Scalable, Repeatable, and Contention-Free Parallelization of Traffic Simulation

by Cordelia Avery

S.B., Computer Science and Engineering, M.I.T., 2017

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

Massachusetts Institute of Technology

June 2018

© 2018 Cordelia Avery. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author: ______________________________
Department of Electrical Engineering and Computer Science
May 25th, 2018

Certified by: ______________________________
Moshe Ben-Akiva, Professor of Civil and Environmental Engineering, Thesis Supervisor
May 25th, 2018

Certified by: ______________________________
Andrea Araldo, Postdoctoral Associate, Thesis co-Supervisor
May 25th, 2018

Accepted by: ______________________________
Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Scalable, Repeatable, and Contention-Free Parallelization of Traffic Simulation

by

Cordelia Avery

Submitted to the Department of Electrical Engineering and Computer Science
on May 25th, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Tripod is a project funded by ARPA-E and partly carried on by the Intelligent Transportation Systems (ITS) Lab at MIT that aims to promote more energy efficient travel options by offering commuters incentives to make smart travel choices. These incentives depend on the current network state, and the ability to estimate the state of a given road network in real time is crucial. It relies on the DynaMIT system to determine what these incentives ought to be in order to optimize traffic flow on the network. Developed by the ITS lab, DynaMIT uses simulation to compute the current network state, predict its state in the future and, by extension, compute the incentives to travelers that optimize the global energy gain. While DynaMIT is able to do this effectively within smaller areas, it is unable to simulate traffic for the Greater Boston Area, or GBA, due to the scale of the network. The goal of this thesis is to scale the DynaMIT system so that it is less affected by network sizes. First, we outline a custom, lightweight profiling tool that is able to better track down the problems with scalability; next, we build off of previous work to address design errors that slow serial execution time; and finally, we implement a novel way to parallelize traffic simulation that avoids the race conditions and concurrency issues generally associated with such systems.

Thesis Supervisor: Moshe E. Ben-Akiva
Title: Edmund K. Turner Professor of Civil and Environmental Engineering

Thesis Supervisor: Andrea Araldo
Title: Postdoctoral Associate

Acknowledgments

I would like to acknowledge several people who made the completion of this thesis possible.

First and foremost, I would like to thank Andrea Araldo, my direct supervisor, without whom this research would never have been possible. He continually supported and advised me during this process, helping me to refine and continue to refine the work that is presented here. I would also like to thank Professor Moshe Ben-Akiva, my thesis supervisor, for encouraging me to strive to achieve my utmost during my time in the ITS Lab. During my time here, I feel I have not only gained experience in research, but also seen my drive and self-discipline grow.

I also owe a debt of gratitude to the ITS Lab in general, whose members made my time in this program enjoyable, and the pressure that often arises as a graduate student manageable.

I would also like to thank my undergraduate and graduate academic advisors, Albert Meyer and Michael Carbin, who continually guided and supported me throughout my time at this Institute, and my teachers Gordon Campbell and Vance Condie, who were pivotal to my interests in science and my choice to matriculate here.

These acknowledgments would not be complete without mentioning my roommates, who were forever supportive during this process. They were willing to listen to and discuss my research despite it being unrelated the theirs, were there to encourage me to take a break when needed, and overall have acted as and will continue to be incredible friends.

Finally, I would like to thank family, who have always been there to support me throughout my academic career. I have my parents in particular to thank not only for encouraging me to continue my academic pursuits, but also for raising me to be strong enough to have reached the point where I am today. I am incredibly fortune to have grown up in an environment where, as a young girl fascinated with math and science, I was encouraged to pursue my interests and dreams with confidence. I would like to thank my mother Elis-

abeth and my grandmothers Dorothy and Antonia for raising me to be a strong woman, never leaving a doubt in my mind that I deserve to be here; my father John for always pushing me to be the best I can be; my grandfather George for teaching me to laugh through everything, good and bad; and, particular, my grandfather John for continually and wholeheartedly supporting me in my academic career. I would never have accomplished what I have without all of them in my life.

Contents

1 Introduction 13
1.1 Main Objectives . . . 14
1.2 Thesis Contributions . . . 16
1.3 Outline . . . 17

2 Related Work 19
2.1 Previous work in DynaMIT . . . 21

3 The DynaMIT System 23
3.1 State Estimation and Prediction/Guidance . . . 25
3.2 Network Representation . . . 27
3.2.1 Initializing Network Topology . . . 31
3.2.2 Initializing Supply . . . 31
3.3 Path Representation . . . 32
3.4 Simulation Control Flow . . . 34

4 Necessary Properties for DynaMIT 37
4.1 Running Time Efficiency . . . 38
4.2 Repeatability of Parallel Traffic Simulations . . . 40
4.3 Maintainability . . . 42

5 Serial-execution Optimizations 43
5.1 Profiling . . . 43
5.1.1 Updated profiler design . . . 44
5.2 Optimizing Network Initialization . . . 46
5.3 Memory Locality Management . . . 48

6 Parallelization **51**

6.1 Goals 53

6.2 Design 54

6.2.1 Node Pipelining 54

6.2.2 Parallelization via Vertex Coloring 56

6.3 Update and Independence Properties 57

6.3.1 Update Properties 58

6.3.2 Independence Properties 60

6.4 Node-Pipelined Algorithm 62

6.4.1 Reduction to Vertex Coloring 64

6.4.2 Notes on Node Ordering 67

6.5 Implementation on a Multi-Core Machine 69

6.5.1 Updated Network Representation 69

6.5.2 Generating Network Bands 71

7 Results **77**

7.1 Serial-implementation Results 77

7.1.1 Datatype Improvements 77

7.1.2 Memory Allocation 78

7.1.3 Updates to Network Representation 78

7.2 Parallelization 79

7.2.1 Running time evaluation 79

7.2.2 Determinism 80

7.3 Moving Forward 81

7.3.1 General Changes to the Code 81

7.3.2 Memory Use 82

7.3.3 Parallelization 82

List of Figures

3-1 Example of hierarchy of network topology structure. 30

3-2 Memory ordering in current DynaMIT implementation 33

3-3 Pseudocode for the `advanceTraffic` function. 35

4-1 Demand, scaled logarithmically, vs. running time in GBA. 40

5-1 Sample call graph for inline profiling. 45

5-2 Sample running times for inline profiler. 47

5-3 Updated order of network elements in memory. 49

6-1 Example of a good candidate for pipelining. 54

6-2 Example pipelined execution for basic system. 55

6-3 Nodes p, p' and k with their corresponding update ranges. 62

6-4 Examples of band dependence graphs in 1D and 2D. For non-color readers, color is denoted for band b_i by $(i \mod 2^D)$. 67

6-5 Perfectly balanced partition with node weight 1 and insufficient band spacing. 72

6-6 Pseudocode for the updated `advanceTraffic` function. 73

6-7 2D partition with axes balanced independently in x-then-y order. 75

7-1 Sample log output for determinism validation. 81

List of Tables

3.1 Sample states for rolling horizon. 26

4.1 Sample running times for main DynaMIT functions over 6 simulation intervals in Boston CBD. 39

5.1 Initialization times for DynaMIT networks. 46

6.1 Execution with 10 bands (5 processors), with the current *Advance* process (abbreviated as A) for $s = [0, ..., 12]$. Note that, for every execution of some $A(i, t)$, $S(i-1, t)$ and $S(i+1, t-1)$ will have been set on the previous clock period, and are constant during the current clock period. Completion of the final node in a time step is marked in blue. 64

6.2 Sample execution for 5 processors using alternating banding. 66

7.1 Running times for original and improvement system loading times on CBD and GBA. 77

7.2 Running times for 6 simulation intervals before and after changes to memory allocation. 78

7.3 Running times for 6 simulation intervals before and after updates to the network representation. 79

7.4 Full and partial running times for 1x1, 4x1, and 8x1 partitions on CBD. . . 79

Chapter 1

Introduction

As congestion increases in cities, and focus on the environmental impact of transportation continues to grow, researchers have increasingly turned to technology as a way to minimize energy use across a traffic network, and mitigate the effects of an increasing number of commuters within a given system. Unfortunately, simply throwing technology at a problem does not always provide a solution. Notably, ride sharing applications, which promised to decrease congestion by taking private vehicles off the road, have actually been shown to *increase* it as users opt to hail a ride over walking, biking, or taking public transit [9]. Instead, researchers turn to the concept of intelligent transportation systems, which aim to make use of technology to provide travelers with guidance that capitalizes on knowledge about the traffic system in which they are traveling. The goal of such systems is to make controled choices on how to direct traffic, minimize congestion or environmental impact, and increase the overall efficiency and safety of the network for travelers [7].

The Intelligent Transportation Systems (ITS) Lab at MIT, in conjunction with ARPA-E, has developed the Tripod system for precisely this use, offering travelers incentives in order to make travel choices that make the overall system more efficient. The accuracy of these estimations depends on the ability to estimate real-time congestion on the network, and predict the ways in which individuals' travel decisions will affect it in the near future. To this end, Tripod relies on DynaMIT, a system developed by the ITS Lab for use in traffic simulation.

DynaMIT is a software designed to estimate the current state of a given road and rail

network, make predictions about how this state will change in the immediate future, and use these estimations and predictions to provide a control signal to influence travel decisions, e.g., guidance or incentives to travelers. DynaMIT's goals are for its control signal to be optimal with respect to its current knowledge of the network, and for its estimation of the network state to be consistent with actual conditions. The function that optimizes the control signal can take many forms, e.g., minimizing the congestion, the energy consumption or the pollution, or maximizing the revenues of the road operators. Currently intended as a backend engine to generate guidance for Tripod, DynaMIT's main objective is to minimize energy use on a given network.

Congestion on the network, or *supply*, is estimated using a combination of historical data on congestion and travel times, and real-time input from sensors on the network. Current travel plans, i.e., the origin, destination and departure time of travelers' trips, generally referred to as *demand*, are inferred from the information received from the sensors and estimated by simulating the actual movement of users on the network. Both supply simulation and demand estimation are necessary in order to appropriately estimate network state and generate control signals accordingly. DynaMIT generates state estimation, prediction, and control signals using a rolling horizon window, where an estimate of the network state over the next *estimation interval* will be computed using real and historical data. Then, given this estimated state, prediction and control will be generated over the next *prediction interval*. Default values for the estimation and prediction intervals are 5 and 15 minutes, respectively.

1.1 Main Objectives

The main objective of this thesis is to optimize the execution time of DynaMIT, with the goal of enabling it to run in real time regardless of the size of the network it seeks to control. This is crucial for its use in Tripod. In order for the control signals to be effective, the system must be able to generate them as demand is generated by real-world users. This means that estimation for the state of the network 5 minutes ahead of time must take

at most 5 minutes, so that the output is ready to be used at that time. The same holds for state prediction over a 15 minute prediction interval. Efficiency of simulation is therefore crucial to DynaMIT's efficacy as a guidance system.

At present, DynaMIT is effective in performing estimation and guidance in small traffic networks, such as Boston's Central Business District (CBD). However, the ITS Lab's goal is to be able to run Tripod, and by extension DynaMIT, on the entire Greater Boston Area (GBA). The Boston commuter belt covers a large geographic area, and contains a complex road network that poses problems for traffic simulation. In order to offer guidance to users, DynaMIT must simulate traffic movement across this entire network in real time. While it is able to run sufficiently fast to enable guidance within Boston CBD, it is unable to do so for full demand on a network the size of GBA.[1]

In particular, the requirement that DynaMIT run below a constant speed regardless of network size suggests that parallelization will be required to achieve the main goals of the Tripod project. A previous thesis from the ITS Lab, from Yang Wen [18], makes strides to optimize the core of the DynaMIT code, and also implements a parallel version of the system that lends itself to greater scalability. Unfortunately, Wen's improvements to the serial code, while significant, are still insufficient to enable DynaMIT to run on a large-scale network. Elsewhere, the integration of the parallel code into the system is very complex, and therefore has proved difficult to maintain. Moreover, while the execution of Wen's parallel implementation closely mimics the serial one, it is unable to produce completely identical results across runs. Although a near-perfect parallel system is sufficient to generate guidance, it is insufficient if one wants to perform fine-grained testing of traffic effects or assess the impact of changes to the code.

Another obstacle to running DynaMIT on larger networks is the memory use. In order to generate guidance for travelers, DynaMIT must have access to a set of most efficient routes for all origins and destinations that travelers may attempt to travel between. The size of this set of paths grows quadratically with the size of the network, and DynaMIT's

[1] The full GBA network has over 16000 nodes, which roughly correspond to intersections; and over 48000 links, which roughly correspond to a section of street between intersections. Even having the memory to be able to access paths, in order guide users between nodes, poses a problem on a network of this size.

current implementation loads all of these paths into memory at the beginning of the execution. Even though networks the size of GBA can be stored in memory, their corresponding pathsets quickly becomes too large to load.

With these goals and prior efforts in mind, we divide our work into two main stages: first, we aim to optimize the serial execution of the DynaMIT system, and develop tools to closely evaluate its performance; and second, we design and implement a new parallel version of DynaMIT that seeks to correct problems with the previous implementation. This updated parallel implementation seeks to fulfill three main goals: it must provide scalability, i.e., the running time ought to decrease with the level of parallelization; the code must be maintainable, which means that its integration into the existing coding must be simple enough that developers can change the serial code without affecting the parallel system; and finally, its execution ought to be deterministic to allow for repeatability between runs.

1.2 Thesis Contributions

The contributions of this thesis fall into three main sections.

First, we designed a custom profiling tool for the DynaMIT system, which can be integrated into the code with virtually no running time overhead, and without the need to install external profilers. This avoids the prohibitive increase in running time from tools such as `valgrind` [17], allowing it to be run alongside the code by default. This enables developers to perform profiling in parallel with development. Not only has this addition aided us in our own work, but it will continue to be beneficial throughout future iterations of the system.

Second, through addressing bottlenecks in the serial execution we were able to achieve significant decreases in running time, both in one-time startup functions and in the simulation itself. The former were prohibitively long in the original implementation, hindering development and testing, while improvements to the latter lend themselves directly to scalability.

Third, we designed and implemented a novel way to parallelize the execution of the DynaMIT system. Our design avoids the race conditions typically associated with parallel simulations *without* the use of mutex locks or other common solutions to concurrency issues. Elegant in a theoretical sense, this solution also aims to eliminate many of the overheads typically associated with parallelism. Moreover, our design is sufficiently simple that we were able to integrate it without significant changes to the existing code, indicating that its long-term maintenance will be more feasible than that of the previous implementation.

These improvements have been made in parallel to efforts to enable dynamic memory loading, which will ease the burden of pathset size. Although this work is not directly related to this thesis, and thus is not discussed here in detail, it is worth noting because it has an impact on our ability to run the system on larger networks. As these improvements are still ongoing, most of the results discussed throughout this thesis have been produced with a dummy demand on the larger GBA, or run on the smaller CBD network.

1.3 Outline

In Ch. 2 we discuss prior work, in the field in general as well as in the ITS Lab. Next, we give background on the structure and execution of the DynaMIT system in Ch. 3. We then outline the improvements to the serial code in Ch. 5, before discussing the parallel design in Ch. 6. Finally, we discuss the results we were able to achieve, and steps moving forward, in Ch. 7.

Chapter 2

Related Work

There is a wealth of previous work on scalability of traffic simulations, both in DynaMIT and in the field in general. Most published work on running time improvements focuses on parallelization, as it is essential to providing scalability as networks continue to grow.[1] All of these implementations partition the network geographically, but they differ in how they control parallel processors (either using a master/slave node setup to simplify synchronization, or a decentralized approach to eliminate bottlenecks at a single machine), and how they resolve conflicts at borders (which are largely implementation-dependent).

Domain decomposition refers to the process of segmenting the network geographically, simulating over a certain interval, and then using some strategy to merge the borders between partitions. [14] is widely cited as a model for this approach. The authors were able to achieve near linear speedup for traffic simulation in their system, using master-slave coordination and PVM for inter-processor communication [15]. However, their serial implementation differs from DynaMIT's in that they implement a *lookahead scheme*, where the execution at each time $t + 1$ depends *only* on the state of the network at the end of time t for all nodes. This allows the system to synchronize processors after each time step, effectively reproducing serial execution. DynaMIT, in contrast, has a mixed dependency, where some nodes at time t depend on the neighbors' state at previous time $t - 1$, while some depend on neighbors at current time t, complicating the synchronization process.

