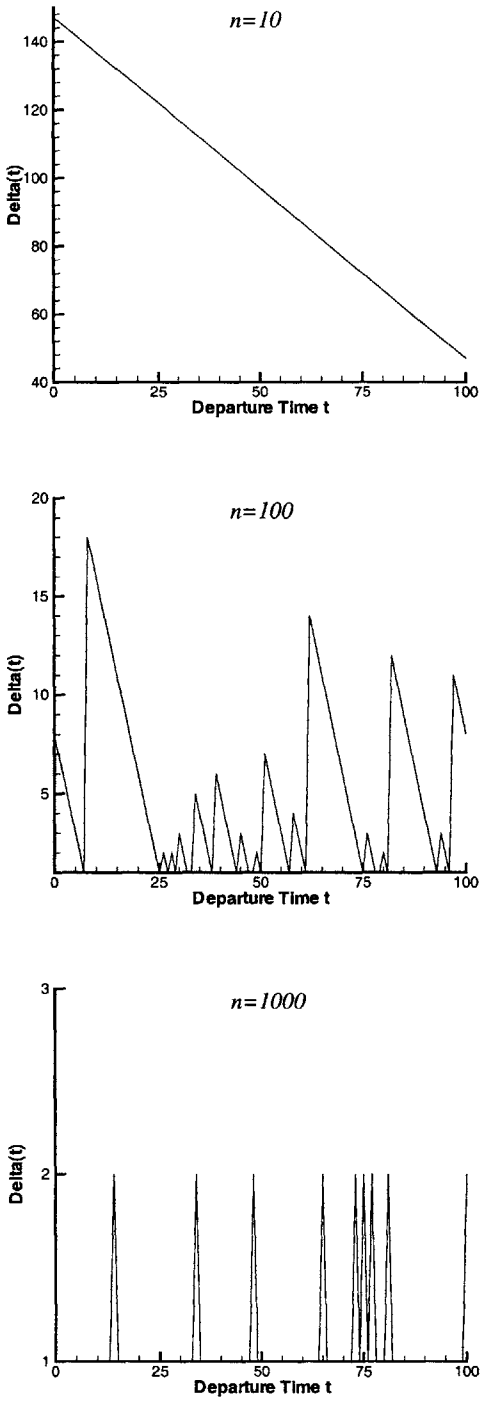

Figure 4-5: Δ as a function of the size of the network. $\alpha = 0.1, \beta = 0.005, m = 3n$

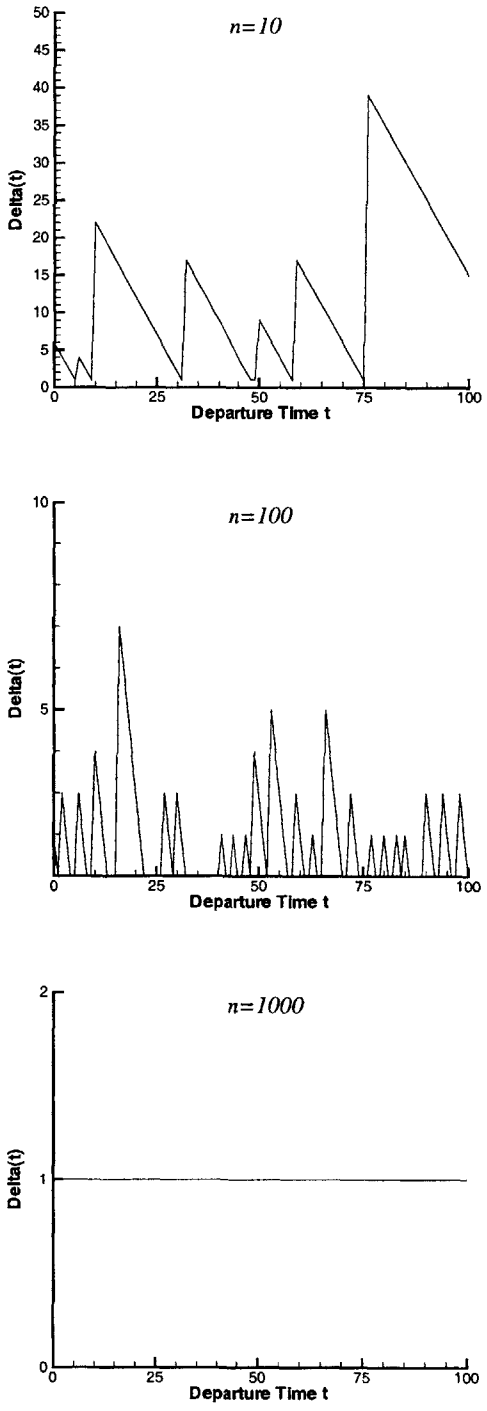

Figure 4-6: Δ as a function of the size of the network. $\alpha = 0.1, \beta = 0.01, m = 3n$

