Speedup of DTA-Based Simulation of Large Metropolises for Quasi Real-Time ITS Applications

Agop Koulakezian[1], William E. Graydon[2], Hossam Abdelgawad[3], Yi-Chang Chiu[4], Baher Abdulhai[5], and Alberto Leon-Garcia[6]

***Abstract*—The assessment of real-time intelligent transportation system (ITS) applications, such as traffic management and adaptive route guidance systems, requires the use of fast and near real-time dynamic traffic simulation models. Even off-line applications, used for testing planning scenarios, often require fast-enough traffic simulation models that enable the required repetitive simulations. This is even more critical for large-scale networks with millions of vehicles. This paper investigates the speedup of DTA simulation models, using compiler optimizations and parallelism. DynusT as a widely used DTA model was evaluated as a test case, while its results could be generalized because we have used real-networks and calibrated them using real data sets in the Greater Toronto and Hamilton Area (GTHA). Extensive testing is performed to evaluate various dimensions for speed-up including: network size, number of processors, various optimization levels and operating systems. The performance results show that compiler optimizations and parallelism allow to: 1) double the speed required for a 4-hour simulation after 12 iterations to reach equilibrium, and 2) bring down the initial simulation time (required for network loading) by 2.5 times; enabling the testing of various real-time ITS applications.**

I. Introduction

The requirements of a dynamic traffic assignment (DTA) model for modelling ITS applications can be categorized into two modes of use: off-line and on-line. Off-line models are typically used to quantify the performance of existing conditions or investigate the effectiveness of ITS strategies. These applications typically require multiple iterations and scenario evaluations using limited computational resources [1]. The need for on-line DTA applications arises in cases where monitoring the transportation network, estimating the network state, and forecasting the next state all are required to be done in real-time to assist operators and management centers in taking informative decision and implement mitigation strategies in real-time. For example, in urban networks, detecting the arrival patterns and presence of vehicles at a traffic light enables algorithms to estimate the network state (e.g. queue length), and therefore take appropriate action (e.g. control the green time of a traffic light) in real-time. Another example, in rural networks, detecting a high probability of an accident (using incident detection algorithms) would trigger a set of traffic management strategies that need to be evaluated in real-time depending on the characteristics of the accidents.

The need to dynamically model the time-varying flow of vehicles has generated many contributions in developing DTA-based simulation models. The first set of models, called "micro-simulation" models, represent the behavior of each vehicle based on car-following, gap acceptance and lane choice models [1]. Models such as PARAMICS [2], AIMSUN [3], VisSim [4], have a great level of detail and computational complexity; thus, their successful use has been commonly limited to relatively small size networks [1]. The need to model larger networks with reasonable computational times has led to the development of "mesoscopic" simulation models, which provide less detail in modeling of individual vehicle movements but are less cumbersome computationally. Examples include CONTRAM [5], DYNASMART [6] and DynusT [7].

Nevertheless, even mesoscopic simulation models in their typical form do not satisfy real-time requirements when analysing large metropolises with millions of vehicles [8]. The required run-time of DTA models typically consists of 2 components: 1) initial simulation required for network loading and assigning the demand, 2) iterative simulation to reach a certain convergence criterion. The initial simulation run-time depends on the number of vehicles in the simulation and assigning a shortest path for them; while the iterative simulation run-time depends on the efficiency of the iterative traffic assignment algorithm, used to reach the convergence criterion [7], [9]. Thus, when reporting on run-time of DTA simulation models, it is important to breakdown the time for these 2 components. For example, the run-time for 4 simulation hours (Morning Peak period, from 6-10 AM) of the Greater Toronto and Hamilton Area (GTHA) model, reaching equilibrium conditions after 12 iterations, on a Windows machine (using 8 processing cores) is approximately 5.5 hours, while the required time for the initial simulation time and network loading is approximately 42.5 minutes. In our view, these computational times make it challenging to assess real-time traffic management strategies and adaptive route guidance systems. Moreover, in cases where only the initial simulation run (referred to as One-Shot simulation in the literature [10]) is required, e.g. to model the effects of incidents, construction and weather conditions, that run-time of 42.5min in the case of the GTHA, is way beyond what could make the simulation output useful to warn or give drivers alternate routing options.