The lookahead functionality in [14] circumvents the need for the merge operation in the

[1] This is not to discount the importance of serial optimizations; however, these are less widely applicable than parallelization schemes, as they are highly dependent on implementation.

parallel case, largely because merge conflicts are deterministic, and are already handled in the sequential execution. A non-parallel system with lookahead 1 essentially performs updates to all nodes concurrently,[2] making the extension to parallel updates trivial. While implementing lookahead functionality is something to be considered for DynaMIT, it would require a significant change codebase, and we cannot simply adopt this parallel approach in the current system. This introduces an important goal in any maintainable codebase, which is to provide a parallel system that is *representation independent*. Ideally, a parallel approach could be adapted to any system without having to change the underlying sequential implementation.

Other attempts at domain decomposition reflect the fact that it is not easily integrated into other systems that do not already implement a lookahead. [10], [11], and [12] all fail to produce deterministic results through geographic partitions of the network. Instead, they aim to achieve statistical equivalence between runs, concluding that their results are accurate if a satisfactory percent of the network closely resembles the serial result, or if, over a large number of runs, the average output resembles the sequential results. In contrast, we seek to produce a truly deterministic output from our parallel implementation.

In the field of computer science, determinism is a necessary condition for the correctness of a parallel implementation; however, some question whether or not traffic simulations ought to be held to the same standard. Interestingly, [19] see the certain amount of randomness inherent in parallelization as an advantage. As with other implementations, they aim to make their parallel simulation statistically equivalent to their sequential one, but believe that small local discrepancies between runs due to concurrency issues mimic the natural variations in human decisions and movements. It certainly provides an interesting talking point on the value of nondeterminism. That being said, while this reasoning certainly has merit, we believe that these sorts of variations ought to be introduced intentionally rather than be allowed to persist due to nondeterminism.

[2] In most systems, this is implemented by maintaining two *views* of the network, for current time t and previous time $t-1$. All values calculated at time t are based off the previous interval's values, which remain unchanged until all calculations have completed for this step. All "previous value" variables can then be updated with the "current value" variables, and the system is ready to perform calculations for time $t+1$.

2.1 Previous work in DynaMIT

As mentioned above, there already exists a parallel version of DynaMIT using domain decomposition. A previous PhD thesis from the ITS Lab, [18], focuses both on serial optimization and on parallelization to achieve a significant speedup in execution. The DynaMIT implementation at the time included a wealth of less efficient data structures and coding practices. Improving upon these, the author was able to achieve up to a 4-factor speedup in the serial execution of the code. Most straightforward improvements from changes to data structures have already been explored in the thesis. DynaMIT's codebase still contains a large number of areas in which serial improvements on the same scale can be made, as outlined in §5.3; however, these improvements do not directly relate to Wen's work. With respect to structural changes to the serial code, the thesis serves more as a guide to those paths for improvement that have already been exhausted.

The most relevant part of Wen's thesis to our work here involves its multi-threaded extension of the DynaMIT system. The implementation is able to achieve significant speedup of the execution via parallelization. Analysis of the results concludes that it is sufficiently accurate to be able to replace serial execution in offering real-time guidance. However, the implementation does introduce some problems. First and foremost, it is unable to deterministically resolve problems created at border zones in the partition. As mentioned above, one of the goals in parallelizing DynaMIT is to achieve repeatability of simulations, and determinism is crucial to this effect. Second, its complicated partitioning and merging processes are very maintain, as updates elsewhere in the code can create unexpected problems that are difficult to track down. In order for the parallel implementation to be functional throughout development, additions to the serial code must take it into account. This often forces developers to choose between adding enhancements to the serial version of DynaMIT and maintaining the parallel version. The ITS Lab has overwhelmingly sided with the former option, and as a result the parallel implementation cannot be run on the current DynaMIT system.

To justify producing an entirely new design as opposed to bringing the previous one up to date, any new implementation ought to substantively improve upon the previous one. As such, we aim to directly address these problems of determinism and maintainability when introducing our parallel scheme.

Chapter 3

The DynaMIT System

Before discussing any changes to the code, we present the architecture of the DynaMIT system prior to our work, as documented in the DynaMIT Programmer's Guide, produced by the ITS Lab. DynaMIT has a very large codebase, written in C++ over the past two decades. The code is divided into three main sections: processes, which are the tasks the system must perform; modules, which are the building blocks with which to carry out these tasks; and components, which store requisite information about the traffic system and its travelers to be passed to the modules. There are two processes, state estimation and prediction-based guidance, which estimate the state of the network for a given interval in the future, and generate guidance for travelers based off of this estimation, respectively [2]. There are several modules and components which carry out various tasks, but we discuss in detail only two of these, the Network Topology component and the Supply module, as they are the most relevant to the work presented here.

DynaMIT's execution is largely dictated by the two processes, which serve as the entry point into the program and an interface into the modules and components. Both simulate traffic on the network; however, they carry out slightly different tasks, and the length of time (in the real-world) for which they perform these simulations depends on parameter settings that are provided for each DynaMIT run. We discuss their use and parameters in §3.1.

We next provide an overview of the Network Topology component. The *network topology* refers to the physical structure of the network (streets, intersections, lanes, etc.), which we

define more formally in §3.2. Its corresponding component is responsible for loading the topology, verifying that it is valid, and passing the requisite data to the modules and other components. The network topology is used in close conjunction with the *path topology*, which refers to possible routes that travelers can take within the given network to get from one location to another. The path topology has a large impact on the memory use of the system. Storing paths for each possible (origin, destination) pair in the network is quadratic in network size. The actual representation of the paths is not central to the work done in this thesis, and we therefore do not discuss it or its respective Path Topology component in detail; however, it does have an effect on our ability to parallelize the system, and its terminology is relevant when discussing the simulation. We therefore provide a cursory explanation of the path representation in §3.3 to give background in both of these areas.

Most of the time spent in these components is in one time start-up costs associated with loading the topologies, as opposed to during the simulation itself. As such, increasing their efficiency will not dramatically help with the scalability of longer-running simulations; however, making these components more efficient is invaluable to development and testing, where developers will have to restart the code regularly as they make small changes (see §5.2 for our contributions with respect to this problem). Their structure also dictates, to a certain extent, the structure and functionality of the Supply module.

The vast majority of the work presented in this thesis focuses on the Supply module. This module is responsible for the main traffic simulation in DynaMIT, and its execution constitutes the vast majority of the time spent in long-running simulations. Its name derives from the concept of supply and demand on a network, where demand refers to the traffic that must move (or perhaps more accurately, will attempt to move) on the network given travelers' plans, and supply refers to the capacity of the network to allow for this traffic to flow [4], [5]. As one may surmise, all simulations experience increases in execution time due to larger networks and demand; in DynaMIT's implementation, these are almost entirely due to increased time spent in Supply (see §4.1 for precise running times). Lessening the impact of network size on the running time of Supply is a major component of this thesis. We therefore discuss in detail the control flow within this module, found

in §3.4. The Supply module also contains its own representation of the network, which is constructed at the beginning of execution based on information from the Network Topology component, and whose memory allocation we detail in §3.2.2. Addressing problems both in the control flow and the representation is crucial to improving the running time of the system.

3.1 State Estimation and Prediction/Guidance

DynaMIT's execution consists of two processes, State Estimation and Prediction/Guidance. The system implements a rolling horizon model for its simulation [13], whose use in DynaMIT is introduced in [1], [3]. The model alternates between estimating a future network state and offering guidance based on this estimation.

Given a real-world view of the network from sensors and other data, the actual state of the network is computed for the current time, and this state is used to estimate the state that the network will be in at the end of the next *estimation interval*, which is by default set to five minutes. For example, at 8:00 AM, the State Estimation process will simulate 5 minutes worth of traffic, and use its end state as an estimation of what the network will look like at 8:05 AM. This process is called *state estimation*. DynaMIT uses the estimated network state until the next estimation interval, when it has access to the actual state at this time, and begins to estimate the state for the next interval.

The system then generates guidance for travelers over the *prediction horizon* following this interval, according to this estimated network state, in order to predict how to best direct travelers to minimize a given cost function.[1] This process is called *prediction and guidance*. The prediction horizon is at least as long as the estimation interval, usually longer.[2].

A step-through of the current state over several time steps is shown in Table 3.1, for an estimation interval of 5 minutes and a prediction horizon of 15 minutes. For ex-

[1]In DynaMIT's use case as a component of Tripod, this cost function will be the total energy use in the network over this interval.

[2]Ideally the prediction interval will be at least as long as the longest trip expected to occur on the network.

real time	have real network state for	using guidance generated at time	directing travelers for interval	estimating state for	generating guidance for
7:55	<7:55	7:50	7:55-8:00	8:00	8:00-8:15
8:00	<8:00	8:00	8:00 -8:05	8:05	8:05- 8:20
8:05	<8:05	8:05	8:05-8:10	8:10	8:10-8:25

Table 3.1: Sample states for rolling horizon.

ample, at 7:55AM, DynaMIT has access to the real-time state of the network up to this time, and uses this information to estimate what the state will be at 8:00AM. It then generates guidance for travelers, assuming this state is correct, for the prediction interval of 8:00AM-8:15AM. It must able to perform both the estimation and the guidance in at most 5 minutes, so that when the real-world time is 8:00AM, it will have the guidance for this time interval ready for travelers based on the predicted state. From 8:00-8:05, it directs travelers using the guidance it generated based on the *estimated* state during this interval; meanwhile, it now has access to the real-time state at 8:00AM, which it uses to estimate the next state, and have guidance ready based on this updated state at 8:05AM.

In the long term, the goal is to have DynaMIT run multiple prediction instances per simulation interval, each with a slightly different guidance strategy, to be able to choose the optimal one to minimize energy use on the network.

Both State Estimation and Prediction/Guidance[3] rely on the traffic simulation which is carried out in the Supply module. The network state is estimated by simulating the movement of vehicles and other modes of transportation through the network using a discrete time step model, and running aggregate functions to interpret the movement of the traffic at regular intervals. There are four main time intervals that dictate the Supply module's simulation and the interaction between the State Estimation and Prediction/Guidance components:

advance interval (*default=5s*) the step size in the simulation (smallest interval value given in the system)

update interval (*default=60s*) the frequency with with aggregate functions are run on

[3] We use *estimation* and *guidance* as general terms moving forward, but these are capitalized when referring specifically to the DynaMIT implementation of these processes.

the simulation, which calculate congestion values across the network, report incidents, etc, all of which may have an effect on the path choice for a given traveler

estimation interval (*default=5m*) the length of time into the future for which we estimate the state of the network

prediction horizon (*default=15m*) the length of the prediction horizon used to generate guidance strategy

3.2 Network Representation

The representation of DynaMIT's network topology is outlined in DynaMIT Programmer's Guide, Ch 9.1 and Ch 11.2.5. Both the Network Topology component and the Supply module contain their own representation of the network. These representations differ slightly in terms of how relations between different network elements are stored; however, their overall structure and the data they hold is the same.

The network topology consists of the following network elements:

(packet) DynaMIT uses a disaggregate model to simulate demand [5]; in other words, demand on the network is not represented as a smooth function, but rather as individual packets moving between origin-destination (OD) pairs on the network. More specifically, a packet refers to an individual vehicle on the network, i.e., a car, bicycle, pedestrian, train, or bus.[4] While not part of the *topology* per se, packets are an essential part of the Supply module, and are referred to as network elements below as well as in the code itself. Thus for simplicity they are defined here.

Packets on the network are represented as either moving along a segment, or queueing, i.e., waiting to move into a particular lane from the previous one. Note that (somewhat counterintuitively) queueing packets are stored at the lane level, while moving packets are stored at the segment level. In other words, packets must queue to get into a lane, but once they have gotten out of the queue they are merely represented as moving along the lane's parent segment. Packets may also be forced to wait while moving between

[4]There is some nuance here, as there may be 2 travelers attempting to travel between a given OD pair at a specific interval, i.e., there is a demand of 2, but this may only translated into a single packet if these two travelers decide to carpool, for example.

links, at which point these packets are not technically represented in any real section of the network (i.e., there is no physical manifestation of this state in reality). The queue of such packets at a particular link is referred to as a link's "virtual queue", and packets in this queue are also referred to as "virtual". Links, segments and lanes are defined below.

(node) Nodes correspond roughly to intersections on a road network. Each node contains a list of uplinks and downlinks, i.e., links that flow into and out of the node, respectively

(link) A link is a directed connection between two nodes. In other words, links correspond roughly to streets running between one intersection and another (this distinguishes a link from a "street" as a human might think of it, which runs for multiple blocks, and will be represented as multiple links in the network). Links are comprised of one or more segments.

(segment) A segment is a lengthwise portion of a link. Notably, a segment contains all lanes in its parent link along its length. Often, different segments on a link will have different levels of congestion or different traffic rules, which we want to be able to represent. This more fine-grained representation allows the system to make these distinctions while still considering shortest paths through the network at the link level. A segment contains one or more lane groups.

(lane group) Lane groups are collections of lanes that share common characteristics (e.g., in a 3-lane highway, one lane might be a carpool/high occupancy vehicle lane, or the two rightmost lanes on a large street could be turning lanes, etc.; it is useful in directing traffic to group these sorts of lanes together). Each lane in a segment should be contained in exactly one of the segment's lane groups. Lane groups should each contain at least one lane.

(lane) Lanes are the most fine-grained network element. They correspond to physical lanes on a street that run the length of a single segment. When we simulate traffic on the network, the position of packets is specified at the lane level.

(sensor) Sensors refer to real-time sensors present on the network that can be used to inform the simulation (sensors are *real world* elements; their output is not simulated).

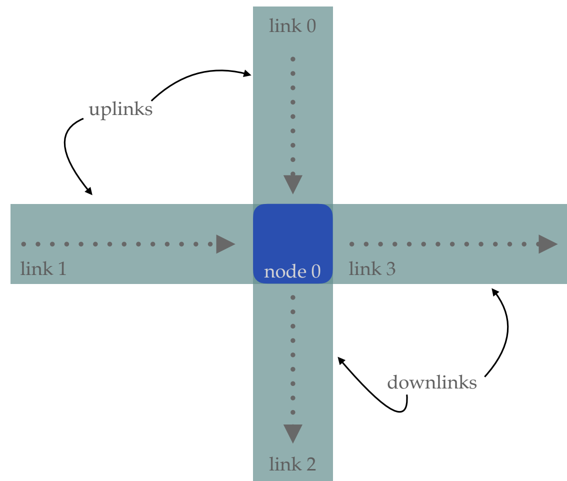
Each sensor is associated with a node.

(loader) A loader is an element that loads packets onto the network at the start of each estimation interval. Each loader is associated with a node.

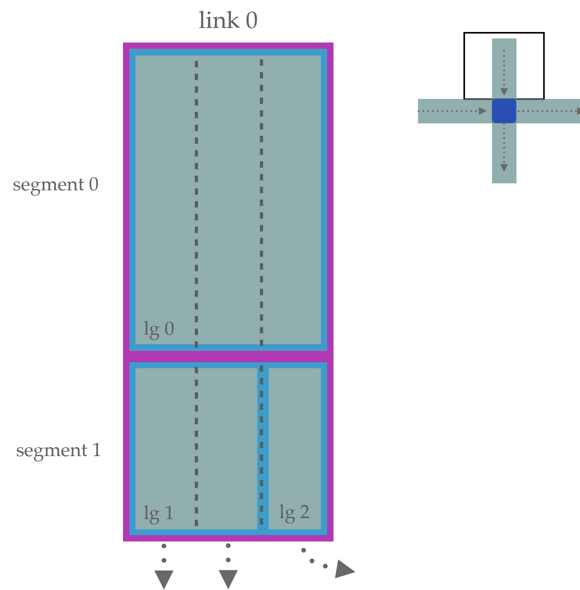
Because there is a clear structural relationship between nodes, links, segments, lane groups, and lanes, we refer to the group of these five element types as the *network hierarchy*. An example of the way in which the elements in this hierarchy are arranged is shown in Fig 3-1, which illustrates how each of the levels of specificity play different roles. Recall that packets technically move at the lane level, although represented at the segment level. The link, segment, and lane group structures dictate how the packets transfer between links. When generating the path topology, which will find the most efficient routes for travelers between requested OD pairs, paths are constructed at the link level. Because a packet cannot change links halfway through, and there are fewer links than segments, it is most efficient to represent paths as a series of links as opposed to segments.

In Fig 3-1, for example, `Link 0` is divided into two segments. In this case, all of the lanes in `Segment 0` have the same functionality, and `Segment 0` contains only a single lane group. However, the leftmost lane (note that traffic is flowing down the page in this link, so the leftmost lane is actually shown on the right) is a left-turn lane, while the right two lanes go straight. Thus in `Segment 1`, DynaMIT needs some way to model the fact that packets attempting to turn onto `Link 3` must be in this leftmost lane, as opposed to continuing straight onto `Link 2`.[5] It accomplishes this by dividing `Segment 1` into lane groups. `Lane Group 1` knows that its lanes connect to lanes in `Link 2`, while `Lane Group 2` knows that its lanes connect to lanes in `Link 3`. Thus if a packet is attempting to follow a route that travels along `Link 3`, it will be aware of the fact that it must be in one of the lanes in `LaneGroup 2`. In this case, this lane group only contains a single lane; however, if it contained more than one, packets select the best lane based on availability.