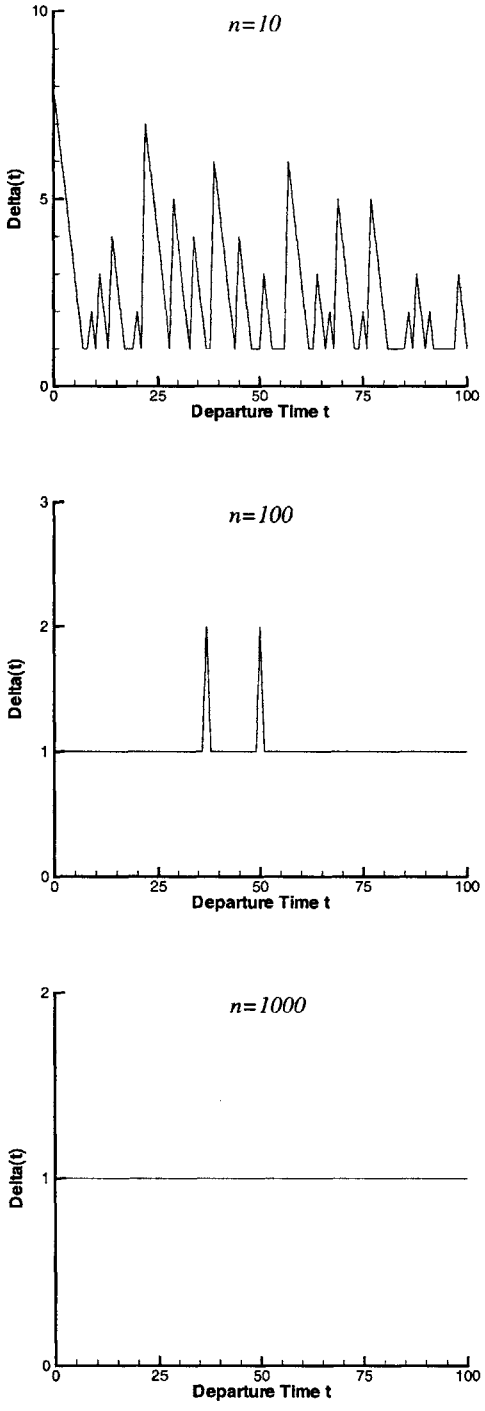

Figure 4-7: Δ as a function of the size of the network. $\alpha = 0.1, \beta = 0.05, m = 3n$

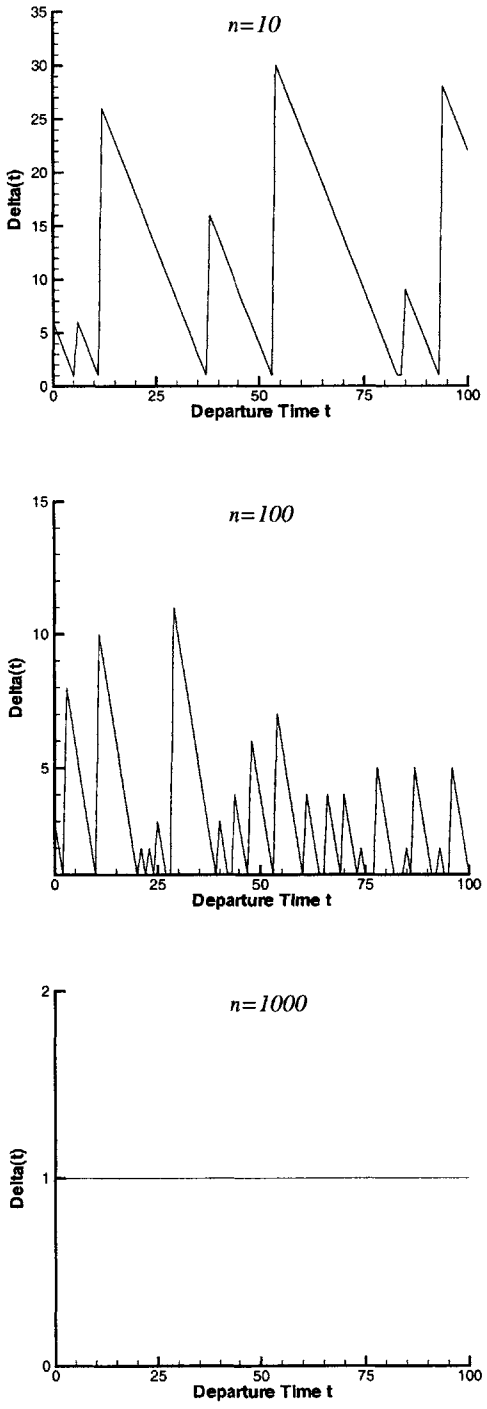

Figure 4-8: Δ as a function of the size of the network. $\alpha = 0.2, \beta = 0.005, m = 3n$