Therefore, research attention has been devoted to speeding up DTA-based microscopic and mesoscopic simulation models using 2 sets of approaches: 1. Using Parallelism with multi-threading and 2. Partitioning the traffic network into segments handled by different processors [9]. One example of using

[1] [2] [6] A. Koulakezian, W. E. Graydon and A. Leon-Garcia are affiliated with the Department of Electrical and Computer Engineering, University of Toronto. Email: {agop.koulakezian@, b.graydon@mail., alberto.leongarcia@}utoronto.ca. [3] H. Abdelgawad is affiliated with the Department of Civil Engineering, University of Toronto and Faculty of Engineering, Cairo University. Email: hossam.abdelgawad@alumni.utoronto.ca. [4] Y. C. Chiu is affiliated with the Department of Civil Engineering and Engineering Mechanics, University of Arizona. Email: chiu@email.arizona.edu. [5] B. Abdulhai is affiliated with the Department of Civil Engineering, University of Toronto. Email: baher.abdulhai@utoronto.ca.

parallelism is SEMSim traffic simulation [11], which uses multi-threading to speed-up an agent-based micro-simulation, based on the assumption that the agents in SEMSim have complete routes to follow in order to reach their destinations, which reduces the number of route calculations required. Another example is CPU/GPU-based speedup for a mesoscopic simulation based on the "Entry Time-based Supply Framework (ETSF)", which uses the assumption that vehicles on the same lane of a segment are moving at the same speed at a time step [12]. Major issues that need to be considered with such approaches include load-balancing, inter-processor communication and synchronization between processors [9], [12]. On the other hand, an example of using partitioning is MALTA [13], where a network partioning and recursive on-line load balancing tool is used outside of the DynusT [7] mesoscopic simulator for achieving speedup. Major issues that need to be considered for such approaches include finding good partitions, re-calculating the traffic demand matrices after partitioning, minimizing overheads, and modelling traffic dynamics across segment boundaries [9]. Due to the complexity and major issues of using partitioning, the speedup in this paper is based on using parallelism and compiler optimization. However, key differences from [11] and [12] in this paper include focusing on mesoscopic simulation and not imposing assumptions that reduce DTA accuracy as the ones used in [11] and [12].

Contributions: This paper discusses the speedup of DTA simulation models using compiler optimizations and parallelism, and enabling them to run on Linux, designed for High-Performance Clusters. DynusT, which stands for Dynamic Urban Systems for Transportation [7], was used as a mesoscopic simulation platform to determine the efficiency of the compiler optimization and parallelisation on a number of simulation models with different network sizes to test the scalability of the system. The experimental setup was tested on calibrated networks, that had used real data to generate the transportation demand and build the network geometry [14]. This makes the interpretation and generalization of the results suitable to be used in actual implementations of real-time ITS strategies.

In this paper, we first provide a background on User Equilibrium Assignment and describe how it is achieved in DTA models in Section II. Section III and Section IV present the improvement options using compiler optimizations and parallelism of the DTA model, respectively. Section V presents the context of the DynusT DTA model used and the specific optimizations made on it. Section VI presents the performance results of the system on a 2 large-scale models in the GTHA. Finally, we conclude and discuss future work in Section VII.

II. User Equilibrium Traffic Assignment

This section provides a background on the User Equilibrium (UE) Assignment and its implementation in DynusT.

A. Background on User Equilibrium

Traffic assignment is the problem of routing vehicles on a road network from their origins to their desired destinations, given a road topology with network links and a set of vehicle trip demands, represented by an Origin-Destination (O/D) matrix [15]. Assumptions used for a traffic assignment include: 1. all vehicles/users start at the same time, 2. users are rational; they try to reduce their own travel cost, if possible, 3. Demand is constant; although in reality, it varies within the day, day to day and with cost, 4. supply is constant; neglecting the occurrence of incidents or the building of new roads, and 5. users have perfect knowledge of routes and costs.