[5]If packets were unaware of this, it would mean that packets could be seen to jump across lanes suddenly while making turns. At best, this results in unrealistic behavior that goes unnoticed. At worst, this could actually have an effect on the viability of the simulation. If the turning lane is particularly congested, but packets are allowed to spread out across all three lanes and then hop onto the next link, we lose valuable information about congestion at this turn.



(a) Example of a single node and 4-link network. Node 0 has uplinks 0, 1 and downlinks 2, 3. Direction of traffic is shown.



(b) View of Link 0, with three lanes, denoted by dashed lines. Lane groups (denoted by "lg") ringed in blue, segments in purple. Direction of traffic/turning lanes shown.

Figure 3-1: Example of hierarchy of network topology structure.

3.2.1 Initializing Network Topology

The network topology is read in from a text file, which specifies the characteristics of the network elements and their relations, with the exception of packets, which are generated separately at the start of each estimation interval based on travel demand. It maps database indices in the input to DynaMIT-internal IDs, which are contiguous starting from 0 to increase efficiency.[6] Nodes' uplinks and downlinks are not given explicitly in the input file. Instead, the link definitions include start and end nodes, and the corresponding node is looked up and its values are updated when the link is initialized. Segments, lane groups, and lanes are all given in the input file as part of the link object. The Network Topology component also verifies that the network file is consistent, i.e., that each network component provided is unique and that all of the links' start and end nodes exist.

3.2.2 Initializing Supply

The network topology is loaded into the Supply module from the Network Topology component. The original implementation of the module included only top-down dependencies. In other words, nodes contained pointers to their up and down links, but links did not know their start and end nodes; links contained a list of pointers to their corresponding segments, segments to lane groups, and lane groups to lanes; but the lane groups only had access to the ID of their parent segment, and lanes only had access to the ID of their parent lane group. While this implementation is simpler in terms of class dependencies (i.e., alleviates the need for mutual imports, an ever-frustrating albeit mild inconvenience when developing in C++) and the initialization of the Supply module, its asymmetry eliminates type safety and forces interactions between network elements and their parents to be passed through an instance of the Supply module, which contains mappings between the IDs of network elements and their actual objects.

Current Memory Allocation

Another important consequence of the previous implementation is the order in which various element are arranged. Particularly in a lower-level language such as C++, this ordering, and its relation to the order in which the program iterates over the corresponding memory,

[6] This enables DynaMIT's data structures to use arrays or `C++` vectors as opposed to hash maps to locate network elements based on ID.

can have a large impact on performance [6]. The previous iteration of the Supply module, given the way that it establishes parent/child pointers, builds the network hierarchy from the bottom up, first creating all of the lanes, then the lane groups, then segments, then lanes, and finally nodes. Loaders and sensors, which exist outside of this hierarchy, are added afterwards. A view of how these elements are physically laid out on the heap is shown in Figure 3-2. Each element is stored in the heap, and a pointer to an array of pointers to these elements is stored in the Supply instance. There are no conditions on the ordering of the links with respect to the order of their parent nodes.

3.3 Path Representation

The pathset generation and representation are discussed in Chapter 10.7 in the DynaMIT Programmer's Guide. The pathset is generated using Dijkstra's shortest path algorithm; random perturbations in weight (and sometimes deletions) are made to various links in each shortest path to generate multiple "best" paths in the network. These various path choices are meant to give flexibility to travelers should the absolute shortest path in theory prove slower in practice due to traffic conditions. During simulation, the weight of links along these paths is adjusted dynamically to reflect current conditions on the network, and as a result travelers may be redirected along a more efficient route dynamically.

The pathset is stored in a decentralized manner, which follows directly from the output of Dijkstra's algorithm. Instead of representing paths as global elements, each link is aware, given a destination nodes, of an array of paths to this node. Each of of these paths is simply represented as a (**link**, **path**) pair, where the link is the next link in the path, and the path is the index that corresponds to this path at that link. Thus to reconstruct a global "path" object containing each of the links in a given path, one must traverse the entire path from beginning to end to find all of the links.

While this representation may seem unintuitive, it has several advantages: first, that its representation is a direct result of Dijkstra's algorithm, and is thus easy to generate; second, it reflects how actual travelers are given guidance on the network–a packet does not need to know its full path to advance, only the link it should move to next and what

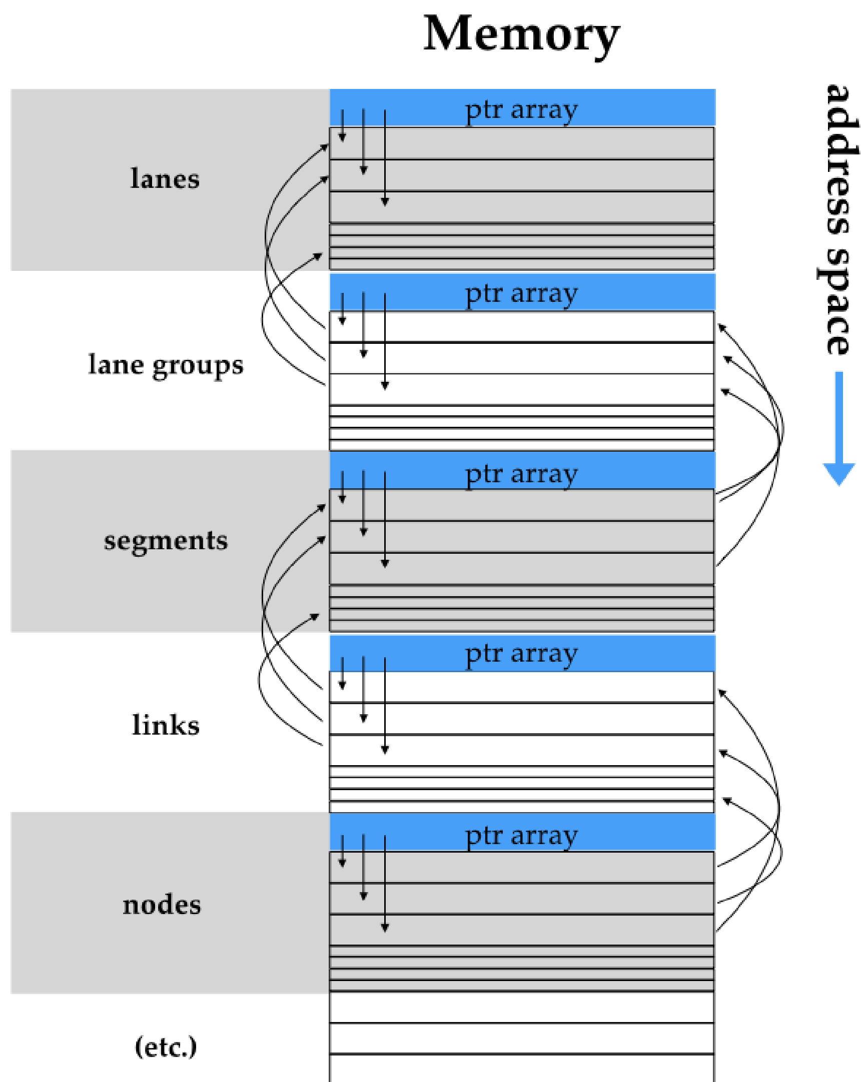


Figure 3-2: Memory ordering in current DynaMIT implementation

it should do when it gets there[7]; and third, it lends itself easily to parallelization, as there is no need to worry about breaking the pathset across various processors or machines. We can simply partition the pathset representation in the same way we partition the network representation.

3.4 Simulation Control Flow

DynaMIT uses a discrete time step model in its simulation. At each advance interval, it iterates over all the nodes in the network. For each node, it sorts all of its upstream packets in increasing order based on their distance to the node, and advances each packet on the network for the distance it would expect to travel given congestion, speed limit, and the size of the time step. After it has advanced all of the packets present on the segments, it advances those in the virtual queue. Note that advancing packets is recursive, i.e., once a packet has been moved out of a virtual queue onto a lane, it may be advanced further within the same advance interval.

Pseudocode for the advance traffic functionality is given in Fig 3-3. Importantly, the `advanceAllPackets`, `advanceAllVirtual`, and `loadNewPackets` functions, which are the main functions which exhibit an increase in running with largest network sizes and demand, all consist only of a simple loop over the nodes (or loaders, which are each associated with a node). This suggests that these functions are all good candidates for parallelization given a geographic partitioning of the network.

[7] This of course does not fully reflect a users experience, as a human traveler would likely want to know their full route at all times; however, within the simulation this is not necessary. The full route can easily be reconstructed to pass to a traveler when they are assigned a path.

```
1: function advanceTraffic
2:     advanceAllPackets()
3:     advanceAllVirtual()
4:     loadNewPackets()
5: function advanceAllPackets()
6:     for node n in network do
7:         packets = getSortedUpstreamPackets(n)
8:         for packet p in packets do
9:             advancePacket(p)
10: function getSortedUpstreamPackets(node n)
11:     pkts = list of packets
12:     for link l in n.uplinks do
13:         for segment s in l do
14:             pkts.add(s.movingPackets)
15:             for laneGroup lg in s do
16:                 for lane ln in lg do
17:                     pkts.add(ln.queuingPackets())
18:     pkts.sort(key=dist_from(n))
19: function advancePacket(packet p)
20:     (some recursive moving in the network...)
21: function advanceAllVirtual
22:     for node n in network do
23:         for link l in n.uplinks do
24:             for packet p in l.virtualQueue do
25:                 p.moveInQueue
26:                 if ! p.queuing then              ▷ p has been moved out of virtual queue
27:                     advancePacket(p)
28: function loadNewPackets
29:     for loader ld in network do
30:         for packet p at ld do
31:             add p at ld.node
```

Figure 3-3: Pseudocode for the `advanceTraffic` function.

Chapter 4

Necessary Properties for DynaMIT

Efficiency of a program can be calculated along two axes: running time and memory use. As these are often inversely proportional, an good implementation generally involves a tradeoff between the two. Both time and memory use are prohibitively large in the current implementation of DynaMIT. Generating the pathsets is time-intensive, and must be done in advance of the simulation; the size of this pathset for the Greater Boston network far exceeds memory limits for DynaMIT's execution. Due to its intended use as a guidance system, it is absolutely necessary for DynaMIT to run on larger networks in less than real time. While this has already been achieved on smaller networks, it currently is not the case for GBA, even with a very limited demand. This problem will only grow as we attempt to simulate a larger number of packets on the network. Our main focus in this thesis is to achieve scalability in DynaMIT's running time, with consideration for parallel efforts to address the problem of memory use.

Under the umbrella of efficiency lie three subgoals, which we discuss here in detail: first, that the serial optimizations of the code be exhausted before entertaining the idea of a parallel implementation; second, that any parallel implementation be a significant improvement upon the previous one, with the main objective of achieving deterministic (i.e., repeatable) execution; and third, that these parallel improvements are sufficiently transparent such that their maintenance will not hinder improvements to the serial code, allowing for continued use of the parallel implementation in the future.

The former two requirements follow directly from our main goal of real time execution.

The latter is important in guiding our design, as parallel implementations and their effects are often obscure. Our improvements will likely fall by the wayside if an inordinate amount of effort is required to ensure they are consistent with the serial code. A lack of maintainability was one of the larger difficulties that arose in the previous parallelization scheme, and we specifically aim to forestall this problem in our implementation.

Efficient running time is important given DynaMIT's intended use case; however, without dynamic memory loading to handle larger pathset sizes, DynaMIT cannot be run on larger networks at all, let alone in real time. Thus our efforts here have been made in parallel to those to improve memory management, which is a continuing project within the ITS Lab at this time.

4.1 Running Time Efficiency

DynaMIT's intended use as a real-world control system makes it imperative that its simulation is able to run in real time. Because of the rolling horizon model, discussed in §3.1, "real time" does not merely mean that five minutes on the network must run in five minutes. Within this time we must be able to simulate traffic for a full estimation interval during state estimation; simulate traffic traffic for a full prediction horizon for prediction/guidance, which is at least as long as the estimation interval but often significantly longer; and allow time for any additional overheads associated with interactions between the two processes and the various components and modules.[1]

DynaMIT spends most of its execution time in the State Estimation and Prediction/Guidance processes, although the latter takes a greater amount of time because it simulates traffic over a longer interval. The bulk of the running time in both processes comes from advancing the traffic in the system. Recall from Fig. 3-3 that the program must iterate over the entire network hierarchy in order to determine an ordering for the packets, and each of the packets is advanced, if possible. This means that the running time for advancing traffic is linear in both the size of the network and the number of packets.

[1] In the future, the goal is actually to run *multiple* instances of prediction/guidance per interval, each with its own parameters settings. Because these simulations will only share initial values, but can otherwise be completely independent of each other, we can naively launch them in parallel without worrying about race conditions. However, it is worth noting that setting up these parallel threads and merging their results will require additional overhead, and thus should not be considered entirely "free".

It is worth noting that network size informs the number of packets.[2] On a network with any amount of congestion, one would expect the number packets to dominate the number of number of links and nodes; moreover, changes to the representation discussed below make network size less relevant than packet count. Thus when we discuss the problem of scalability on larger networks, we largely refer to the fact that the number of packets on the network will be higher if the network encompasses a larger area; the number of nodes and links, therefore, may increase running time indirectly rather than directly.

Sample running times for the full simulation, the two processes, and their respective advances of traffic are shown in Table 4.1. These running times are taken from the execution

Function	Total Calls	Total Time	Time Per Simulation Interval	Time per Call	Variance
State Estimation	6	59.9887s	9.99812s	9.99812s	0.654413
Prediction Guidance	6	485.654s	80.9424s	80.9424s	363.863
SE: Advance Traffic	720	43.683s	7.37488s	0.0606709s	0.00471351
PG: Advance Traffic	2160	396.532s	66.3029s	0.183580s	0.005037
Main Sim Loop	6	**546.054s**	90.7094s	90.7094s	290.322

Table 4.1: Sample running times for main DynaMIT functions over 6 simulation intervals in Boston CBD.

of DynaMIT in Boston CBD, with an estimation interval of 5 minutes and a prediction horizon of 15 minutes. This means that the main simulation loop must run in at most 300s per simulation interval to be within real time. While the current running time of approximately 90s per loop is sufficient for a network of this size, we would expect this running time to scale roughly linearly with network size. Boston CBD has 834 nodes and 1802 links.[3] The GBA network has 18016 nodes and 46763 links, making the complete

[2] When expanding a network representation from a downtown area such as Boston CBD to the area and its commuted belt, such as GBA, we do not remove packets from the original area because we are now able to represent outlying areas. However, packets do not scale entirely linearly in the number of links, because we would except outlying areas to have less congestion. We therefore carry out most testing on the smaller Boston CBD for consistency.

[3] These number may vary slightly in newer versions of the network, but the approximate measurement remains the same.

network 24.575 times the size. Even if we assume that congestion is significantly less in farther flung areas, we would not expect DynaMIT to be anywhere near real time when run on a network the size of GBA.

Testing on a limited pathset[4] for GBA confirms these expectations. A plot of the running time in GBA for various demands is shown in 4-1. While running with a small demand

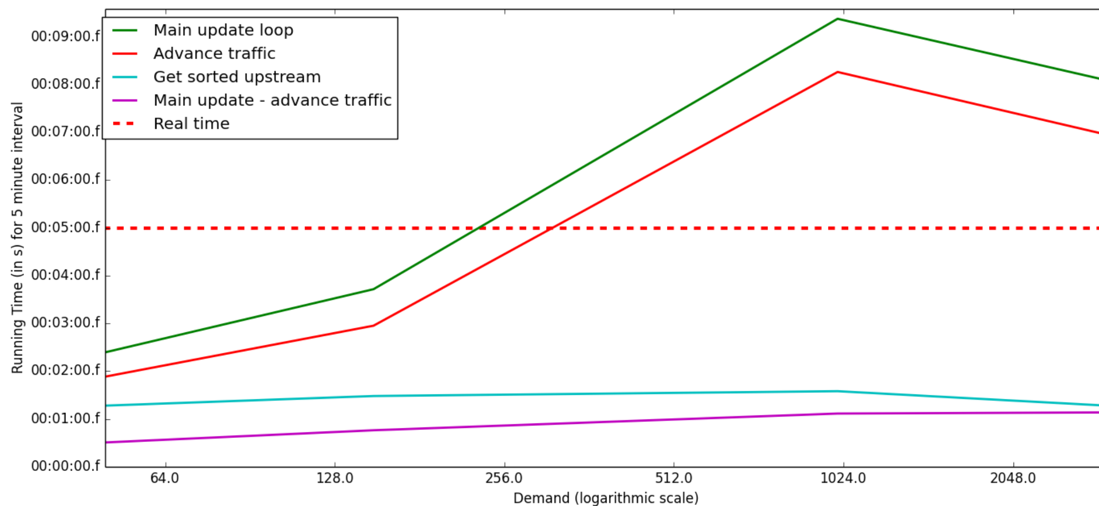


Figure 4-1: Demand, scaled logarithmically, vs. running time in GBA.

between a single origin and destination runs within real time, we see a sharp spike in the running time for higher demands. A mere 200 or so packets causes the simulation to jump above real time; the Greater Boston Area can see as many as 3 million trips taken in a day. It is encouraging to note, however, that the running time of the `advanceTraffic` function closely tracks that of the overall running time. In other words, a parallel implementation that achieves scalability for this function will go a long way towards achieving scalability for the overall simulation.