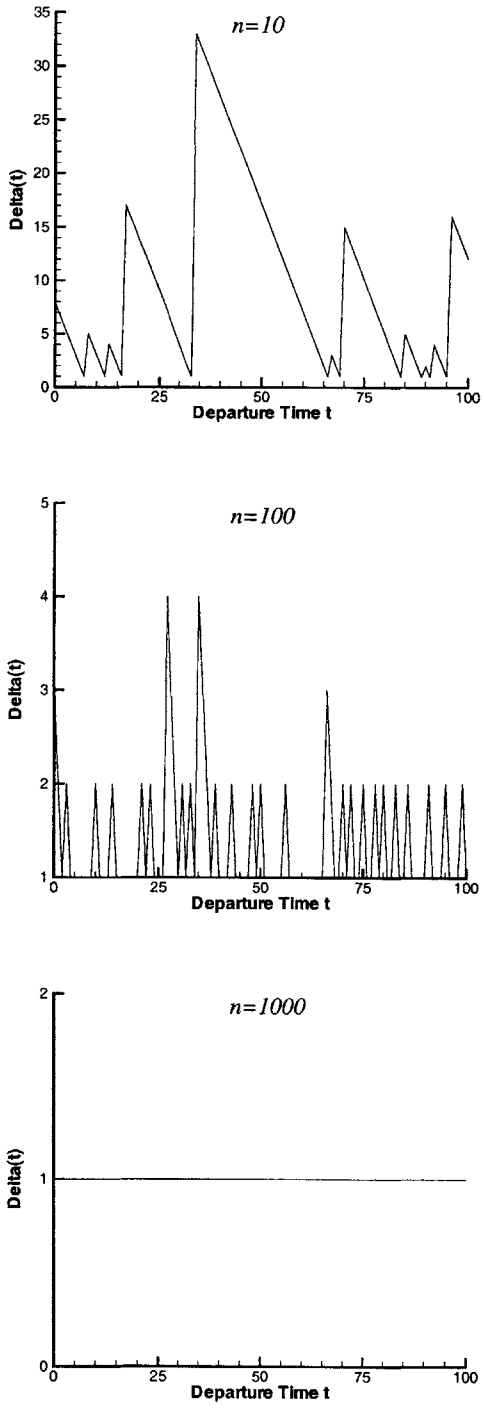

Figure 4-9: Δ as a function of the size of the network. $\alpha = 0.2, \beta = 0.01, m = 3n$

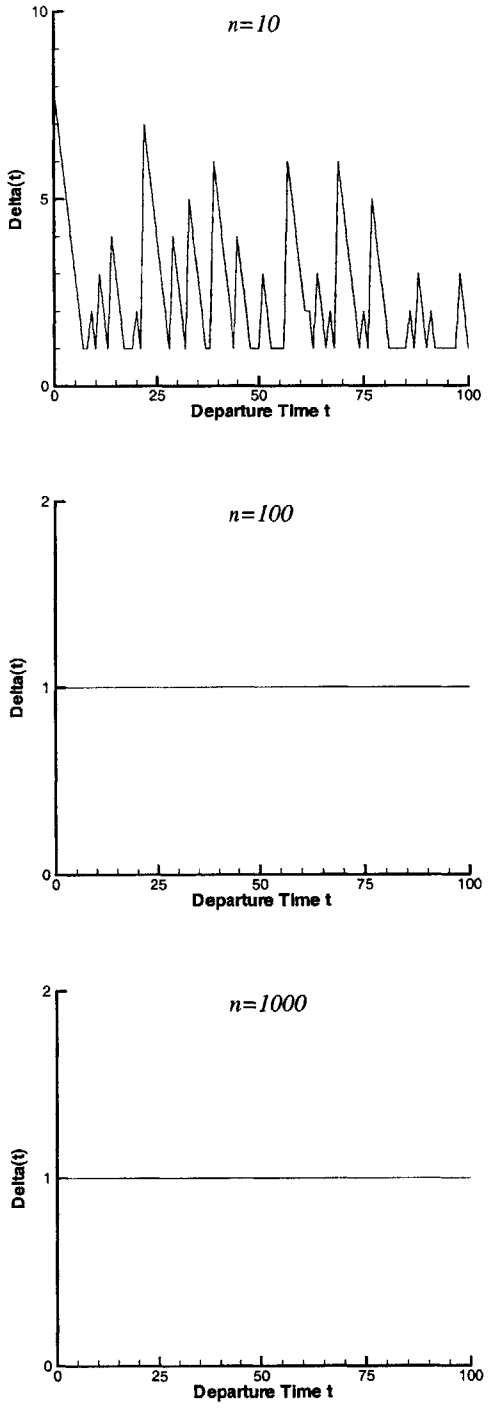

Figure 4-10: Δ as a function of the size of the network. $\alpha = 0.2, \beta = 0.05, m = 3n$

4.2 Algorithm Hierarchy

In this section, we start from the one-to-all shortest path problem in static networks. In sparse networks, the queue implementation of Bellman-Ford algorithm (Algorithm BF) [47, 48] is very efficient in practice and performs much better than the binary-heap implementation of Dijkstra's algorithm [45, 46]. However, when the network is dense, the performance of Algorithm BF degrades. Algorithm BF spends a lot of time scanning the forward star of each node in the scan eligible list, although only one of those outgoing arcs will actually appear in the shortest path tree.