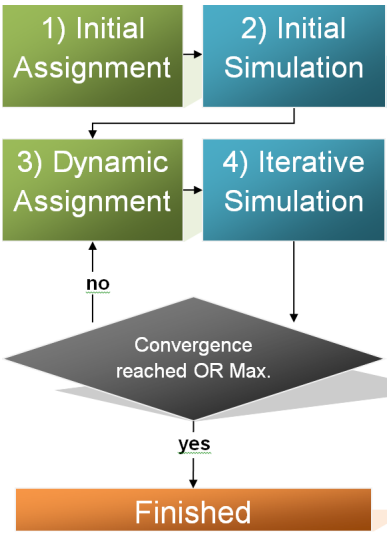


Figure 1: Iterations in a DynusT simulation

The User Equilibrium (UE) Assignment method seeks to maximize each user's welfare, by minimizing individual travel times of all users. This is done through an assignment where all alternate routes between an O/D pair have the same travel time. As no user can reduce his or her travel time by choosing another route, UE is an equilibrium assignment. This is called a static UE assignment in the literature [15]. On the other hand, the dynamic UE assignment relaxes the assumption that all users start at the same time, and instead minimizes travel times such that only vehicles departing at the same time on routes between an O/D pair will have the same travel time [10]. Dynamic traffic simulators implement dynamic UE since it best models traffic behavior of users on real transportation networks. They also relax some of the other assumptions listed above, such as varying supply with incidents, specifying sets of vehicles that have knowledge of routes and costs, varying demand, etc., to model different traffic scenarios.

B. User Equilibrium implementation in DynusT

DynusT [7] is a dynamic mesoscopic traffic simulator, as described above, which means that it adjusts link costs mid-simulation based on their use. In order to simulate UE, DynusT iterates over the following steps (1-4), illustrated in Figure 1:

1) DynusT analyses which route is the shortest for each vehicle, and assigns vehicles initial paths. This is similar to an unfamiliar driver chosing a route assuming free-flow conditions.
2) DynusT performs an initial simulation, in which drivers follow the assigned routes to completion without changing course. This updates the route costs based on the routes taken by the drivers.
3) These initial conditions leave some vehicles taking routes with higher cost than their alterna-

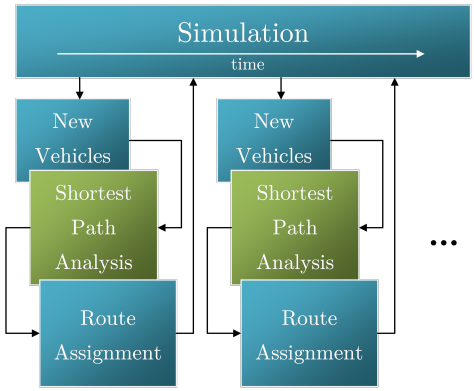


Figure 2: Adding vehicles mid-simulation in DynusT

tives. DynusT will perform dynamic assignment, re-assigning a % of the vehicles to their respective best-routes according to the calculated route costs [10].

4) It will then perform a simulation again, and updates the route costs based on this dynamic assignment.

DynusT will then repeat steps 3 and 4, reassigning traffic based on the latest simulation, and re-simulating with this new assignment. It will stop when one of the following occurs:

1) The simulation iterates a maximum number of times, as assigned by the user running the simulation.
2) The simulation reaches convergence, defined as when all vehicles have no incentive to switch routes, i.e. when the gap between the current assignment solution and the ideal shortest route time, divided by the total shortest path times (a ratio called the relative gap), is below a pre-specified tolerance level [10].

Each simulation emulates a period in time, and as the simulation time progresses, new vehicles navigate through the network. Assigning the routes for these new vehicles based on the shortest paths (determined at the start of the simulation) does not represent reality simply because network conditions dynamically change over time. That is why it is important to distinguish between what is referred to as the instantaneous travel time (cost) vs experienced travel time (cost) within a transportation network. The instantaneous travel time refers to the travel time calculated when the routes are generated without considering congestion during subsequent time periods, while experienced travel times account for the times needed for traversing various links using the expected congestion state of those links during the times of entering those downstream links [10]. As illustrated in Figure 2, DynusT performs basic shortest path analysis and assigns routes for newly generated vehicles. These new vehicles then form a part of the simulation and continue in the same manner as the ones loaded in the previous time step. This process repeats many times per simulation (known as assignment interval), once for each new batch of vehicles entering the network.