4.2 Repeatability of Parallel Traffic Simulations

Intuitively, one can parallelize the execution of a traffic simulation easily by dividing the network geographically, a process known as domain decomposition [14]. This is advantageous in that the partitioning step is transparent to later developers (although merging

[4] As previously mentioned, loading the GBA pathset into memory requires additional changes to the code that are being developed in parallel to this work in the ITS Lab. These adaptations are outside the scope of this thesis. We instead provide data for full demand in Boston CBD and a dummy demand in GBA, and extrapolate, with the expectation of testing on GBA with full demand when these improvements are made.

the output of the partitions is certainly not) and can be relatively inexpensive, computationally.[5] Because paths in DynaMIT are stored implicitly at each node, allocating links to various processors will, by extension, divide the pathset amongst these processors. This sort of parallelization therefore has the potential to decrease both execution time and per-processor memory usage by a factor linear in the number of processors. Partitioning the network such that all sections are small enough to run in real time thus effectively solves the scalability issue in DynaMIT.

Unfortunately, traffic systems are quite complex, and enforcing repeatability for these simulations can be very difficult, as discussed in §2. While a single section of a parallelized network will internally be identical to the serial version of DynaMIT on that subset of the network, race conditions abound[6] at the borders between sections. At these boundaries, each processor must be aware not only of the traffic state in the links of its own domain, but also of the state of the links directly across its borders, which belong to the neighboring domain under some other processor. Failure to correctly synchronize this information will at best lead to incorrect output, and at worst could crash the program due to concurrency issues.

In most applications, a parallel implementation is considered correct only if its output is indistinguishable from the serial one. In traffic simulators, these conditions are relaxed somewhat. Instead, researchers ([18], [12]) accept a parallel implementation as "good enough" if its results are within a certain threshold of the serial execution. While these parallel implementations may be sufficiently accurate to reliably generate control signals, they are not guaranteed to be repeatable. Unrepeatability hinders researchers who want to able to see the effects of tweaking parameters or to repeat a certain phenomenon, perhaps with more detailed output.[7] While overall traffic flow will likely average out to be consistent with the serial execution, smaller effects on the network may be easily confused with the side effects of parallelization.

[5] This only holds if the network partitioning is static, e.g., evenly distributing the number of nodes over the partitions. Some systems, DynaMIT included, will instead dynamically adjust partitions based on the demand on the network. This can improve simulation time, but computing the partitions themselves can be computationally expensive depending on the algorithm.

[6] Pun unintended, but apt.

[7] Print statements notoriously affect race conditions, as they take significantly longer than other operations, and can actually eliminate the data race that caused the original problem.

4.3 Maintainability

The third requirement for our parallel design is that any changes made to enable parallelization must be maintainable. We require two conditions to hold to consider this to be the case: first, that any parallel design and implementation be sufficiently simple that new developers are able to understand and update the corresponding code; and second, that the parallel implementation be sufficiently independent of the underlying representation and control flow of DynaMIT that changes to the serial code do not render the parallel code unusable.

Neither of these goals is realized in the previous implementation of DynaMIT. While it is effective in achieving scalability, the way in which the parallel implementation interacts with and depends on the serial code is very complex. Because of this, it has proved difficult to maintain, and at present cannot be used with the latest version of DynaMIT. It is certainly possible to simply update the old parallelization to work with the current codebase; however, it is sufficiently complicated that it would take a significant amount of time, and it is likely that this problem would reoccur as DynaMIT continues to evolve. It is simply not maintainable as a solution to the scalability problem if updates to the system render the parallel code unusable for long periods of time between patches.

Chapter 5

Serial-execution Optimizations

Before embarking on a parallel implementation, we aimed to exhaust the optimizations for the serial code. First, in order to assess any improvements we were able to achieve, we implemented a custom inline profiling tool that can track key functions and bottlenecks, as laid out in §5.1. We then restructured or changed various data structures to improve performance when loading values into memory, which we discuss in §5.2; and finally, we rearranged how network elements are stored in memory in the Supply module, which is outlined in §5.3.

5.1 Profiling

Before starting on any improvements to the code, parallelized or otherwise, we wanted to be able to closely track its execution time in order to assess the benefits of any changes. In order to see improvements not only to individual functions, but to the hierarchy of different function calls, it is important that we are able to capture the call graph for the code, as opposed to merely performing flat profiling.

Unfortunately, pre-packaged profilers such as `valgrind` [17], while certainly useful in producing these sorts of results, can significantly increase execution time, sometimes by a factor of 50 or more. In long-running simulations such as DynaMIT, this hinders development that closely relies on profiling for the smallest of changes. Other tools such as `perf` [16] could not be run on the ITS Lab servers as currently configured. In an attempt to make profiling both easy and portable, we instead chose to implement an inline profiler,

enabled via compiler flags, that can track the running time for specific functions within DynaMIT. This allows profiling to be run by default alongside development, but still be switched off during production.

Wen's thesis [18] also discusses the addition of an inline profiler. However, this profiler currently only performs flat profiling. This means that it can track the running time of a single function from start to finish, but cannot identify the fact that one function is called from within another, i.e., makes up part of its parent's running time, and cannot perform flat profiling in a parent and child function call simultaneously. While this profiler is still useful for large-scale evaluations of different guidance strategies, it is insufficient for the level of granularity that we wish to achieve here.

5.1.1 Updated profiler design

The updated profiler is able to handle both flat and hierarchical profiling. The main functionality is within the `Profiler` class, all of which is placed within compiler guards to easily disable profiling in production. The Profiler class maintains a stack of `profilerFunc` elements, which represent the current call graph. The `profilerFunc` struct contains metadata about the function (the location of the function call, its name, etc.), the system time when it was last called, and its most recent `startTimer` time, which is a simple timer maintained for each function in the stack, used for flat profiling. When a function is called, its corresponding `profilerFunc` is pushed to the function stack; when it is exited, the time spent in the function is calculated and pushed to vector of call times corresponding to this function, and it is then popped from the function stack. This allows us to calculate the total time, average time, and variance of each function.

The `Profiler` class also keeps track of the parent function of each call (i.e., the function currently at the top of the stack when a new function is pushed), allowing us to differentiate between calls to the same function made at different locations. This is important in maintaining the hierarchy for the call graph. Sample call graph output is shown in Fig 5-1. Note that some function names are equivalent to the function call made within the C++ code, while some are user-specified. We specify macros within the profiler that enable both options, giving the user more flexibility when adding profiling to the code.

```
[1]     idlAssignmentMatrixList::the()->load()
=====================================================
DynaMIT/DynaMIT.cc:348 TIME TO LOAD: 58.0784s
=====================================================

[2]     main_sim_loop_0
[3]         idlLinkToll::the()->load(
                idlParameters::the()->getLinkTollFile())
[4]         idlEstimation::the()->execute(timeInterval)
[5]             executeODEstimation(timeInterval_)
=====================================================
processes/Estimation/dtaEstimation.cc 1.59551s
=====================================================

[6]                 idlSupply::the()->simulateTraffic(timeInterval_.start
                        simInt, upInt, advInt, eps, flag_report)
[8]                     main_update_loop
[9]                         advanceTraffic()
[10]                            reset_processed
[11]                            advanceAllPackets()
[12]                                startingForLoop
[14]                                    sorting_upstream_packets
[16]                                    iterating over packets
[17]                                    iterating over links
[18]                            advanceAllVirtual()
[19]                            loadNewPackets()
[20]                        reportTraffic(flag_report)
```

Figure 5-1: Sample call graph for inline profiling.

Outputs framed by = lines correspond to flat profiling. The flat profiling includes the location in the code where the timer was started, and an optional user-specified message; it does not affect any profiling maintained by the call graph.

The function stack also conveniently allows us run to flat profiling on multiple different functions in the hierarchy at once. In the previous implementation, there is only a single timer, which meant that if the flat profiling was initiated in a child function, it invalidates the profiling started in its parent function. Since each function in the stack maintains its own `startTimer` variable, each time the flat profiler is called, we simply start the timer for the function at the top of the call stack, without affected the flat profiling of any of its parent functions.

Running times for the call graph shown in Fig 5-1 are given in Fig 5-2.

5.2 Optimizing Network Initialization

Previous efforts to improve DynaMIT's running time did little to address the time the system spends reading in the initializing the network topology. This is likely due to the fact that this only occurs once at the beginning of the execution, and thus does not affect the running time of the actual traffic simulation.[1] This functionality therefore becomes unimportant in long-running tests. During development, however, it is often necessary to see the effects on running time from changes in the code as opposed to the output from a guidance strategy. This means that DynaMIT must be run for only a single round of state estimation and prediction/guidance, making long startup times cumbersome. Average loading times for various DynaMIT networks are given in Table 5.1.

Network	Nodes	Links	Loading Time
Boston CBD	843	1877	49.609s
Greater Boston	18016	46763	1278.64s

Table 5.1: Initialization times for DynaMIT networks.

Waiting for nearly half an hour to initialize the network topology each time small changes

[1]Some of functions we improved are called during the main simulation, but not frequently enough that we saw these changes have a sizable impact on simulation running time.

```
TOTAL SIMULATION TIME: 26.3021
TOTAL EXECUTION TIME: 76.6198


Func    Call loc   Num Calls       Total Time      Avg Time    Variance
-----------------------------------------------------------------------
 [1]    DynaMIT/DynaMIT.cc:225    1    2e-06
 [2]    DynaMIT/DynaMIT.cc:367    1    23.7821
 [3]    DynaMIT/DynaMIT.cc:390    1    4.5e-05
 [4]    DynaMIT/DynaMIT.cc:449    1    4.88701
 [5]    processes/Estimation/dtaEstimation.cc:193    1    4.88699
 [6]    processes/Estimation/dtaEstimation.cc:1449    2    0.120867
 [7]    modules/Supply/dtaSupply.cc:472    4588    0.002251
 [8]    modules/Supply/dtaSupply.cc:613    10    0.099964
 [9]    modules/Supply/dtaSupply.cc:655    120    0.045669
 [10]   modules/Supply/dtaSupply.cc:2056    120    0.017015
 [11]   modules/Supply/dtaSupply.cc:2083    120    9.4e-05
 [12]   modules/Supply/dtaSupply.cc:2089    120    8.2e-05
 [13]   modules/Supply/dtaSupply.cc:2095    120    0.000162
 [14]   modules/Supply/dtaSupply.cc:682    10    0.02923
 [15]   DynaMIT/DynaMIT.cc:503    1    18.7538
 [16]   processes/PredictionGuidance/dtaPredictionGuidance.cc:140    1
                 6.12141
 [17]   processes/PredictionGuidance/dtaPredictionGuidance.cc:244    1
                 0.118276
 [18]   modules/Supply/dtaSupply.cc:472    9071    0.004189
 [19]   modules/Supply/dtaSupply.cc:613    15    0.093001
 [20]   modules/Supply/dtaSupply.cc:655    180    0.042297
```

Figure 5-2: Sample running times for inline profiler.

are made to the code is simply infeasible for development. Most of the running time comes from the fact that each of the references between the various network elements must be resolved in order to verify that the network file is valid, and to build the internal network dependencies required for simulation. While network elements in the input file all have unique IDs that can be used as references, these IDs are not contiguous, and can be very large. DynaMIT instead assign each element its own internal IDs, which contiguous and 0-indexed, and can thus be used as indices into the vectors in which each of the network elements are stored. This is more efficient during simulation, where iterating over elements in a vector is faster than find values in a hash map[2]. Therefore, when passed to DynaMIT, each network element has the index of its parent element specified via input ID, or `userID`,

[2]Asymptotically, the running time of these two operations is the same. However, in optimizing the simulation, we want to get every advantage we can; accessing vectors in order is marginally more efficient.

while internally they must be accessed via the DynaMIT ID, or `id`.

In the current implementation, the code iterates over all elements to resolve these references and verify that the network file is valid. Instead, we add lookup tables to resolve these references in constant time. This has previously been attempted in DynaMIT, but not to the same extent. This previous implementation was not deemed effective enough to maintain in the code; however, we believe our results were able to significantly contribute to development.

5.3 Memory Locality Management

As shown in Fig. 3-2, the previous DynaMIT system allocated a contiguous chunk of memory for each type of network element. While this makes it easy to instantiate each group of network elements, it does not reflect the order in which these elements are accessed during the simulation, resulting in bad cache locality. This does not have an effect on asymptotic running time, but can have a significant impact on the real-time performance [6].

In our updated implementation, we allocate memory for the network elements in the order in which they are accessed when traversing the network hierarchy. This updated structure is shown in Fig. 5-3. As we can see in the figure, a list of lanes is stored beneath their parent lane group, a list of lane groups (including their child lanes) stored beneath their parent segment, etc.. Recall that, during the advance traffic step, a list of packets upstream of each node is compiled. In order to do this, DynaMIT must check up the node's uplinks, and their corresponding child elements. The nodes are iterated over in the order in which they are specified in the input. Thus to advance traffic on the full network, DynaMIT merely has to traverse elements in order on the heap. This is significantly more efficient in terms of cache use. Particularly on larger networks, where the storage of the higher level cache is likely to be insufficient, this can have a significant impact on running time.

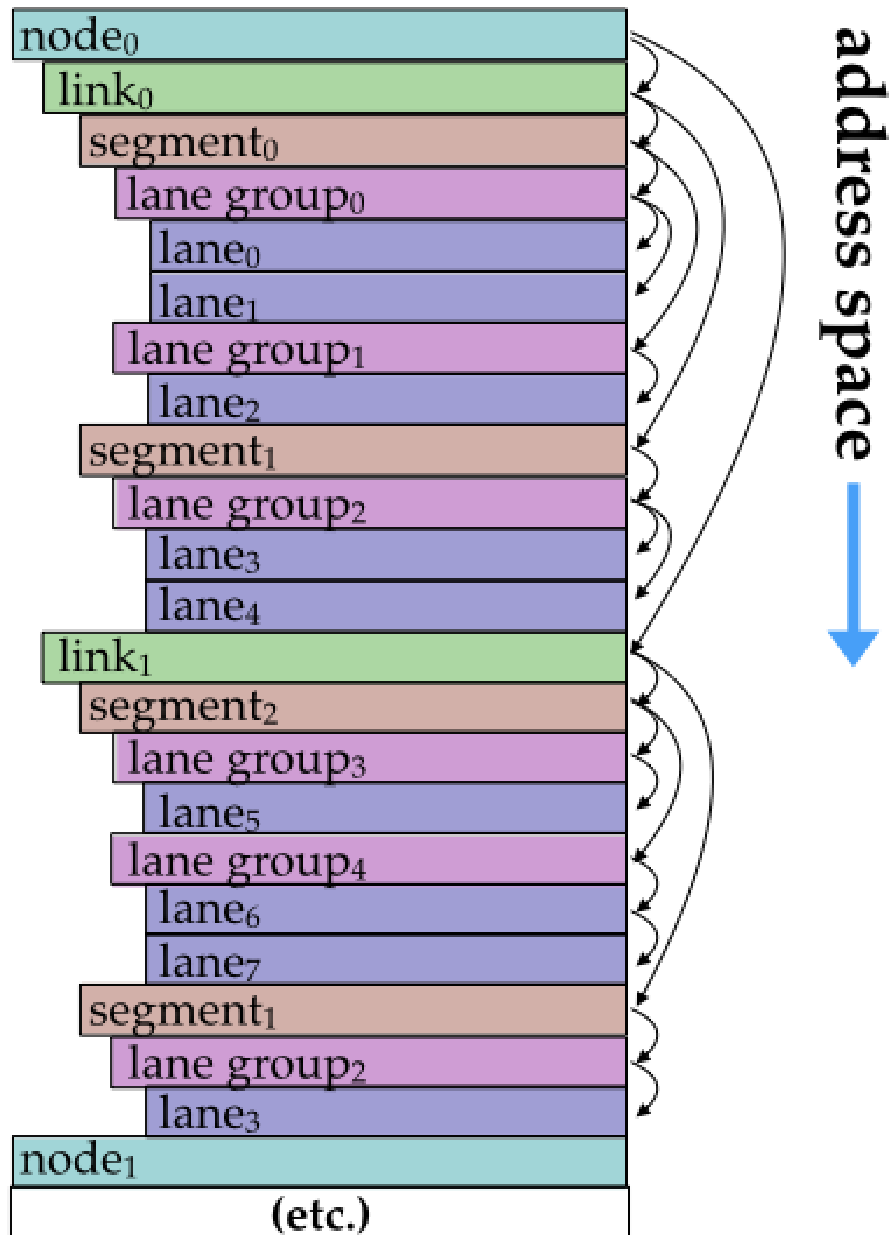


Figure 5-3: Updated order of network elements in memory.