4.2.1 The Algorithm

An idea to improve the performance of Algorithm BF is to partition the arcs in the network into two hierarchical layers. The lower level layer A_l contains arcs with small link travel times; while the higher level layer $A_h = A \setminus A_l$ contains arcs with large link travel times. We want to keep as few arcs in A_l as possible while most or all of the arcs in the shortest path tree are contained in A_l. The algorithm is shown in Figure 4-11. Although it is motivated by the bad performance of Algorithm BF in dense networks, it can also speed up the algorithm in sparse networks if A_l is well chosen.

There are several ways to partition A. For example, find a threshold value $d_{threshold}$. Let $A_l = \{(i,j)|d_{ij} \leq d_{threshold}, (i,j) \in A\}$, $A_h = \{(i,j)|d_{ij} > d_{threshold}, (i,j) \in A\}$.

Algorithm Hierarchy can be viewed as a strategy of prioritization of links in shortest paths computation. The set of links in the lower level layer are those links that has a higher probability to appear in the shortest path tree. Whenever a node is picked and scanned for its forward star, we scan its forward star in the lower level layer first and delay the scan of its forward star in the higher level layer. The forward stars in the higher level layer are only scanned when all the links in the lower level layer satisfy the optimality condition. If the shortest path tree computed in the lower level is the real shortest path tree for the network, then the higher level layer links will be just scanned only once to check their optimality.

```
ALGORITHM HIERARCHY (N, A, A_l, A_h, s)
1   d_s ← 0 ;
2   d_j ← ∞ for each j ∈ N \ {s} ;
3   compute the shortest path tree in (N, A_l) ;
4   SE ← N; PSE ← ∅ ;
5   while SE ≠ ∅ do
6     remove a node i from SE ;
7     for j ∈ A_h(i) do
8       if d_j > d_i + d_ij then
9         PSE ← PSE ∪ {j} ;
10        while PSE ≠ ∅ do
11          remove a node i from PSE ;
12          for j ∈ A_l(i) do
13            if d_j > d_i + d_ij then
14              PSE ← PSE ∪ {j} ;
15              SE ← SE ∪ {j} ;
16            endif
17          endfor
18        endwhile
19      endif
20    endfor
21  endwhile
22  return the shortest path tree
```

Figure 4-11: Statement of Algorithm Hierarchy

4.2.2 Runtime Complexity Analysis

There are two loops in the statements of Algorithm Hierarchy shown in Figure 4-11. The outer loop is from line 6 to line 21. The inner one is from line 11 to 18. Suppose that the outer loop is executed at most n_h times, and the inner loop is executed at most n_l times in each outer loop. Note that $O(n_l)$ is also the bound for the number of node visits in line 4. We have the following proposition:

Proposition 10 *The runtime complexity of Algorithm Hierarchy is in* $O(n_l|A_l| + n_h(|A_h| + n_l|A_l|))$.

Proof: The first term $O(n_l|A_l|)$ comes from line 4. In one iteration of the inner loop, $O(|A_l|)$ arcs are scanned. In one iteration of the outer loop, $O(|A_h| + n_l|A_l|)$ arcs are scanned. Therefore the complexity for the outer loop is $O(n_h(|A_h| + n_l|A_l|))$. The overall complexity is $O(n|A_l| + n_h(|A_h| + n_l|A_l|))$.

We now investigate the complexity of Algorithm Hierarchy in different situations. Note that there are two parameters n_l and n_h in the expression.

The best-best case complexity: If A_l is exactly the set of arcs in the shortest path solution, $n_l = 0, n_h = 1, A_l = n-1$, and $A_h = m-n+1$. Therefore the runtime complexity is in $O(m)$.

The worst-worst case complexity: We know that both n_h and n_l is in order of n. Therefore the worst worst case complexity is $O(nm + n^2|A_l|)$.

The best-worst case complexity: If A_l contains the shortest path tree and is in $\Omega(n)$, we obtain the best-worst case. The complexity is $O(|A_l|n + |A_h|)$. The time to find the shortest path tree in the lower level graph (N, A_l) is $O(|A_l|n)$. The time spent to check whether the arcs in the higher level verify the optimality condition will cost $O(|A_h|)$. Therefore the overall complexity is $O(|A_l|n + |A_h|)$.