It is important to note that due to its algorithmic structure and software implementation, DynusT is capable of performing DTA on regional-level networks over a long simulation period. This is primarily due to the Anisotropic Mesoscopic Simulation (AMS) algorithm [16] used in DynusT to perform the traffic assignment. As a mesoscopic algorithm, DynusT does not simulate car following, lane changing and gap acceptance within individual vehicles (as a microscopic simulator would), however it emulates system responses to factors affecting individual vehicles, such as queues for making left turns. AMS simulates groups of vehicles which are in close proximity using the so-called "Speed Influencing Region" (SIR), enabling it to differentiate speeds and other characteristics for vehicles on the same link but in different SIRs. This is in contrast to other mesoscopic models that imprecisely assume that traffic flow/speed/density are uniform along these links [16]. The next sections present the compiler optimizations and parallelism that can be used to speed up DTA model run-times.

III. Compiler Optimizations

Compiler Optimizations are modifications to how algorithms are implemented in the processors to maximize their efficiency, reducing their run-time [17]. There are many compiler optimizations that are possible for every programme code. These optimizations generally require understanding of the algorithm being implemented and exploiting that provides options for further speedup using parallelism. These options include manual optimization and automatic optimization.

Normal loop

```
for (i = 0; i < 100; i++)
{
    a[i] =b[i];
}
```

After loop unrolling

```
for (i = 0; (i+1) < 100; i += 2)
{
    a[i] =b[i];
    a[i+1] =b[i+1];
}
```

Figure 3: Loop unrolling example, modified from [18]

A. Manual Optimization

Manual optimization is making a set of changes in the way a code is compiled, based on specific knowledge of certain bottlenecks in it [17]. One example is loop optimizations, which act on statements that execute the same operation until the condition to exit the loop is satisfied. These optimizations can lead to significant speedup as programs can spend the majority of their run-time inside loops that might be designed inefficiently [18]. Figure 3 shows an example of loop unrolling, where the body of the loop is unrolled into 2 independent statements that can be easily executed on 2 separate processors at the same time, giving a speedup of around 2 [18].

B. Automatic Optimization

Automatic optimization is allowing the compiler to look through the code automatically to try to find options for automatic parallelism [17]. There are 4 different optimization levels for codes written in C, C++ or Fortran and compiled with the Intel C++ Compiler (ICC) [19]:

1) When compiled with "O0" for "optimization level 0", the code is completely unoptimized.
2) "O1" is optimized to create the smallest (by size rather than time) optimized code in most cases.
3) "O2" is optimized as far as it can deterministically speed up the code.
4) "O3" has aggressive optimizations applied based on heuristic approaches that are likely to boost the speed, but not guaranteed. These include automatic loop unrolling for all loops (which could be beneficial or disadvantageous based on the specific scenario).

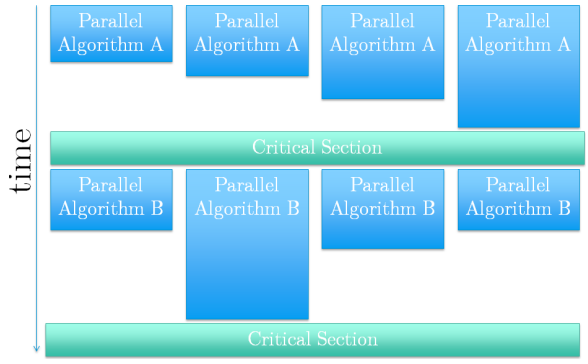


Figure 4: Critical sections resulting in wasted resources

IV. Parallelism

Programming for a high-performance environment to achieve speedup makes use of a number of techniques to maximise efficiency. Multi-core computers have multiple, separate processors (or processor cores), which individually execute sequential commands. The code executed on each processor (core) is known as a thread, and aims to utilize powerful multi-core computers to speed up execution time. While a perfectly parallel programme can run n-times faster utilizing n threads, there are a number of issues that may limit speedup in practical applications. In the following sections we discuss speedup techniques using parallelism and their limitations.

A. Critical Sections of Code and Load Balancing

Critical sections of a programme are sections in which all threads must be at the same place in order for the critical section to run. This is usually due to data the algorithm needs from all the threads, which requires that all threads to finish calculating the needed data for the algorithm to continue. This is problematic because processors that finish first must wait until the slowest one finalizes its execution before continuing. To illustrate, in the case of DynusT, there are 3 sections of explicit critical code, as well as 12 places in which a parallel section ends. Since the sequential section following each ending parallel section will need data calculated in the parallel section, this serves as an implicit critical section. It forces all processors to catch up to the same point before any of them can continue into the sequential code.