Chapter 6

Parallelization

While serial optimizations to the code are important, to truly enable DynaMIT to run at a near-constant speed regardless of network size, one would expect that some level of parallelism would be necessary. The previous work in DynaMIT succeeds in producing this sort of scalability, but it fails to achieve determinism or transparency. We instead propose an alternate parallel implementation, based off of geographic partitioning, that eliminates the race conditions generally associated with this method, and greatly simplifies the process of partitioning and merging between processors. The main premise behind this idea is that staggering execution of neighboring partitions will avoid the problems that arise from their interaction. We argue from a theoretical perspective (and later demonstrate from an empirical one) that this parallelization method fully avoids race conditions, and is therefore repeatable without the use of mutex locks; and that maintain that its implementation is more transparent than previous attempts at geographic partitioning within DynaMIT.

The largest problems in parallel traffic simulation stem from the fact that dependencies within a network, and by extension between processors in a geographic partition, can be quite complex. This is true in DynaMIT in particular, as some nodes depend on the execution of their neighbors at the previous time step, while some depend on their neighbors' state at the current one. These dependencies are created by the order in which traffic is advanced across the system. Recall from §3.4 that packets are advanced based on their down node at the beginning of each interval. DynaMIT advances nodes in a constant order based on their ID; at each node, packets at the node's uplinks are advanced in ascending order based on their distance to the node. Thus if nodes n_i and n_j are adjacent, and

packets at n_i are advanced before those at n_j, then the behavior of packets at n_j at time t will depend on the congestion at n_i at time t; however, when advancing packets at n_i at time t, those at n_j will not have been advanced yet. This means that the behavior at n_i at time t is actually dependent on congestion at n_j at the *previous* timestep $t-1$. We refer to the fact that the behavior at n_i informs the behavior at n_j during the same time step as a forward dependency, denoted at $n_i < n_j$. If the execution is repeatable, either n_j will *always* advance its packets after n_i does, or the construction of the network is such that its end state is identical regardless of which node advances first. Any correct parallelization scheme must be able to ensure that one of these properties holds across processors, to ensure that the state at the end of a particular interval is identical for each run.

Say we begin with a simple 1D partition, where nodes are assigned to partitions, or *bands*, based on their x position. We can then impose an ordering of the nodes that is consistent with our geographic partition, i.e., given two processors, p_m and p_n, without loss of generality we order the nodes such that all nodes in p_m will update before any nodes in p_n, while maintaining their original ordering within each band. We can thus say that p_m is advanced before p_n, or $p_m < p_n$. While moving dependencies to the band level in this way simplifies the problem posed, it does not actually eliminate race conditions at boundaries when executing all processors in parallel.

To address this, we take advantage of the idea that traffic effects do not propagate instantaneously. In other words, the movement of a car driving in Harvard Square will not affect the movement of a driver in Kendall Square within the same second.[1] In the real world, these effects are constrained by speed limit and other physical restrictions on the network. In a simulation, they are similarly dictated by the maximum distance the system attempts to move a packet within a time step. This separation implies that it ought to be possible to eliminate some dependencies in the network such that certain bands can be executed in parallel without race conditions.

[1] In today's world, a serious accident may, in fact, change drivers' decisions this quickly thanks to systems such as Google Maps, Waze, etc., which distribute information about network state almost instantaneously to remote locations. However, when we talk about parallelizing traffic movements, we are discussing simulating all traffic *within a given time step.* Thus, in a parallel simulation, if this sort of information is included, it should not be pushed to nodes until the end of a time step, both in the serial and parallel execution.

Depending on the ordering of the nodes within the network, we can derive two types of independence properties from this finite velocity assumption. The first we refer to as *update independence*. We say a band b_i is update independent from another band b_j if no packets that originate in b_i will reach b_j within the same time step. The second property we refer to a *full independence*, or just *independence*. Two bands are independent if, for any pair of packets p_i, p_j, where p_i originates in b_i and p_j originates in b_j, the state of the network at the end of the advance interval will be the same regardless of which packet is advanced first.

While it may seem that full independence is a strictly stronger property than update independence, their relative strength and implications are highly dependent on node ordering, and thus may be deemed as more or less effective depending on the requirements for the ordering of the nodes. As such, we discuss both properties and provide the parallel algorithms that they enable, although we only use the latter property in our final implementation.

We first reiterate our goals for parallelization in §6.1 before discussing our design in more detail in §6.2. In §6.3 we formally define the propagation of traffic, and go on to formally define update independence (§6.3.1) and full independence (§6.3.2) and what they imply about concurrency. We then discuss two versions of the parallel algorithm in §6.4, valid under update and full independence, respectively. Finally, we provide our implementation in DynaMIT, which relies on full independence, including a preliminary partitioning algorithm and updated network representation, in §6.5.

6.1 Goals

As stated above, we have three main goals with respect to parallelization:

1. **scalability** is clearly the first goal in any parallel implementation. In order to run DynaMIT on larger networks, its running time ought to be inversely proportional to the number of processors

2. **simplicity** is important in ensuring the longevity of any parallel implementation. While a more complex solution will potentially produce the required results, if it is sufficiently difficult to maintain such that it falls out of sync with the rest of the

code, in the end it still fails to produce a long term scalable solution

3. **determinism** is generally considered a necessary property of a correctly implemented parallel system. As discussed in §2, this constraint is often relaxed in traffic simulations, which are notoriously complicated and rife with race conditions and interdependencies. However, we still aim to produce a deterministic output, which will allow for both small scale and large scale analysis of the results of the system. While a nondeterministic, statistically equivalent system will suffice to provide guidance to travelers, it will not enable researchers to closely examine traffic patterns, and will hinder developers who want to be sure of whether changes in the output are due to changes in the input or code as opposed to nondeterminism.

6.2 Design

We first outline our design before proving its validity and discussing the specifics of the implementation. As mentioned above, we propose two similar parallelization schemes: the first, enabled via update independence, we refer to as *node pipelining*; and the second, which relies on full independence, we model as a vertex coloring problem.

6.2.1 Node Pipelining

The first iteration of our parallel design draws on the idea of pipelining. In a system with a non-cyclical dependence structure, as shown in Fig. 6-1, we can start the execution of

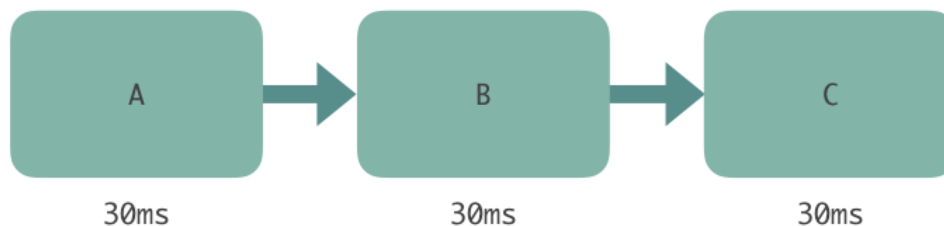


Figure 6-1: Example of a good candidate for pipelining.

any given module for a time step t as soon as the previous module has completed for that time step, and the following module has completed for time step $t-1$. An example of the pipelined execution for such a system is shown in Fig. 6-2.

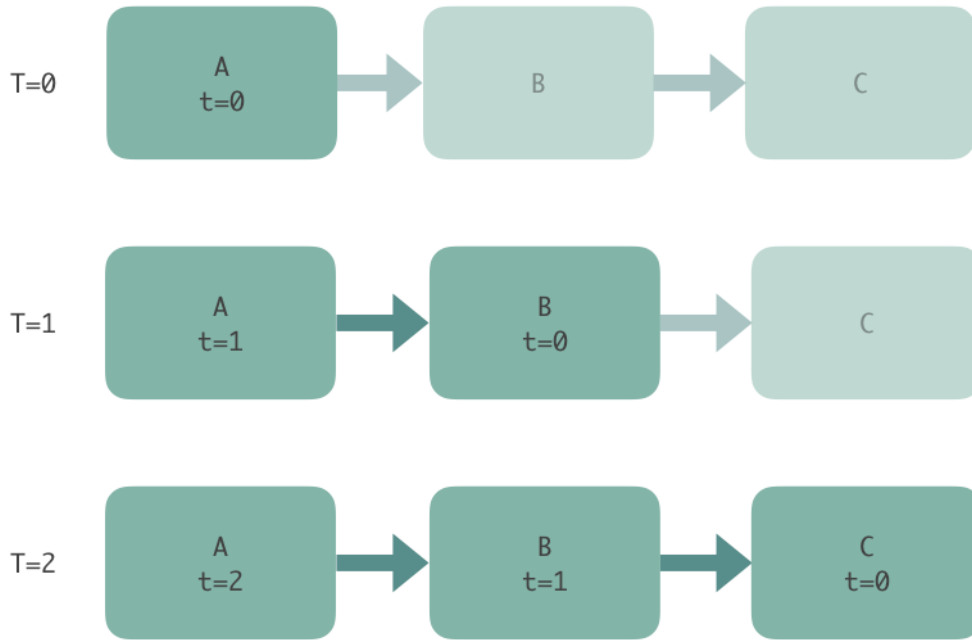


Figure 6-2: Example pipelined execution for basic system.

The system defines a CLK, which is equal to the execution time of the longest running module, and each module moves to the next time step together. The time it takes for values at a single time step t to move through the system (or the *latency*) is equal to $CLK \cdot (\#\ modules)$, while the frequency with which new outputs are produced (the *throughput*) is equal to CLK. Notice that, since CLK is equal to the longest running time of all the modules, the latency is at least as bad as in the original system; however, assuming that the modules are evenly balanced (i.e., CLK is around the running time of all the modules), the throughput can be, in the best case scenario, equal to the original running time divided by the number of modules.

Pipelining is most often used in physical systems, both on a small scale in components such as hard drives, and on a large scale in production lines, which are ubiquitous in manufacturing. More relevant to our application, it can also be applied in a similar manner to modular software systems. Generally, however, traffic simulations do fit the mold for such systems. We found only one previous paper, [11], that mentions pipelining. This lack of discussion is likely due to the fact that pipelining the system by modules[2] will only produce a constant-factor speedup, which, while an improvement, does not provide scalability.

[2]Note that "module" here refers to a standalone part of some execution, not the specific DynaMIT *modules*. Misleadingly, pipelined "modules" in the context of DynaMIT would actually correspond to the *processes*, State Estimation and Prediction/Guidance.

Running DynaMIT's Prediction/Guidance and State Estimation in parallel, for example, will give us a speedup in most cases of only 10%, because Prediction/Guidance takes considerably longer to run. To add insult to injury, this running time is not only insufficient, but it also doubles memory use, because each module requires a complete copy of the system.[3]

Instead, we aim to pipeline the traffic simulation by partitioning across *nodes* as opposed to the *processes*. We refer to this idea as ***node pipelining***. Node pipelining relies on the idea that traffic effects do not propagate instantaneously. Assuming that nodes are ordered left to right, this assumption allows us to divide nodes into bands of sufficient width such that, within a given time step, a node in band i will only be affected by updates to nodes in bands $i \pm 1$, assuming bands are indexed contiguously, allowing us to pipeline the system by band. Not only will the number of nodes per band scale linearly in the number of processors, but the pipelining can also be implemented such that it is unnecessary to store multiple states of the network, and we can start the execution of bands left to right as one would execute a pipelined system.

6.2.2 Parallelization via Vertex Coloring

Suppose, instead, that we first impose an ordering on bands, and then order nodes to conform to this ordering. Then, given bands of sufficient width such that non-adjacent bands are independent, we need merely to determine groups of pairwise independent bands, and can then launch all bands in such a group in separate threads. This effectively reduces the parallel execution problem to one of vertex coloring,[4] where the vertices are the domains into which the nodes of the network are partitioned, and the edges, between all adjacent vertices, represent the dependencies in execution, i.e., the fact that two domains' executions may affect each other within a single time step.

Given a valid coloring of the graph, all vertices of a single color can be executed in parallel without race conditions. In practice, given a k-coloring of our partition, we then assign to each processor a cluster of k neighboring partitions, each of a different color, and all pro-

[3] This is similar to how a lookahead functionality may be implemented.

[4] http://mathworld.wolfram.com/VertexColoring.html

cessors will execute each color in parallel, and update their neighbors' values accordingly. In order for the dependencies between nodes to be satisfied, the execution must be such that all nodes of the first color executed are updated before all nodes in the second color in the serial execution, all nodes in the second color executed before those in the third, etc..

6.3 Update and Independence Properties

Before presenting our parallel implementation, we define two types of independence of execution: *update independence* (also referred to below as "update properties"), and *full independence* (referred to below simply as "independence"). The former refers to the independence of various network elements within the advance step for a single packet, while the latter refers to the independence of network elements given the advance of two or more packets.

To introduce both of these concepts, we make the following assumption: [5]

Assumption 6.3.1. *The local effects of traffic propagate at some finite c m/s or less, where "effects" refer to movements of travelers based only on what they are currently experiencing on the road, without any outside guidance.*

This assumption implies that, in an implementation that uses discrete time steps, updates to any given packet will only depend on the current state of the network (i.e., the congestion on a given segment) within a certain physical distance. In the real world, this property is upheld by physical limitations of the network. It is simply not possible for traffic to travel from Harvard Square to Central Square within a 5 second update interval[6], and by extension, the *effects* of this traffic will not travel that quickly. In DynaMIT, these restrictions are enforced by the discrete time step model: packets are only moved over at most a few links in any given time step. We can take the maximum distance of such advances over the network as c. This enforces the following property:

[5] Note that pipelining always occurs within a single simulation interval. We cannot pipeline over interval boundaries, because the results of prediction guidance are not necessarily contained within the area local to a given traffic effect. For example, if the network state develops congestion in a certain area, this will have an effect on the recommended routes given over the next simulation interval.

[6] A distance of roughly 1 mile, for non-locals.

Lemma 6.3.1 (Separation Property)**.** *During the advance of a packet p originating at node n, DynaMIT will not update the state of any node m if $||m - n|| > h \cdot c$.*

Remark 6.3.1. We use set notation $p \in n$ to denote the fact that packet p is at node n (i.e., p is on a link with down node n) at the beginning of time interval. The magnitude of the difference between two nodes $||n - m||$ is equivalent to the euclidean distance in meters between the two. The comparison of two packets $p < p'$ or two nodes $n < n'$ denotes that the lesser element is updated before the greater one within a single time step.

Proof. This follows from the fact the fact that, within a simulation interval, DynaMIT updates values only to network elements which the packet is currently traversing (link, segment, down node, etc.). Under Assumption 6.3.1, no packet $p \in n$ will reach any such network element contained in m.

□

In attempting to introduce parallelism without the accompanying race conditions, we want to formally define which network elements can safely be updated in parallel given Assumption 6.3.1 and its accompanying Lemma 6.3.1. To accomplish this we define two types of properties. The first of these encompasses the *update properties* of packets, nodes, and bands, more formally defined in §6.3.1. These dictate the network elements whose state will be affected by the advance of a single packet in a given timestep. Crucially, the fact that a network element is not affected by a packet's update does not mean that the ordering of these updates does not matter. To enforce this, we need to define the stronger *independence properties*, which we discuss in §6.3.2. Interestingly, while full independence is stronger than update independence, these properties allow for similar, but distinct, versions of our proposed parallel implementation.