4.2.3 Experimental Evaluation

Algorithm Hierarchy performs very well in dense networks. Figure 4-12 shows some statistics. The test networks are fully dense, which mean $m = n(n-1)$ and the maximum link travel time is 100. The shortest path trees are computed and the figure shows the histograms of the link travel times of the links in the shortest path trees. When $n = 50$, the histogram is flat, which means it is hard to find A_l such that it contains the shortest path tree while has a small number of arcs. However, when $n = 300$, the histogram is shifted to the left and the counts decreases sharply when the link travel time increases. If we set $d_{threshold} = 30$, the shortest path tree will be contained in A_l, therefore we will expect a speedup around 3 compared to the Bellman-Ford Algorithm.

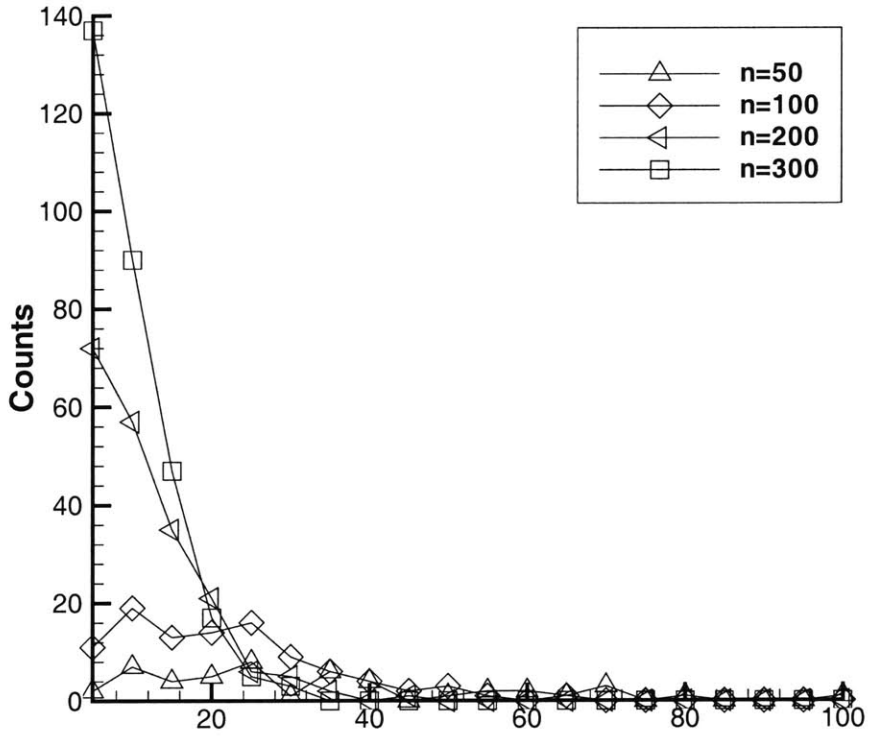

Figure 4-12: The histograms of the link travel times of the links in the shortest path tree in fully dense networks, that is, $m = n(n-1)$. The link travel times vary between 1 and 100

4.2.4 Application in One-to-all Dynamic Shortest Path Problems

The area of one-to-all dynamic shortest path problem for all departure times in FIFO networks provides another venue for Algorithm Hierarchy. For departure time t, we can use the shortest path tree found in time $t-1$ as A_l while let all the other arcs be A_h. Recall that in Chapter 3, we defined T_t as the shortest path tree for departure time t and T as the maximum departure time. The algorithms is shown in Figure 4-13.

```
ALGORITHM HIERARCHY (N, A, T, s)
1   Compute the shortest path tree T_1 for departure time1 ;
2   for t ← 2 to T do
3     T_t ← ALGORITHM HIERARCHY(N, A, T_{t-1}, A \ T_{t-1}, s) ;
4   endfor
```

Figure 4-13: Algorithm hierarchy applied in 1-to-all dynamic shortest path problems for all departure times

Chapter 5

Conclusions and Future Research Directions

5.1 Contributions and Major Results

Dynamic Traffic Assignment (DTA) has been an intriguing topic in the past two decades. It plays a central role in the simulation of traffic, either in real time applications or for off-line purposes. It covers a broad range of research areas, including shortest paths, dynamic network loading, traffic flow theory, and discrete choice theory. In recent years, there is a heightened interest in deploying DTA models in large-scale networks and achieve real-time traffic management, which requires the development of DTA models that can solve large-scale DTA problems fast. The main objective of this thesis is to develop fast DTA models.

A DTA model can be decomposed into several sub-models: the user's route choice model, where a dynamic shortest path problem is imbedded; the dynamic network loading model, where the link flows and link times are computed from path flows. For each model, there is a corresponding algorithm. For the user's route choice model, we have the user's route choice algorithm; for the dynamic network loading model, we have the dynamic network loading algorithm. In order to reach equilibrium for the DTA model itself, there is also a DTA solution algorithm.