Critical sections become especially important if the number of calculations is not spread evenly amongst all processors. For instance, Algorithm A is run in parallel over 4 processors in Figure 4, but 3 processors must wait idly while 1 finishes a longer job before the first critical section. It then parallelises again, but due to poor load balancing between the processors, over half of the processor-time is wasted. If the first critical section in Figure 4 was omitted, then some of the differences between processor execution times would be cancelled, improving the execution time significantly. Note that exploiting the knowledge of these critical sections enables making certain compiler optimizations to improve the load balancing between the processors and thus lead to further speedup.

B. Sequential Sections

Another shortcoming of parallel code is that it does not always scale linearly with the number of processors running

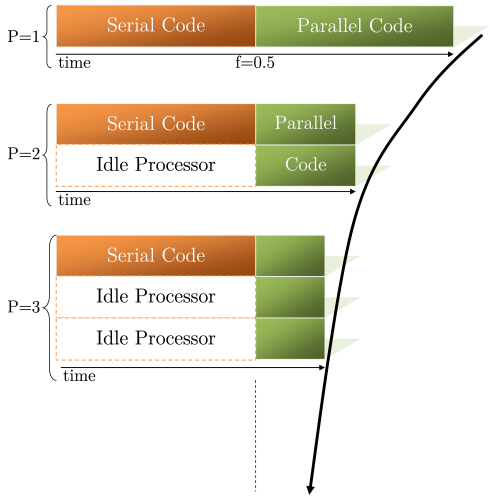


Figure 5: Asymptotic run-time change with more processors

it. A major bottleneck here is sections of the code that must run in series - that cannot be made parallel. Regardless of how many processors we have, the same amount of time must be spent on the serial sections, thus limiting how many times faster the overall code can run. This is called the speedup (S) and this limit is quantified by Amdahl's Law [20]:

$$S = \frac{1}{f + \frac{1-f}{P}}$$

where, S is the speedup; i.e. how many times faster the parallel code runs on multiple processors than its serial counterpart on 1, where P is the number of processors being used, and f is the fraction of the code that can run in parallel ($0 \leq f \leq 1$). This means that increasing the number of processors does not proportionally increase the speedup. In Figure 5, we see an example of code that has an f of 0.5 (50% parallelisable, 50% serial). This makes a theoretical speedup of:

$$S = \frac{1}{0.5 + \frac{1-0.5}{P}}$$

and taking the limit of infinite processors,

$$\lim_{P \to \infty} (S) = \frac{1}{f} = \frac{1}{0.5} = 2$$

This is a well-known corollary of Amdahl's law, and means that the speedup cannot exceed 2. This is because as we increase the number of processors, the parallel code's execution time decreases, however the serial code's execution time remains constant. As the number of processors approaches ∞, the execution time approaches the serial code's time, as seen in Figure 5. This however, represents the ideal case, in which the parallel section scales linearly (doubling the processors doubles the speed). This is not always achievable, due to delays at critical sections as described earlier.

C. Parallelism with Threading

Running a programme using multi-threading involves using the OpenMP compiler extension to implement its explicit parallelism. Threaded code runs across all processors in a single machine — this enables having parallel code running on

at most the number of processors on the machine. For instance, a dual-core machine can only run two threads efficiently. The number of processors a programme is executed on is therefore limited to the number of cores on the machine.

D. Parallelism across Nodes

If we wish to add more processors than the number available on one machine, we can use multiple machines. In most high-performance environments, these are called nodes, and are connected with network cables but are distinct units. A major drawback of multi-node processing is that there must now be communication between the nodes, something that takes time. Furthermore, as described in Section IV-B there are diminishing returns on adding processors. For these reasons, most DTA models do not use parallelism across nodes. The next section covers the context of the DynusT DTA model used and the specific optimizations to exploit its speed up.

V. The Context of the DynusT DTA Model

This section discusses how compiler optimizations and parallelism were used to speed up the DynusT DTA model.