6.3.1 Update Properties

The discrete time step model, in which each packet is advanced for the full time step before the next packet is advanced, gives us the following property:

Lemma 6.3.2 (Update Property). *For any node k, updates to packets at node k at time t will depend solely on the state of all packets at nodes p, $p < k$ at time t, and all packets at nodes q, $q \geq k$, at time $t-1$.*

Proof. This follows directly from the DynaMIT `advanceTraffic` step. Packets at each node are advanced one by one through the network. This means that, when advancing a given packet through the network, the state of all other packets is static. Thus any packet that originated at a node that has already been executed will be already have completed execution for time t, while any packet that originated at a node that has not yet been executed will not yet have begun execution for time t, and thus will be at time $t-1$.

□

Combined, these lemmas provide the basis for the relationship between updates of bands as opposed to individual nodes. First, we formally define a network band:

Definition 6.3.1 (Network Topology Band). A band in a network refers to a contiguous set of nodes; i.e, for any two nodes n, $n' \in b$, there must be a path between the two nodes that traverses only nodes that are also contained in b. A band is thus equivalent to a geographic area, although it is defined by its member nodes as opposed to its boundary.

It is worth noting that this definition says nothing about the ordering of the nodes in the bands, nor does it explicitly rule out the possibility of non-grid based band constructions such as a bullseye shape. For now, we assume the ordering of the nodes is such that they are processed left to right, i.e., all nodes in the first band b_1 are processed before those in b_2, etc.,[7] and that bands are laid out in a one or two-dimensional grid.[8] Corollary 6.3.1 below follows directly from this ordering. We partition the nodes into n bands of (geographic) width at least $h * c$, where once again h is the step size and c is the propagation rate. Let the function $Advance(i, t)$ denote the update function, and $S(i, t)$ denote the state of band b_i at time t[9] for a given time step t.

[7]This changes in our reduction to vertex coloring in §6.4.1; however, because the ordering of the bands is flexible, this will not have a significant impact on the efficacy of the algorithm

[8]Different band layouts will also not affect the algorithm, but make for more difficult explanations.

[9]The update step for a band will be identical to a portion of the serial update step (i.e., if $n = 1$, then $Advance(1, t)$ will be equivalent to running the serial execution for a single time step).

Corollary 6.3.1. *(Band Update Property) Given a partition with bands of width at least $h * c$ and DynaMIT update function f, the following holds for all bands b_i and times t:*

$$S(i,t) = f(S(i-1,t), S(i,t-1), S(i+1,t-1))$$

Less formally, the state $S(i,t)$ depends solely on the state of the previous band at the current time step t, and the states of the current and subsequent bands at the previous time step $t-1$. This follows directly from Lemmas 6.3.1 and 6.3.3.

6.3.2 Independence Properties

We next define the *independence properties* of various network elements. As mentioned above, independence is a stronger property than update independence. We first define independence for packets as follows:

Definition 6.3.2 (Packet Independence)**.** Two packets i, j are said to be independent at time t if the state of the network at the end of execution for t is identical regardless of whether i or j is advanced first.

As discussed in §3.4, the order in which packets are advanced depends on their down node (the node at the end of their current link) at the beginning of the simulation interval. We can thus derive the notion of *node independence* from our definition of packet independence:

Definition 6.3.3 (Node Independence)**.** Two nodes n, m are said to be independent at time t if, for any pair of packets $p \in n$, $k \in m$, p and k are independent at time t.

This provides the basis for the following lemma[10]:

[10] To handle fencepost errors, we actually want $d > 2h \cdot c + \epsilon$, where ϵ is equal to the maximum distance of each of the m and n's incoming links.

Lemma 6.3.3 (Node Independence Property)**.** *Given two nodes m and n, and step size h (in seconds), m and n are independent if $||m - n|| > 2h \cdot c$ and there exists no node q such that $||m - q|| < 2h \cdot c$, $||n - q|| < 2h \cdot c$, and either $m < q < n$ or $n < q < m$, regardless of ordering.*

Proof. This draws on the Assumption 6.3.1, as well as DynaMIT's update step. Assume this were not the case. This means that there must exist some pair p, k, $p \in m$, $k \in n$ such that the state of the end configuration of the network after time t, $S(t)$, differs based on the ordering of the packets. In other words, we have $S(t; p < k) \neq S(t; p < k)$ If p and k are advanced back to back, this is clearly false, due to the Separation Property. None of the network elements p sees will be directly updated by the advance of k, or vice-versa, and thus each packet's view of the network (and by extension, behavior) will be identical regardless of which one is advanced first. However, one could envision a scenario where the advance of p affects the advance of some packet p', which is less than $2h \cdot c$ away from k; this means that k could potentially change its behavior due to the fact that p''s behavior was influenced by p. This is why we enforce the additional property that no nodes closer than $2h \cdot c$ to both p and k can advance in between them; this ensures that there can be no such packet p', because no node is within the update distance of both p and k whose packets could propagate these traffic effects. Thus p and k have an identical view of the network during their updates regardless of ordering. □

A less formal way of thinking of node independence is to say that no nodes are updated between p and k that are within both of their update ranges. A visual representation of this is shown in Figure 6-3. As we can see, no packets advancing starting at p could possibly reach any location that would affect k; the same is true for the closest node p', which updates in between p and k. Our notion of band independence follows:

Corollary 6.3.2 (Band Independence)**.** *Two bands are independent if all of their nodes are independent. This follows directly from Node Independence.*

An important consequence of band independence is that any two independent bands can safely be executed in parallel without producing race conditions or introducing nondeterminism. Because the final view of the network is the same no matter the ordering of the

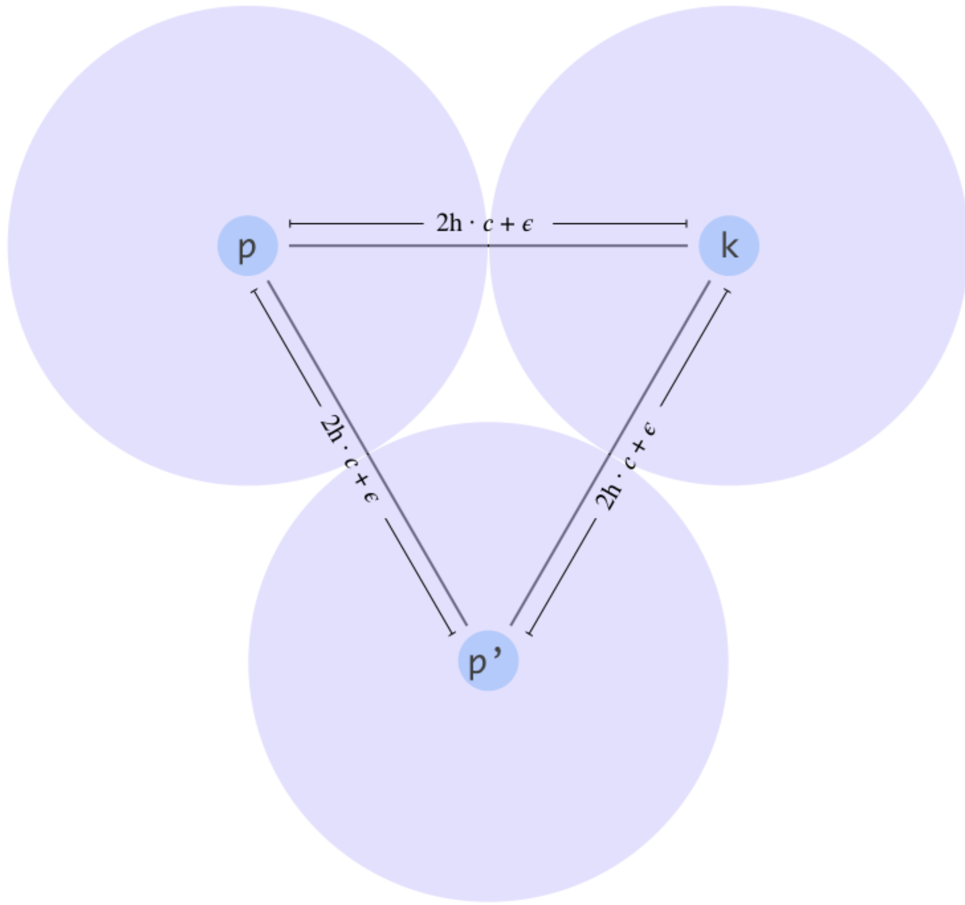


Figure 6-3: Nodes p, p' and k with their corresponding update ranges.

advance of any pair of nodes from each band, we do not need to worry about the fact that processors could potentially interleave their execution.

6.4 Node-Pipelined Algorithm

We first outline node pipelining in 1D[11], and then present an alternate ordering of the algorithm which is easily reduced to a vertex coloring problem, and much better suited to expanding to 2D. The former will make use of update independence, while the latter will make use of full independence.

Let N denote the number of processors allotted to a given traffic simulation. Let T denote the total time for the serial execution of a single time step, and CLK denote the clock rate for our pipelining, which will be equal to the time it takes to execute the update for a single

[11] It is actually possible to only pipeline in 1D, but this will increase running time as the network size increases, because band width must be at least $h * c$, and the height will grow with the size of the network

band[12], giving us $CLK = T/n$, where n is the number of bands. We let $s \in \mathbb{N}$ denote the current clock cycle. Define $p_k(i, i+1, ...)$ as processor p_k, which has been assigned bands $[b_i, b_{i+1}, ...]$ (i.e., if we assigned bands b_4, b_5, and b_6 to the first processor, we could refer to this processor as p_1; if we also want to specify the bands it handles, we use the additional notation $p_1(4, 5, 6)$; this will be useful when stepping through the execution). We then implement node pipelining as follows:

1. Segment the nodes into n bands, where $n = 2 * N$, where n is sufficiently small such all bands have width at least $h \cdot c$, and assign bands to processors in sequential pairs (i.e., b_1 and b_2 go to processor p_1, b_3 and b_4 go to processor p_2, etc.).

2. For each processor p_k, start execution once processor p_{k-1} has run for two clock periods

3. While simulation is not finished:

   (a) for each unfinished $p_k(i, i+1)$, advance band b_i on even s, and b_{i+1} on odd s (assuming s starts at 0)

   (b) after even s, pass updates from band b_i to band b_{i-1} on processor $p-1$

   (c) after odd s, pass updates from band b_{i+1} to band b_{i+2} on processor $p+1$

We want to take the time to make the distinction between N and n. N is the total number of processors, and therefore the amount of parallelization we would hope to achieve, while n is the number of bands. Therefore we would expect a single band to update in time T/n, and phase of the execution update two bands in sequence. Thus the expected time to execute the advance for all bands in a single time step is T/N. However, it is important that we maintain $CLK = T/n$ to allow for correct synchronization between processors.

Remark 6.4.1. Note that, even if with different network elements at different time steps, we do not store multiple states per node. By Corollary 6.3.1 (*Band Update Property*), all we need to ensure valid execution is that, for each band b_i at time step t, both $Advance(i-1, t)$ and $Advance(i+1, t-1)$ have completed before executing $Advance(i, t)$. The alternation between executing even and odd intervals, moving from beginning to end, guarantees this ordering.

[12] In practice this will vary between iterations and bands, but for the sake of argument we take this to be constant

Say we have neighboring processors (assume, for now, that all updates on other processors are performed as needed) $p_k(i, i+1)$, $p_{k+1}(i+2, i+3)$, and are starting $Advance(i, 0)$ at clock period s. Following our execution steps, $Advance(i+1, 0)$ will run at $s+1$; then, at period $s+2$, we will execute $Advance(i, 1)$ and $Advance(i+2, 0)$. It is possible to perform the latter two advance functions, since they only depend on $Advance(i+1, 0)$ having completed. At $s+2$, we run $Advance(i+1, 1)$ and $Advance(i+3, 0)$. $Advance(i+1, 1)$ depends on $S(i, 1)$ and $S(i+2, 0)$ both of which were updated in the previous clock period, and its current state $S(i+1, 0)$; $Advance(i+3, 0)$ only depends on $S(i+2, 0)$ in this case. Then, at $s+3$, we execute $Advance(i, 2)$ and $Advance(i+2, 1)$, which depend on $S(i+1, 1)$ and $S(i+1, 1)$, $S(i+3, 0)$, respectively, which, again, are the current values stored in bands b_{i+1} and b_{i+3}. Thus, as we move along in our execution, for any band b_i, the values in neighboring bands will be exactly those needed to update b_i at any given time. A step through of the execution for 10 bands is shown in Table 6.1 below.

	p_1	p_2	p_3	p_4	p_5
1	$A(1,0)$				
2	$A(2,0)$				
3	$A(1,1)$	$A(3,0)$			
4	$A(2,1)$	$A(4,0)$			
5	$A(1,2)$	$A(3,1)$	$A(5,0)$		
6	$A(2,2)$	$A(4,1)$	$A(6,0)$		
7	$A(1,3)$	$A(3,2)$	$A(5,1)$	$A(7,0)$	
8	$A(2,3)$	$A(4,2)$	$A(6,1)$	$A(8,0)$	
9	$A(1,4)$	$A(3,3)$	$A(5,2)$	$A(7,1)$	$A(9,0)$
10	$A(2,4)$	$A(4,3)$	$A(6,2)$	$A(8,1)$	$A(10,0)$
11	$A(1,5)$	$A(3,4)$	$A(5,3)$	$A(7,3)$	$A(9,1)$
12	$A(2,5)$	$A(4,4)$	$A(6,3)$	$A(8,3)$	$A(10,1)$

Table 6.1: Execution with 10 bands (5 processors), with the current *Advance* process (abbreviated as A) for $s = [0, ..., 12]$. Note that, for every execution of some $A(i, t)$, $S(i-1, t)$ and $S(i+1, t-1)$ will have been set on the previous clock period, and are constant during the current clock period. Completion of the final node in a time step is marked in blue.

6.4.1 Reduction to Vertex Coloring

While the algorithm described in §6.4 above clearly resembles other pipelined systems, its execution and dependencies differ from most traffic simulation systems. We present an extension of this implementation, using an alternate ordering of the nodes in the networks,

which allows for a reduction from the ordering of bands in a pipeline to a vertex coloring problem, which we describe below.[13] This new framing more closely resembles domain decomposition, offering familiarity to developers; simplifies the allocation of jobs to processors, cutting back on overheads associated with parallelization; and allows for all bands to be within a single time step of each other at all points of their execution, a desirable property if we want to be able to simulate real-time updates.

We construct bands as in the previous approach, but in this case with the requirements that bands must be at least $2h \cdot c$ wide. We reorder the nodes such that all nodes $k' \in b_i$, $2 \mid i$ are updated before all nodes $k \in b_j$, $2 \nmid j$. This will allow us to avoid the startup time seen in Table 6.1, where each processor must wait for the previous ones to have begun running before starting its own execution.

Remark 6.4.2. Node ordering is not entirely arbitrary in DynaMIT. For now, we take this to be an acceptable disruption to the original ordering. We discuss the validity of this assumption in more detail in §6.4.2.

While the previous construction only guaranteed update independence between non-adjacent bands due to the node order, the updated ordering and wider band requirement achieves *full independence* for non-adjacent bands in the network. This means that it is now possible to advance all even bands in parallel at time t, and then all odd ones for time t, then even at $t + 1$, and so on. Sample execution for this banding approach is shown in Table 6.2.

The pros of this banding approach include:

1. Has both throughput (frequency at which outputs are produced) and latency (time to execute updates for a single timestep t) equal to $\frac{T}{N}$ (as opposed to original banding, which has the same throughput $\frac{T}{N}$, but latency equal to T)

2. At any given clock time, any two bands will be at a state at most one time step away

[13] Note that while vertex coloring in the general case is NP-hard, the graphs for which we will be computing colorings have well-defined minimum colorings.

	p_0	p_1	p_2	p_3	p_4
1	$A(0,0)$	$A(2,0)$	$A(4,0)$	$A(6,0)$	$A(8,0)$
2	$A(1,0)$	$A(3,0)$	$A(5,0)$	$A(7,0)$	$A(9,0)$
3	$A(0,1)$	$A(2,1)$	$A(4,1)$	$A(6,1)$	$A(8,1)$
4	$A(1,1)$	$A(3,1)$	$A(5,1)$	$A(7,1)$	$A(9,1)$
5	$A(2,2)$	$A(2,2)$	$A(4,2)$	$A(6,2)$	$A(8,2)$
6	$A(0,2)$	$A(3,2)$	$A(5,2)$	$A(7,2)$	$A(9,2)$
7	$A(1,3)$	$A(2,3)$	$A(4,3)$	$A(6,3)$	$A(8,3)$
8	$A(0,3)$	$A(3,3)$	$A(5,3)$	$A(7,3)$	$A(9,3)$
9	$A(1,4)$	$A(2,4)$	$A(4,4)$	$A(6,4)$	$A(8,4)$
10	$A(0,4)$	$A(3,4)$	$A(5,4)$	$A(7,4)$	$A(9,4)$
11	$A(1,5)$	$A(2,5)$	$A(4,5)$	$A(6,5)$	$A(8,5)$
12	$A(0,5)$	$A(3,5)$	$A(5,5)$	$A(7,5)$	$A(9,5)$

Table 6.2: Sample execution for 5 processors using alternating banding.

from each other; this is desirable when pushing real time updates to the nodes

3. Simplifies starting of processors, allowing for more generalized implementations and a easier extension to 2D banding

This banding not only has a greater resemblance to the domain decomposition parallelization, but it also provides a simple reduction to a vertex coloring problem. First we define a band dependence graph:

Definition 6.4.1 (Band dependence graph). Let a band dependence graph G be a graph corresponding to a banding of a network, where the vertices of G are valid bands (i.e., bands of width at least $2h \cdot c$), and edges represent a dependency between two bands (i.e., if there is an edge between ands b, b', then they are not independent).