The research approaches we took is to develop fast DTA sub-models to improve the

overall efficiency of the DTA model. To be specific, in Chapter 2, we developed parallel implementations of the dynamic network loading algorithm and the DTA solution algorithm. To the best of our knowledge, this concerns the first parallel implementations of macroscopic DTA models. Two network loading algorithms are studied: the iterative network loading algorithm (I-Load) and the chronological loading algorithm (C-Load). We developed two decomposition strategies for the iterative loading algorithm. A network topology decomposition strategy and a time-based decomposition strategy are tested in a distributed-memory platform using the Amsterdam A10 Beltway network example. Numerical results show that for the network topology decomposition strategy, a speedup of 5 is observed when the number of processors is 10 and the asymptotic speedup is about 10. For the time-based decomposition strategy a speed-up of 6.5 is observed when the number of processors is 10 and the asymptotic speed-up is about 25. For the chronological loading algorithm, the network topology decomposition strategy is tested in the same distribute-memory platform. The speedup is not significant due to the highly efficient sequential algorithm and the small size of the test network.

In Chapter 3 we proposed a new framework for static shortest path algorithms. The characteristic of this framework is that it allows the prioritization of nodes with optimal distance labels (or close to optimal distance labels), which means nodes with optimal distance labels in the scan eligible list is pulled out before processing the other nodes. This framework is applied in dynamic FIFO and strict FIFO networks to design efficient one-to-all dynamic shortest path algorithms. Numerical tests are done in two types of random networks: networks with random topology and networks with grid topology. Computational results show that the new framework achieved significant speedup (up to a factor of 4) compared to a repetitive application of the queue implementation of Bellman-Ford algorithm. We extended our discussion to the one-to-all, one-to-one and many-to-all shortest path problem in static networks. The average speedup in the on-to-one problem is about 2. In the many-to-all problem, the speedup highly depends on the position of the rest source nodes in the shortest path tree rooted at the first source node; the average speedup is about 1.2.

In Chapter 4, we continued our discussion in the area of shortest paths. We presented two ideas that could potentially lead to more efficient algorithms. The first idea originates from the observation that in one-to-all fastest path problem for all departure times in FIFO networks, the shortest path tree may remain unchanged for a period of time. This means that we need not recompute the shortest path tree for departures during that interval. Algorithm Delta is developed based on this idea. Experimental evaluation showed that the performance of Algorithm Delta depends on the size of the network and its dynamics. The smaller the network and/or the less the dynamics of the network, the better the performance. The second idea is related to improvements in static shortest path algorithms. It partitions the network into two layers and solves the shortest path problem by moving back and forth between the two layers. Algorithm Hierarchy is developed and the theoretical complexity is analyzed.

5.2 Future Research Directions

In this section, we summarize future research directions related to the topics covered in this thesis.

For the parallel implementations of DTA models in Chapter 2, future research can be categorized into two directions: the improvements of the DTA models and the improvements of the parallel implementations. Several limitations of the DTA models are identified in [5], including the modelling of incidents, queues, spill-backs, and the method to update path flows. It would also be useful to use a more sophisticated link performance model. In the DTA models used in this thesis, a volume delay function link model is used, which is a fairly simple one. It does not take into consideration the distribution of traffic along links and this poses a potential threat to the accuracy of the loading results. A more realistic link performance model shall be used in the implementation.

On the side of parallel implementations, it would be useful to design better load partition algorithms. The partition algorithms presented in the thesis are heuristics

and can be improved. It is would also be interesting to investigate the feasibility of hybrid parallel implementations. In this thesis, we looked at distributed-memory implementations and shared-memory implementations separately. One can also consider a hybrid implementation combining the two and take advantages from both.

For the shortest path algorithms developed in Chapter 3, several directions can be identified. 1) Both SE and PSE are implemented using a queue; however, they could be implemented using other data structures, for example, dequeue. It is interesting to see how the algorithms perform with different data structures. 2) A new method should be sought to determine the frequency of the $findmin$ operation when the framework is applied in 1-to-all problems in static networks. 3) In the computational tests, it is helpful if we can compare Algorithm LCP with other algorithms besides Algorithm BF.

Appendix A

More on Parallel Implementations

The appendix is provided to get the reader started. We briefly introduce how to set up the parallel computing environment, compile, and run the parallel codes. We do not cover any MPI and Pthread related functions, because we believe it is much easier and more convenient for interested readers to refer to reference books, such as [22, 23, 25, 26, 27] if they want to look at the source code. For an extensive description of the format of the input and output files, please refer to [5].