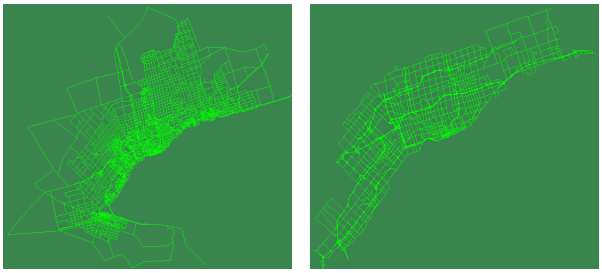
A. Compiler Optimizations on DynusT DTA Model

After analyzing the implementation of the traffic assignment algorithms in the DynusT DTA model (Section II-B) and their performance in detail, we performed many manual optimizations within the DTA model (to enable achieving further speedup using parallelism), including the following: 1. vectorising matrix calculations (e.g. directly summing a vector of vehicle data rather than 1-by-1), 2. removing unnecessary code jumps using loop unrolling, and 3. changing the implementation to reduce memory-access, by making it use as much of the processor cache as much as possible, rather than system memory [18]. In addition, the DTA model was run using the 3 automatic optimization levels, O0, O2 and O3 (O1 was not used as it optimizes for space rather runtime).

B. Parallelism in DynusT

Analysing how parallelism can be used to speed up the DynusT DTA model, our findings showed that explicit critical sections occur only 3 times in DynusT. One of these occurrences is in the section of code that moves all vehicles mid-simulation (vehicles move along a road as time progresses). Periodically in the simulation, DynusT needs to write the information (e.g. location) for each vehicle to an array to use it later for dynamic assignment. It is important that all vehicle locations written to this array are written at the same time, thus making this code a critical section. Since this section occurs in a loop to move all vehicles, the parallel-critical transition scenario similar to Figure 4 happens many times (two are illustrated). For instance, if this cycle were to occur once for each vehicle, the run-time would increase proportionally to the number of vehicles. This amplifies the delay caused by the critical section, for a large network like the Greater Toronto and Hamilton Area with 1.6 million vehicles.

Based on our analysis, DynusT has an empirical f value of 0.163 (the serial part of the code takes 16.3% of the total time with 1 processor as explained in Section IV-B). Thus, its theoretical speedup is $1/f = 6.126$. This represents the ideal case with infinite processors, where the parallel section run-time scales down linearly with the number of processors. This is not the case in DynusT, due to delays at critical sections as described in Section IV-A, which leads to its run-time not reaching that asymptote, as explained in detail in Section VI-C.



(a) GTHA Network (b) GTA Network

Figure 6: Layout of the GTHA and GTA networks

Finally, running with OpenMP enables DynusT to run parallel operations, such as assigning vehicles to a route based on the costs for each route — item 3 in Figure 1. Thus, we used multiple processors to achieve speedup in DynusT (up to 8 processors based on the theoretical speedup findings). The performance analysis results with DynusT are discussed next.

VI. Application to DynusT: Performance Analysis

In this section we apply the compiler optimization and parallelism approaches discussed above into DynusT, a mesoscopic simulation software that is widely used in many cities. DynusT is originally designed as a Windows-based programme. As many High Performance Computing (HPC) facilities [21] are rapidly emerging to be running on Linux systems, the DynusT source code was converted to run on Linux as a first step. As Windows and Linux differ in their performance running the same programme, we discuss some of these differences and show extensive testing on the performance of DynusT for Linux under various conditions.

A. Experimental Setup

In order to assess the performance of the proposed speed-up approaches, the experiments were designed to study the following factors: efficiency (with number of CPUs and optimization levels), performance in various systems (Linux vs Windows), and scalability (with different network sizes).

Table I: Sizes for the GTHA and GTA networks

	GTHA	GTA
n	11,713	14,225
m	29,184	26,444
v	1,981,571	1,602,717

The simulation test networks include those of the Greater Toronto and Hamilton Area (GTHA) [14] and the Greater Toronto Area (GTA), which have been extensively calibrated using real data. These networks are shown in Figure 6 and they differ in size (shown in Table I), based on these 3 criteria: 1. n, the number of nodes (intersections) and origin/destination