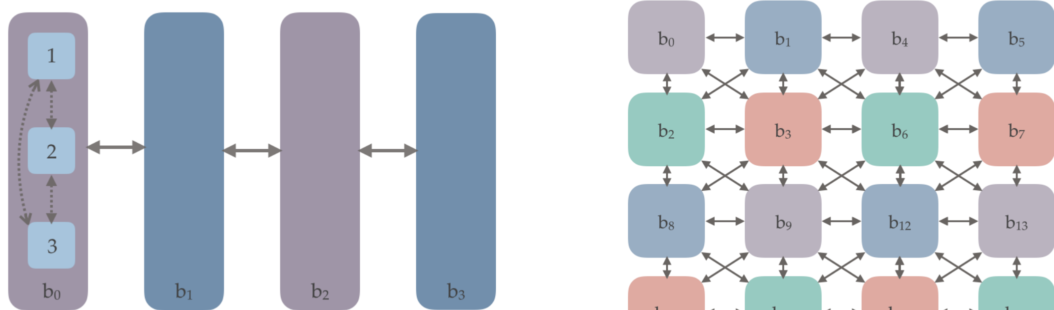
We extrapolate on this definition to relate a coloring of a band dependence graph to a valid execution of the bands:

Corollary 6.4.1 (Band-coloring lemma). *Given a valid coloring of a band dependence graph G, any two bands of the same color can safely be executed in parallel.*

Proof. Follows directly from Corollary 6.3.2. □

While in theory, proving that independence holds is sufficient to guarantee determinism, modeling the dependencies as a vertex coloring problem creates an elegant way to assign bands to processors in practice. Examples of band dependence graphs, with given colorings, are shown in Figure 6-4. Edges exist only between vertices that are *physically* adjacent.

This reflects the fact that dependencies between bands are dictated by geographic proximity in the network.



(a) Coloring of band dependence graph with 1D banding. Internal node dependencies are shown for b_0.

(b) Coloring of band dependence graph with 2D banding.

Figure 6-4: Examples of band dependence graphs in 1D and 2D. For non-color readers, color is denoted for band b_i by $(i \mod 2^D)$.

Given a k-coloring of the graph, processors are each allocated k bands. CLK is still equal to the execution time of a single band, and processors execute each color in parallel, synchronizing between each color. Note that, given a grid layout in the 2D case, we cannot do better than a 4-coloring, because the dependency graph contains cliques of size 4 (the chromatic number of a graph,[14] i.e., the size of its minimum coloring, must be greater than or equal to its clique number,[15] the size of its maximum clique[16]).

If nodes are ordered according to color, such that all nodes of color 0 come before those of color 1, and so on, this execution maintains the same invariants that the previous pipelining algorithm did; namely, that a preceding band is always updated before a succeeding band within the same time step, and does not begin the following time step until the succeeding band has finished the current one.

6.4.2 Notes on Node Ordering

In an ideal simulation, when advancing packets at a node, all of the packets on the downstream links will already have been advance, so the system will know the number of spaces available on these links. If packets on these downstream links have not yet been advanced,

[14] http://mathworld.wolfram.com/ChromaticNumber.html

[15] http://mathworld.wolfram.com/CliqueNumber.html

[16] The proof of this is fairly straightforward via contradiction and the pigeon hole principle.

then it is necessary to estimate the number of spaces that *will* be available after they have been advance. This is referred to as a processing dependency. As discussed in the DynaMIT Programmer Guide (Ch. 11.2.2), the nodes in DynaMIT are intended to be ordered such that the number of such dependencies is minimized. In other words, the ordering of the nodes ought to reflect the flow of traffic.

Both parallel implementations discussed above make assumptions about the ordering of nodes. These assumptions may not be consistent with the optimal ordering to minimize processing dependencies, particularly as this ordering will change over the course of the day (rush hour traffic tends to flow in opposite directions in the mornings and evenings, for example). Because this ordering is not currently enforced in DynaMIT, and scalability is essential to its efficacy as a guidance system, we believe that enforcing an ordering of nodes that enables parallelization is more important at this juncture. However, moving forward, band partitioning ought to take these other dependencies into account, should they be enforced in the future.

At present, it is worth noting that the vast majority of links are band-internal, and the relative ordering of nodes *within* a band has no impact on the parallelization. Thus the disruption to the final simulation will likely be small regardless of how nodes are allocated to bands. However, this disruption can be minimized completely by finding a min-cut partition of the network, as discussed in [8]. By weighting the edges in our graph according to expected traffic flow, we can generate a partition that does less to disrupt processing dependencies. Generating such a partition can be prohibitively expensive, computationally, but can be done ahead of time based on historical traffic data.

At present, we content ourselves with a naive partition to demonstrate the validity of the parallelization. While the node ordering may differ from the original serial execution, it will *not* differ between runs of the parallel implementation, and therefore will not affect determinism, which is our main concern here. When node ordering is enforced in DynaMIT, a longer discussion as to the importance of a min-cut partition can be taken up.

6.5 Implementation on a Multi-Core Machine

We outline the implementation of node pipelining using a reduction to vertex coloring on a multi-core machine. In line with our goals, this implementation is a lightweight addition to the existing system. By design, we do not have introduce additional threadsafety or merge handling into the existing code. The vast majority of the changes will lie in expanding the codebase to include colors and bands, whose representation we discuss in §6.5.1, and in actually generating the partition, as described in §6.5.2. Finally, we must update the way we call our advance and update steps in order to call the parallel version, as highlighted in §6.5.1.[17]

6.5.1 Updated Network Representation

We make two main changes to the network representation in order to enable parallelization: first, we introduce the additional band and color representations; and second, we update the representation of packets to include their down node ID and their current distance to this node, and store packets at the band level as opposed to at segments and links.

We update the Supply module to contain a vector of colors, which in turn contain a vector of pointers to their child bands. Each band contains a vector of moving packets and virtual packets. Once a packet has arrived at its location, we store it in a vector of arrived packets stored in the supply module, so that it can be offloaded and its travel time can be reported at the next update interval. Bands do not store their member nodes explicitly; rather, a band is represented implicitly via a lookup table that maps node IDs to band pointers.[18] While this representation may seem less intuitive, it is more efficient in terms of both memory use and execution time, as there is no reason for bands to have direct access to their member nodes in our implementation.

Updates to Advance Traffic

The `advanceAllPackets` step now iterates over all colors in the network, which then launch threads to advance all their bands in parallel. To advance packets within a band, it is no

[17] Note that, should we want to execute this across multiple machine as opposed to multiple processors, we will have to design a more complex protocol to synchronize across servers; however, this is outside the scope of this thesis.

[18] Because DynaMIT ensures that nodes' IDs are contiguous starting at 0, we can implement this lookup table as an array as opposed to a hash map.

longer necessary to iterate over all of the network topology; we merely sort the packets in each band based first on their node ID, and then their distance to their down node. Pseudocode for this updated function can be found in Fig. 6-6.

This does mean that we potentially sort larger number of packets at a time, which could be problematic because sorting takes worst-case $O(n \log n)$ time[19]; however, in practice, the packet vector will be mostly sorted each time this function is called, because packets will be advanced in roughly the same order at each interval. There may be some small perturbations due to packets moving between bands and nodes, but generally we can expect the packets to the mostly sorted, which can give performance approaching linear time depending on the sorting implementation.

The previous implementation resorted all of a node's up link packets each time; while packets were stored in order at the segment and lane level, it was still necessary to merge the packets between each lane on a segment, and then again between each link every time. This required additional asymptotic running time, in addition to the overhead of creating and destroying data structures to hold the sorted list of packets each time.

Advancing virtual packets now operates the same way. Each color's bands are called in parallel. Packets are already sorted based on the order in which they were added to the virtual queue. It suffices to perform a stable sort (i.e., a sort that preserves the original ordering of elements a, b if $a \not< b \wedge a \not> b$), again ordered based on down node, and advance as in the original implementation.

Moving Packets between Bands

The above implementation assumes that packets are stored at the correct band. We must update the packet list after each advance interval to ensure that this is the case. After a packet is advanced, its down node is updated based on its current position.[20] Once all packets' have completed the current interval, we iterate over all the packets in each band.

[19] With superlinear worst-case complexity, in the scenario where all packets are completely shuffled, sorting $\frac{d}{n}$ packets at n nodes will be more efficient than sorting d packets all at once

[20] We do not do this during the advance step because some of the logic within `advancePacket` relies on knowing whether a link that a packet has already been advanced this interval; we can easily determine this by checking if that link's down node is less than the packet's current node.

For each packet, we use the band lookup to see which band they should be in based on their down node, and send the packet to this new band, if necessary.

As with the sorting of packets, while the running time of this function could be high in the worst-case scenario, in practice the number of packets moving between bands at a given time step will be small relative to the total demand; in tests in the CBD network, we found that this reshuffling step took around 2.64% of the time it took to advance packets within the same time step. As band size grows, the area of a band will grow quadratically with its perimeter; thus the demand in a band to grow quadratically with the number of packets moving in or out of this band each time step. This means that this ratio will likely be even better on larger networks.

6.5.2 Generating Network Bands

As discussed in §6.4.2, we constrain our current partition to one that conforms to either a 1D or 2D grid. To partition the bands, we sort the nodes topologically along each axis in our partition, and define boundaries such that bands are balanced along each *individual axis* based on a given weighting of the nodes. If demand at the current time is available, we assign the weight of each node to be equal to the number of packets that have that node as their down node; if no demand is available, each node is assigned weight 1.

We balance the weight along each axis independently to prevent the need to verify that each pair of bands in a given color is sufficiently far apart. If we first balance the load along one axis, and then balance the load along the second axis based on the existing partition, load will be more evenly balanced, but borders between bands will not necessarily be aligned along the entire axis, potentially producing undersized bands. Fig. 6-5 provides an example of such a partition, balanced first along the x axis, then along the y axis. This partition is perfectly balanced, with each band receiving two nodes (which we take to all be of weight 1); however, due to the distribution of the nodes, there are two nodes assigned to different bands of the same color that are within update distance of each other.

Performing checks to prevent this can be computationally expensive. Each of the $w \cdot h$ bands needs to be checked against its neighbors, and its neighbors' neighbors, to ensure

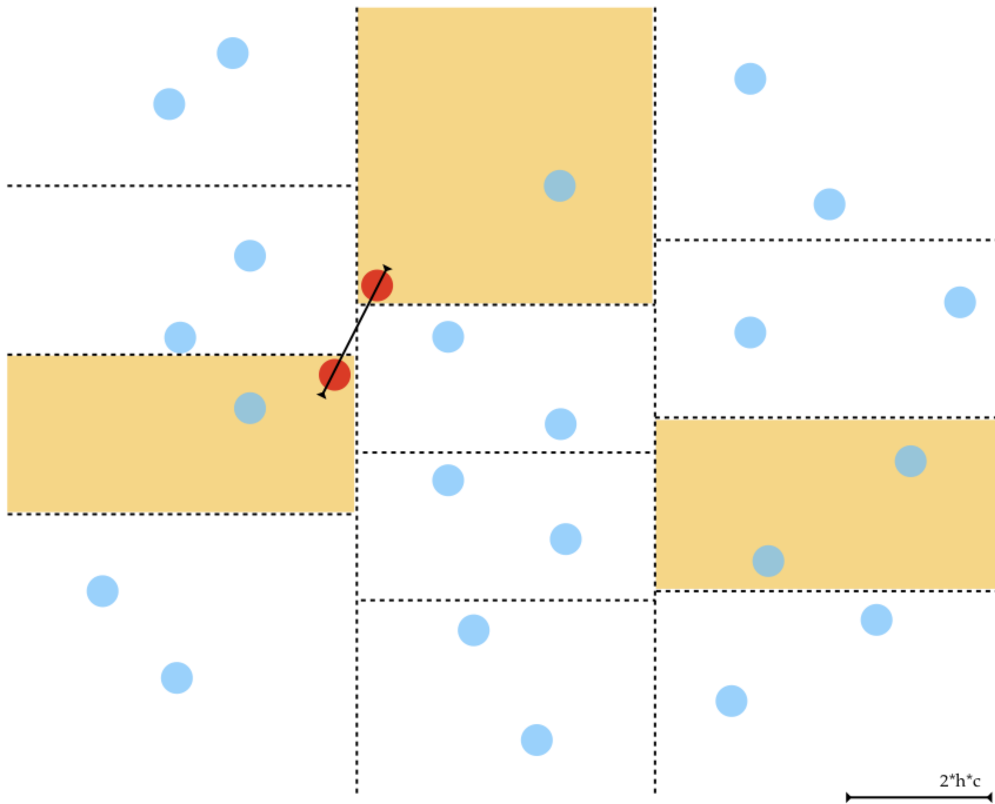


Figure 6-5: Perfectly balanced partition with node weight 1 and insufficient band spacing.

that bands of the same color are sufficiently far away from each other. Furthermore, because these neighbors are not aligned, it would potentially be necessary to perform a binary search over the band's color to determine which neighbors to check; because there are a constant number of colors (therefore the number of bands in each color is linear in the total number of bands) this will take $O(\log(w \cdot h)$ time per band, for a total running time of $O(w \cdot h \cdot \log(w \cdot h))$ to perform the validation.

In contrast, if we partition along each axis individually, we can keep track of the width of each band as we go along, and add additional nodes if the width is insufficient. While this partition may not be perfectly balanced, it will closely approximate a balanced load while vastly improving the running time. This partitioning algorithm takes linear time in both directions independently, resulting in a running time of $O(w + h)$. The efficiency of the partitioning algorithm is important because, to ensure optimal performance, the bands must be adjusted dynamically in order to balance load.

```
1: function advanceTraffic()
2:     advanceAllPackets()
3:     advanceAllVirtual()
4:     loadNewPackets()
5:     cleanupPackets()
6: function advanceAllPackets()
7:     for color c in network do
8:         for band b in c do
9:             launchThread(advanceBand(b))
10:        sync
11: function advanceBand(b)
12:     b.sortPackets()
13:     for packet p in b.packets do
14:         advancePacket(p)
15: function advancePacket(packet p)
16:     (some recursive moving in the network...)
17: function advanceVirtualPacket(packet p)
18:     (some recursive moving in the network...)
19: function cleanupPackets()
20:     for color c in network do
21:         for band b in c do
22:             for packet p in b.packets do
23:                 newBand=lookupBand(p.node)
24:                 if newBand ≠ b then
25:                     newBand.addPacket(p)
26:                     b.removePacket(p)
```

Figure 6-6: Pseudocode for the updated `advanceTraffic` function.

Online Partitioning

While in the original DynaMIT implementation we must iterate over the entire network topology, the bulk of the `advanceTraffic` function is spent advancing packets. Because each color runs only as fast as its slowest band, a poorly balanced network can result in little to no parallelization.

We index bands as one would assign indices in a grid, i.e., a node is assigned to some $band[i]$ in 1D, or $band[i][j]$ in 2D. We assign the first coordinate by iterating over all moving packets[21] to calculate the demand d_n at each node, and the total demand d on the network. Thus our target demand for each axis a is equal $\frac{d}{a}$. We then iterate over the nodes in order along the axis, and begin assigning them to $band[0]$, keeping track of the sum of their demands and the distance between the first node and last node that we have

[21] Generally, there are far fewer virtual packets than there are moving packets, which means that achieving balance for moving packets will have a greater affect on running time.

seen. Once we have hit our target demand $\frac{d}{a}$ for this band, and the band has sufficient width, we begin filling $band[1]$, and so on.

Note that, as a consequence of our implicit representation of bands' nodes, we can assign a node to some $band[i]$ even if our partition is two dimensional (i.e., there is no literal data structure $band[i]$, but instead a collection of bands along this parallel). If we are running in 2D, we simply hold the index of a node's band for the first axis, run the same protocol again along the second axis, and use these values together to index into the band array. To avoid unnecessary checks at runtime, we place the differences between 1D and 2D functionality within compiler guards.