A.1 Distributed-memory Implementations

This distributed-memory implementations in this thesis are developed using MPICH, the portable implementation of MPI. It is available for download at *http://www-unix.mcs.anl.gov/mpi/mpich/*. We used MPICH-1.2.5 in the development and the operating system is Red Hat Linux 8. The installation is quite straight forward following the instructions in the Installation Guide. If we install MPICH in /var/local/mpich and use SSH, the commands are:

```
%./configure -prefix=/var/local/mpich -rsh=ssh
%make
%make install
```

We use the GNU g++ compiler to compile the source code against MPICH li-

braries. The command is:

```
%g++ -Wall -o dta dta.cpp -I/var/local/mpich/include -lmpich -
L/var/local/mpich/lib
```

To run the code, use the command *mpirun*:

```
%mpirun -np <n> -machinefile <machinelist> dta <arguments...>
```

The meaning of the arguments are:

-np <n>	specify the number of processors to run on
-machinefile <machinelist>	Take the list of possible machines to run on from the file <machinelist>
<arguments...>	Supply input to the dta program, for example, the number of iterations, the name of the control file, maximum demand time, etc. If no argument is supplied, the program outputs the format of input and quits.

A.2 Shared-memory Implementations

The shared-memory implementations in this thesis are developed using POSIX Thread. The pthread library is available in Red Hat Linux 8, therefore there is no need for installation.

To compile the code against pthread library, use command:

```
%g++ -Wall -o dta dta.cpp -lpthread
```

To run the code, use command:

```
%dta <arguments...>
```

If no argument is supplied, the program outputs the format of input and quits.

Bibliography

[1] D. Schrank and T. Lomax. The 2001 Urban Mobility Report. Technical report, Texas Transportation Institute, The Texas A M University, 2001.

[2] Energy Information Administration. Annual Energy Outlook 2003 With Projections to 2025. Technical Report DOE/EIA-0383(2003), Department of Energy, January 2003. also online at http://www.eia.doe.gov/oiaf/aeo.

[3] C.D.R. Lindveld. *Dynamic O-D matrix estimation.* PhD thesis, Delft University of Technology, 2003.

[4] K. Nagel and M. Rickert. Parallel Implementation of the TRANSIMS Microsimulation. *Parallel Computing*, 27(12):1611–1639, Novermber 2001.

[5] Y. He. A Flow-Based Approach to The Dynamic Traffic Assignement Problem: Formulations, Algorithms and Computer Implementations. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1997.

[6] http://web.mit.edu/its/dynamit.html.

[7] http://www.ce.utexas.edu/prof/mahmassani/dynasmart-x/.

[8] H.J. Payne and H.M. Koble. A comparison of Macroscopic Simulation Models (MACK and INTRAS). Technical Report OCR-78-11-9367-1, ORINCON Corp., March 1978.

[9] D.A. Wicks. INTRAS - A Microscopic Freeway Corridor Simulation Model. *Overview of Simulation in Highway Transportation*, 1:95–107, 1977.

[10] D.A. Wicks. Development and Testing of INTRAS, A Microscopic Freeway Simulation Model. *Program Design and Parameter Calibration*, 1, 1980.

[11] C.F. Daganzo. The Cell-Transmission model: A Dynamic representation of highway traffic consistent with hydrodynamic theory. *Transp. Res.*, 28B(4):269–287, 1994.

[12] N.J. Grier. Improved Methods for Solving Traffic Flow Problems in Dynamic Networks. Master's thesis, Massachusetts Institute of Technology, 2001.

[13] H.J. Payne. FREFLO - A Macroscopic Simulation Model of Freeway Traffic. *Transportation Research Record*, 722:68–75, 1979.

[14] R.D. Coombe, T.J. Annesley, and R.P. Goodwin. The Use of CONTRAM in Bahrain. *Traffic Engineering and Control*, March 1983.

[15] D. Leonard, P. Gower, and N. Taylor. CONTRAM: Structure of the Model. *TRRL Research Report*, (178), 1989.

[16] B. Ahn and J. Shin. Vehicle Routing with Time Windows and Time-varying Congestion. *Journal of Operational Research Society*, 42:393–400, 1991.

[17] S.E. Dreyfus. An Appraisal of Some Shortest Path Algorithms. *Journal of Mathematical Analysis and Applications*, 14:492–498, 1969.

[18] Intelligent Transportation Systems Program. Development of a Deployable Real-Time Dynamic Traffic Assignment System: Executive Summary DynaMIT and DynaMIT-P. Technical report, Massachusetts Intitute of Technology, June 2000.

[19] D.K. Merchant and G.L. Nemhauser. A Model and an Algorithm for the Dynamic Traffic Assignment Problems. *Transportation Science*, 12(3):183–199, August 1978.

[20] D.K. Merchant and G.L. Nemhauser. Optimality Conditions for a Dynamic Traffic Assignment Model. *Transportation Science*, 12(3):200–207, August 1978.