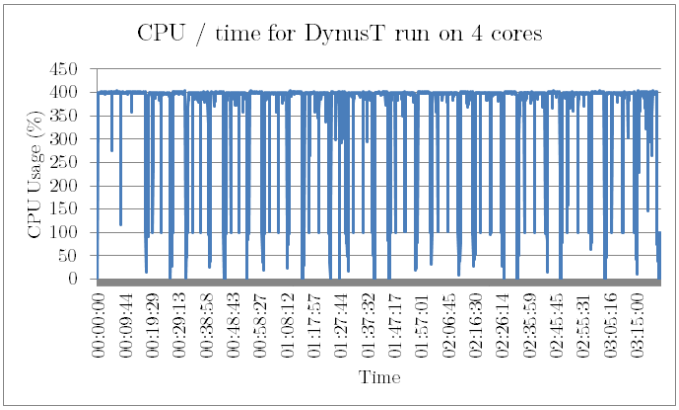


Figure 7: DynusT CPU usage for 12 iterations

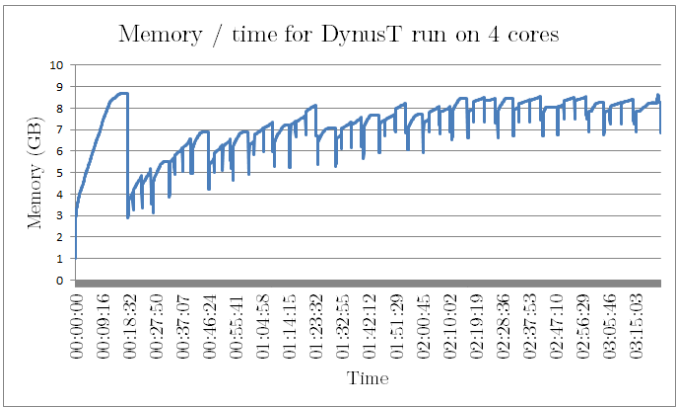


Figure 8: DynusT Memory usage for 12 iterations

points in the network, 2. m, the number of links (roads) in the network, and 3. v, the number of vehicles making a trip in the simulation. Each simulation emulates the morning traffic period from 6 am to 10 am, and a number of iterations to reach user equilibrium conditions. In the case of the 2 test networks, the number of iterations required for convergence was found to be 12. The loading pattern of the vehicles is shown in Table II; although we feed the demand every minute, this table aggregates the demand for every 30-min interval to show the profile of the vehicles loaded into the network. Extensive testing was performed on the performance of DynusT under various conditions, using memory use, CPU use and execution time as metrics. The number of processors used ranges from 1 (serial) to 8 on an Intel Core I7 workstation with 12GB RAM and the compiler optimization levels used are optimization level 0, 2 and 3.

Table II: Vehicle Loading Profile (in thousands of vehicles)

Time (min)	0-30	30-60	60-90	90-120	120-150	150-180	180-210	210-240
GTHA	137	172	291	347	496	370	261	144
GTA	112	147	243	295	416	317	216	115

B. CPU and Memory Usage

DynusT executes a DTA algorithm described in Section II-B, using multi-threading with OpenMP (described in Section IV) for efficiency. To illustrate the required resources to run

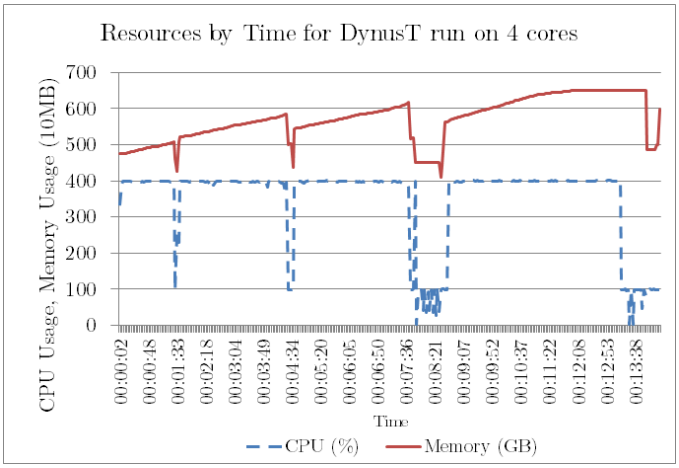


Figure 9: CPU and memory usage during one iteration. Dashed: CPU in %, and solid: memory in GB/100.

the GTHA network until convergence (using 12 iterations), a series of figures are created. In Figures 7 and 8, the CPU usage and memory usage are illustrated against the actual run-time (x-axis) for 12 iterations, respectively. Additionally, Figure 9 helps provide more insights into the resources allocated during one iteration (after the 0^{th} iteration).