We have to iterate over all packets, which are stored at the band level, in order to sum demand, which will take $O(d + b)$ time, where b is the number of bands.[22] We then have to iterate over the nodes once per axis, which will take $O(n)$, where n is the number of nodes. This gives a total running time of the partitioning algorithm of $O(d + n + b)$.

A sample output from this process is shown in Fig. 6-7, run on the same network as in Fig 6-5. This has also been generated using an x-then-y partition. Note that the node in red, if we were partitioning equally, would originally have been in the band above, as indicated by the dotted line; however, this would have left an insufficient gap between bands, and therefore it was moved down to avoid conflicts. The partition is still fairly balanced: the maximum band demand is equal to 3 as opposed to 2, which does increase running time by 1.5 times; however, this still provides scalability with respect to the network overall. In a larger network, with more nodes and larger band sizes, this error will decrease. Note that a 1D x-partition would be perfectly balanced, while a 1D y-partition would increase the maximum band size from 6 to 7, or by a factor of 1.167.

[22] On network with a reasonable amount of demand, the number of packets will, of course, be larger than the number of bands, as it would be silly set up a parallel architecture to advance one packet for every other band. In practice, therefore, this will actually be $O(d)$.

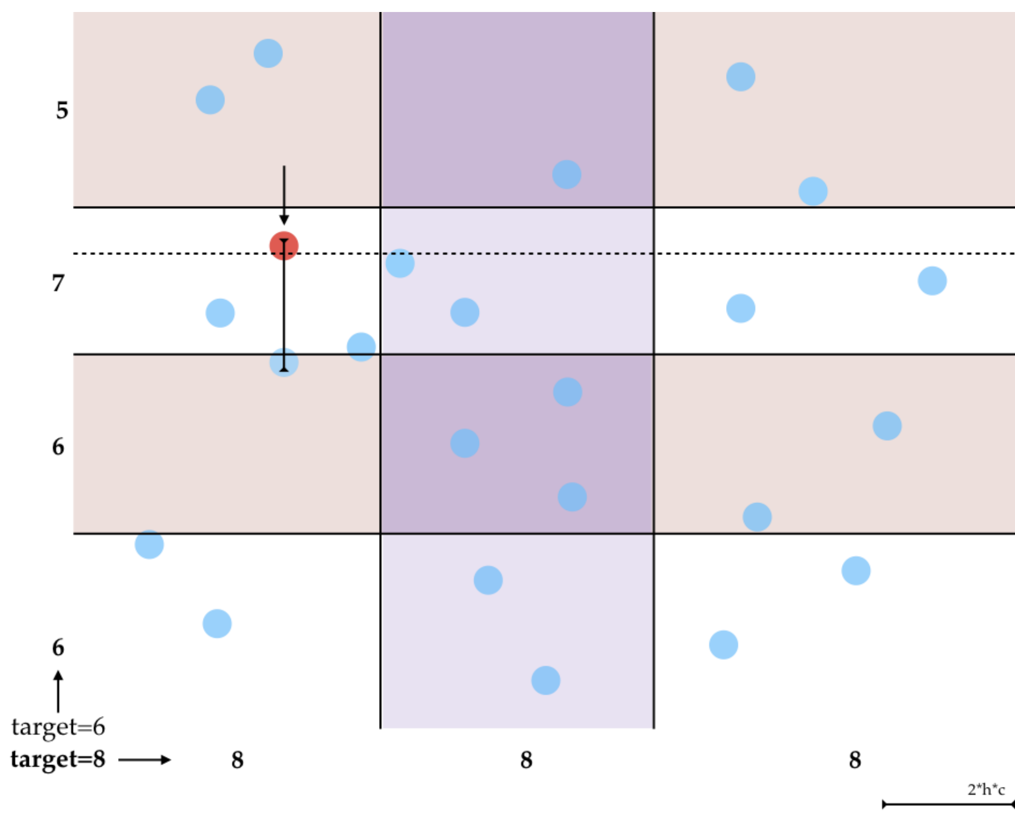


Figure 6-7: 2D partition with axes balanced independently in x-then-y order.

Chapter 7

Results

7.1 Serial-implementation Results

7.1.1 Datatype Improvements

Improvements made to datatypes and control flow in parsing and processing the input files were able to give us incredibly large improvements, taking the startup time for the DynaMIT system on GBA from over 20 minutes, to around 7 seconds. We obtained a more moderate improvement in the smaller Boston CBD network. Original and improved running times are shown in Table 7.1 The times given are only for loading the network, not the demand and pathset, averaged over 10 runs. While this does not produce scalability in terms of the simulation, it has proved hugely important to development and testing, as it allowed us to rapidly test changes to the system without having to wait for half an hour only to run into small errors.

Network	Original Time to Load	New Time to Load	Speedup Factor
Boston CBD	5.235s	3.321s	1.576x
GBA	1278.64s	7.15346s	178.744x

Table 7.1: Running times for original and improvement system loading times on CBD and GBA.

The effects of network size are clear here. While we achieve only a moderate speedup in the relatively small CBD network, the time to load the network on the larger GBA network is decreased significantly. This is to be expected: the previous implementation ran in quadratic time with respect to the network size, while the updated implementation runs in linear time, giving us a more dramatic improvement in a larger network.

7.1.2 Memory Allocation

We assess the impact of improving memory allocation in CBD and GBA with a dummy demand. Using a dummy demand allows us to isolate the running time that comes from the iteration over the network elements. We also test with full demand in CBD.

This change has a greater impact on the time it takes to locate packets on the network than it does the time to update packets; thus these improvements will scale with network size, but not with demand.

We were able to achieve a four-factor speedup in CBD using a dummy demand. Running with a dummy demand on GBA achieves a speedup of 2.5 times. Running times for the original and upated system are shown in Table 7.2. Note that the changes in running

Network	Original	Updated	Factor
CBD (dummy demand)	45.406s	10.9885s	4.132
CBD (full demand)	278.032s	227.436s	1.222
GBA (dummy demand)	1145.982s	539.548s	2.124

Table 7.2: Running times for 6 simulation intervals before and after changes to memory allocation.

time when run with full demand on CBD are minimal. This is due to the fact that, with larger demand, much less time is spent iterating over the hierarchy than spent performing calculations to compute packet movement. However, these changes are necessary in order to update the network representation for the parallel implementation, which we see in §7.1.3 more than makes up for this problem.

Thus we take the updates to memory allocation to be an advantage on uncongested networks, and a necessary change to enable development, although less relevant when run with congestion.

7.1.3 Updates to Network Representation

Before discussing the results of parallelization, we include the impact on running time achieved by changes to the representation that enable the parallelization, namely introducing colors and bands to the hierarchy and storing packets at the band level. Note that these changes necessarily include changes in the memory allocation described above, as

they enable us to initialize the network in the way that we do. We show times for CBD

Network	Original Memory	Updated	Original/representation
CBD (full demand)	227.436s	132.399s	1.718
GBA (dummy demand)	539.548s	406.074s	1.329

Table 7.3: Running times for 6 simulation intervals before and after updates to the network representation.

will full demand and GBA with dummy demand (it would be preferable to run with full demand on GBA, but memory problems prevent this at present). We see that changes to the representation, in conjunction with updates to memory allocation, create a nearly two-factor speedup in the CBD network with full demand over only changes to the memory, and a more modest improvement in GBA. However, we note that the improvement to *memory* in GBA was likely larger because it was run with a dummy demand, so the improvements in GBA may actually be larger in reality.

7.2 Parallelization

7.2.1 Running time evaluation

We evaluate the running time on the CBD network using a 4x1 and an 8x1 partition. We provide numbers for the complete running time for an hour (with an estimation interval of 5 minutes and an prediction horizon of 15 minutes), and the running time of the `advanceTraffic` function for a single interval, averaged over this hour. The results of these runs are shown in Table 7.4.[1]

Partition	Num Processors	Complete Time	Advance Traffic
1x1	1 (serial)	380.105s	1.607s
4x1	2	436.027s	1.269s
8x1	4	387.116s	0.957s

Table 7.4: Full and partial running times for 1x1, 4x1, and 8x1 partitions on CBD.

Recall that the number of processors (i.e., the possible amount of parallelism) is equal to $\frac{b}{2^D}$, where b is the number of bands and D is the dimension of the partition.

The increased running time between the 1x1 and 4x1 partitions is due to the overhead associated with partitioning in contrast to parallelization achieved in the advance traffic

[1] Note that there is some variability in times between these values and memory optimization due to server load at the time of the tests.

step. In a network the size of CBD, there is not enough demand on the network (i.e., the running time of `advanceTraffic` is not long enough) that this parallelization is worth it. However, because DynaMIT runs comfortably within real time on networks the size of Boston CBD, we do not take this to necessarily mean that the parallelization is not successful.

The advance traffic step itself, while not achieving perfect linear speedup in the number of processors (i.e, doubling the number of processors does not cut the running time by 50%), does achieve *consistent* speedup based on the number of processors. We achieve as 21.03% speedup between the 1x1 and 4x1 partitions in `advanceTraffic`, and a 40.44% speedup with 4 processors on a 8x1 partition. While this is not the purely linear speedup we were aiming for, it does provide scalability based on the number of processors. For larger networks, where the overhead can be much lower with respect to network size, we expect to be able to take greater advantage of this.

The next step here is to test this scalability on the full GBA network. While we do not currently have the ability to test this due to memory problems in GBA, we believe that there will be a more marked improvement in this case, as the system spends a comparatively long time in the `advanceTraffic` function on this network.

7.2.2 Determinism

Our implementation was able to achieve perfect determinism without the use of any mutex locks, which was one of the major goals in of our parallel implementation, and one that, as opposed to running time or precise network representation, is widely applicable to other simulation systems.

To evaluate determinism, we implemented a `Checker` class, which logs the demand and virtual demand at each band, and the exact location of each packet on the network at the beginning of each advance interval. A sample log output for `Band 0` with a current demand of 6 is shown in Fig. 7-1. To check whether two runs are consistent, it is simple to compare the output of these log files to ensure that they are identical. We verified this for twenty runs with full demand on CBD using 4x1, 4x4, and 8x1 partitions. Our implemen-

```
BND--------BND
id: 0
demand: 6
virtual: 0
BND--------BND
PKT--------PKT
id: 30
node: 153
link: 1302
loc: 2345
pos: 265.837
qing: 0
virt: 0
PKT--------PKT
PKT--------PKT
id: 33
node: 53
link: 294
loc: 518
pos: 2.39551
qing: 0
virt: 0
PKT--------PKT
PKT--------PKT
id: 12
node: 33
link: 95
loc: 164
pos: 30.7276
qing: 0
virt: 0
PKT--------PKT
```

```
PKT--------PKT
id: 25
node: 86
link: 106
loc: 189
pos: 258.828
qing: 0
virt: 0
PKT--------PKT
PKT--------PKT
id: 26
node: 33
link: 95
loc: 164
pos: 44.1388
qing: 0
virt: 0
PKT--------PKT
PKT--------PKT
id: 48
node: 33
link: 95
loc: 164
pos: 70.9612
qing: 0
virt: 0
PKT--------PKT
```

Figure 7-1: Sample log output for determinism validation.

tation was able to achieve perfect consistency between runs for all of these partitions. We take this to be the larger result from our current implementation, which clearly provides a proof-of-concept for the efficacy of our parallelization scheme in providing determinism.

7.3 Moving Forward

7.3.1 General Changes to the Code

While we attempted to optimize the serial code as best we could, DynaMIT is a very large, and very old, codebase. We took the time to address those aspects of the code that we felt had the largest impact, and were the most applicable to the work on the parallel system presented here; however, we also leave behind a laundry list of potential changes

that could offer important improvements. Many of these are simply stylistic choices that could improve readability (and, by extension, maintainability) of the codebase; however, many of also have the potential to offer real running time improvement. These are less relevant to our present work, and less urgent in terms of the current requirements for the DynaMIT system, but it would be naive to assume that the work presented here could not be built upon to improve the serial execution of the code. These have been discussed in documentation of the system maintained by the ITS Lab.

7.3.2 Memory Use

As there is often a tradeoff between running time and space efficiency, we did little to take memory use into account when making changes to the code. We have made contributions to the improvements in memory use as part of an effort, being made in parallel to this thesis, to dynamically load pathsets into memory; however, these changes were made independently from any running time optimizations. We did take into account the importance of keeping memory use low, i.e., did not introduce additional data structures to the code that could improve running time but would be appreciably large with respect to the current memory use. However, we also did not attempt to make our code compatible with dynamic memory loading. Moving forward, we want to more closely integrate these two processes, to create a better tradeoff between time and memory use.

7.3.3 Parallelization

As mentioned multiple times throughout this thesis, it is impossible to truly assess the efficacy of our parallelization without the ability to test on GBA. That being said, it is also worthwhile to look forward. Additional changes to the code structure will no doubt offer improved performance with respect to parallelization. However, we believe larger improvements will be found in better partitioning algorithms. We discuss a naive partition above, and initial steps to balance load between processors. However, this partition disregards node order, and does little to minimize the movement of packets across boundaries. While DynaMIT disregards node ordering at this juncture, parallelization or no, in the future the parallel implementation ought to address this issue. Disruption to node ordering can be minimized by reducing the number of cut edges in our partition. Generating min-cut partitions is a well-researched area of graph theory, and moving forward we believe a good

first step would be to attempt to implement such as algorithm, as discussed in [8].

Bibliography

[1] Moshe Ben-Akiva, Michel Bierlaire, Jon Bottom, Haris Koutsopoulos, and Rabi Mishalani. Development of a route guidance generation system for real-time application. *IFAC Proceedings Volumes*, 30(8):405–410, 1997.

[2] Moshe Ben-Akiva, Michel Bierlaire, Didier Burton, Haris N Koutsopoulos, and Rabi Mishalani. Network state estimation and prediction for real-time traffic management. *Networks and spatial economics*, 1(3-4):293–318, 2001.

[3] Moshe Ben-Akiva, Michel Bierlaire, Haris Koutsopoulos, and Rabi Mishalani. Dynamit: a simulation-based system for traffic prediction. In *DACCORD Short Term Forecasting Workshop*, pages 1–12, 1998.

[4] Moshe Ben-Akiva, Michel Bierlaire, Haris N Koutsopoulos, and Rabi Mishalani. Real time simulation of traffic demand-supply interactions within dynamit. In *Transportation and network analysis: current trends*, pages 19–36. Springer, 2002.

[5] Moshe Ben-Akiva, Jon Bottom, Song Gao, Haris N Koutsopoulos, and Yang Wen. Towards disaggregate dynamic travel forecasting models. *Tsinghua Science & Technology*, 12(2):115–130, 2007.

[6] CK Chow. On optimization of storage hierarchies. *IBM Journal of Research and Development*, 18(3):194–203, 1974.

[7] George Dimitrakopoulos and Panagiotis Demestichas. Intelligent transportation systems. *IEEE Vehicular Technology Magazine*, 5(1):77–84, 2010.

[8] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 107–114. IEEE, 2001.

[9] Ziru Li, Yili Hong, and Zhongju Zhang. An empirical analysis of on-demand ride sharing and traffic congestion. 2016.

[10] Henry X Liu, Wenteng Ma, R Jayakrishnan, and Will Recker. Large-scale traffic simulation through distributed computing of paramics. *California Partners for Advanced Transit and Highways (PATH)*, 2004.

[11] Bibi Yasmina Yashanaz Mohedeen et al. *Domain Partitioning and software modifications towards the parallelisation of the buildingEXODUS evacuation software*. PhD thesis, University of Greenwich, 2011.

[12] Eoin A O'Cearbhaill and Margaret O'Mahony. Parallel implementation of a transportation network model. *Journal of parallel and distributed computing*, 65(1):1–14, 2005.

[13] Srinivas Peeta and Hani S Mahmassani. Multiple user classes real-time traffic assignment for online operations: a rolling horizon solution framework. *Transportation Research Part C: Emerging Technologies*, 3(2):83–98, 1995.

[14] Marcus Rickert and Kai Nagel. Dynamic traffic assignment on parallel computers in transims. *Future generation computer systems*, 17(5):637–648, 2001.

[15] Vaidy S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency and Computation: Practice and Experience*, 2(4):315–339, 1990.

[16] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, volume 13, 2013.

[17] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*, pages 440–447. Springer, 2004.

[18] Yang Wen. *Scalability of dynamic traffic assignment.* PhD thesis, 2009.

[19] Yadong Xu, Wentong Cai, Heiko Aydt, Michael Lees, and Daniel Zehe. Relaxing synchronization in parallel agent-based road traffic simulation. *ACM Trans. Model. Comput. Simul.*, 27(2):14:1–14:24, May 2017.