[21] Y. Sheffi. *Urban Transportation Networks.* Prentice-Hall, Englewood, New Jersey, 1985.

[22] D.R. Butenhof. *Programing with POSIX Threads.* Addison-Welsley professional computing series. Addison-Welsley, September 2002.

[23] B. Nichols. *Pthread Programming.* O'Reilly Associates, Inc, 1996.

[24] Institute of Electronical and Electronics Engineers. *Draft Standard for Information Technology-Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language], Draft.* October 1993.

[25] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, Cambrigde, Massachusetts, second edition edition, 1999.

[26] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, Massachusetts, second edition edition, 1998.

[27] http://www-unix.mcs.anl.gov/mpi/mpich/.

[28] A. Chronopolous and P. Michalopoulos. Traffic Flow Simulation Through Parallel Processing. Technical report, Center for Transportation Studies, Minnesota University, Minneapolis, MN, 1991.

[29] A. Hislop, M. McDonald, and N. Hounsell. The Application of Parallel Processing to Traffic Assignment for Use with Route Guidance. *Traffic Engineering and Control*, pages 510–515, 1991.

[30] G.N. Frederickson. A Distributed Shortest Path Algorithm for a planar Network. *Information and Computation*, 86:140–159, 1990.

[31] Y. Xu, J. Wu, and M.A. Florian. The Continuous Time-Dependent Shortest Trail Problem with Turn Penalties: Model, Algorithm and a Parallel Implementation

under the PVM Environment. *INFORMS/CORS Spring 1998 Joint Meeting*, April 26-29 1998.

[32] I. Chabini and S. Ganugapati. Parallel Algorithms for Dynamic Shortest Path Problems. *International Transactions in Operational Research*, 9(3):279–302, May 2002.

[33] G.L. Chang, T. Junchaya, and A.J. Santiago. A real-time network traffic simulation model for ATMS applications: Part I – simulation methodologies. *IVHS Journal*, 1(3):227–241, 1994.

[34] W. Niedringhaus, J. Opper, L. Rhodes, and B. Hughes. IVHS traffic modeling using parallel computing: performance results. *Proceedings of the International Conference on Parallel Processing*, pages 688–693, 1994.

[35] G. Karypis and V. Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparce Matrices. University of Minnesota, Department of Computer Science/ Army HPC Research Center, Minneapolis, MN 55455, September 1998.

[36] I. Chabini. Discrete Dynamic Shortest Path Problems in Transportation Applications: Complexity and Algorithms with Optimal Run Time. *Transportation Research Record*, (1645):170–175, 1999.

[37] http://www.cut-the-knot.org/do_you_know/isoperimetric.shtml.

[38] http://www-users.cs.umn.edu/ karypis/metis/metis/faq.html.

[39] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications*, chapter 3, pages 73–76. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, first edition, 1993.

[40] I. Chabini and B. Gendron. Parallel Performance Measures Revisited. *Proceedings of High Performance Computing symposium 95*, Montreal, Canada, 1995.

[41] K. Ashok. *Estimation and Prediction of Time-Dependent Origin-Destination Flows.* PhD thesis, Massachusetts Institute of Technology, 1996.

[42] http://www.myri.com/.

[43] http://www.myri.com/open-specs/.

[44] G. Gallo. Reoptimization Procedures in Shortest Path Problems. *Rivista di Matematica per le Scienze Economiche e Sochiali*, 3:3–13, 1980.

[45] E.W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numeriche Mathematik*, 1:269–271, 1959.

[46] G. Gallo and S. Pallottino. Shortest Path Algorithms. *Annals of Operations Research*, 7:3–79, 1988.

[47] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[48] L.R. Ford Jr. *Network Flow Theory.* The Rand Corportation, Santa Monica, Cal., 1956.

[49] F. Glover, R Glover, and D. Klingman. The threshold shortest path algorithm. *Networks*, 14(1), 1986.

[50] B. Golden. Shortest-Path Algorithms: A Comparison. *Operations Research*, 44:1164–1168, 1976.

[51] I. Chabini and S. Lan. Adaptions of the A* Algorithm for the Computation of Fastest Paths in Deterministic Discrete-Time Dynamic Networks. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):60–74, March 2002.

[52] F. Glover, R. Glover, and D. Klingman. The Threshold Shortest Path Algorithm. *Networks*, 14(1), 1986.

[53] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications*, chapter 2, pages 43–44. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, first edition, 1993.

[54] H. Tan. *C programming.* Tsinghua Univeristy Press, Beijing, China, 1991.

[55] S. B. Lippman and J. Lajoie. *C++ Primer.* Addison-wesley, third edition, 1999.

[56] I. Chabini. Computing Shortest Path Trees for all Departure Times in Discrete Time Dynamic FIFO Networks. *Internal Report*, May 2003.