Figures 7 and 8 show fluctuations in CPU and memory usage with the evolution of the simulation iterations. These correspond to the steps of the DTA algorithm described in Section II-B. On the one hand, the 0^{th} iteration (during which network-loading and initial short paths are generated for all vehicles as shown in Figure 1) exhibits a large memory peak as seen in Figure 8 and takes longer than subsequent iterations, as expected. On the other hand, Subsequent iterations only assign newly generated vehicles to shortest paths based on DynusT's traffic assignment algorithm (described in Figure 2).

Figure 9 provides a clearer view of the iterative simulation (the initial section until around 8.5 min), and the dynamic assignment following it, described in Section II-B. It also shows the variation in CPU usage, where the periods using 1 CPU (100% CPU Usage, rather than 400%) represent the serial sections of DynusT's code execution.

As seen in Figures 8 and 9, the memory usage within a typical iteration increases overall as the simulation progresses. However, a large drop of memory (and CPU) is observed around the 8 min mark in Figure 9; this is due to freeing of memory and writing vehicle info into files (usually by 1 processor) to be used for reassigning some vehicle paths during the dynamic assignment step. Note that this period for writing vehicle info to files is higher (lower) when the number of vehicles in a network is higher (lower). Other drops in memory are attributed to freeing of memory between iterations and within each iteration, seen around the 1.5 and 4.5 min marks in Figure 9, before new vehicles are loaded into the network (described in Figure 2). Overall, the maximum memory usage reaches around 9GB, seen in Figure 8, which is below the memory capacity of the simulation workstation.

C. Evaluation with Compiler optimizations and Parallelism

In addition to the manual optimizations implemented (as discussed in Section III-A), this section discusses how the execution time of DynusT can be reduced using 2 methods: 1. Compiler optimizations, through automatic parallelism, with levels 0, 2, and 3 for optimization, and 2. Explicitly parallel code, using OpenMP. This is defined by the number of cores DynusT is "allowed" to use in one of its configuration files.

Table III summarizes how the run-time of DynusT for the GTHA network changes using different compiler optimization levels and number of cores. The columns break down each section of a DynusT run into the following components: The "Initial Simulation" time, is for the 0^{th} iteration, i.e. items 1 and 2 of Figure 1. The "Iterative Simulation" column lists how long the iterative simulations (item 4 in Figure 1) took, averaged over all iterations. Likewise, the "Dynamic Assignment" column lists the averaged execution times for assignment in-between simulations (item 4 of Figure 1). The total execution time simply equals initial simulation + number of iterations $\times$ (iterative simulation time + dynamic assignment time). Note that the optimization levels O0, O2, and O3 (O1 was not used as it optimizes for space and its runtime is very similar to O0) and the number of cores are included in the 2^{nd} column next to the operating system used. The first 4 rows help compare the optimization levels O0, O2, and O3, and the remaining rows shown in Table III are all using the O3 setting.

Table III: DynusT execution times on the GTHA network

Row	Exec. Time (hh:mm:sec)	Initial Simulation	Iterative Simulation	Dynamic Assignment	Total (12-iterations)
1	Windows O3 - 8 Cores	00:42:29	00:14:28	00:09:24	05:33:18
2	Linux O0 - Serial	01:50:18	00:11:39	00:48:41	13:54:26
3	Linux O0 - 4 Cores	00:36:39	00:07:35	00:21:12	06:21:54
4	Linux O2 - 4 Cores	00:19:25	00:05:30	00:10:12	03:27:51
5	Linux O3 - Serial	00:44:35	00:07:17	00:20:20	06:16:03
6	Linux O3 - 2 Cores	00:17:17	00:07:10	00:09:37	03:38:43
7	Linux O3 - 3 Cores	00:20:56	00:05:20	00:10:55	03:35:59
8	Linux O3 - 4 Cores	00:17:54	00:05:10	00:09:51	03:18:06
9	Linux O3 - 5 Cores	00:17:28	00:05:36	00:09:48	03:22:10
10	Linux O3 - 6 Cores	00:16:57	00:05:42	00:09:50	03:23:16
11	Linux O3 - 7 Cores	00:16:33	00:05:50	00:09:59	03:26:29
12	Linux O3 - 8 Cores	00:16:27	00:06:06	00:10:01	03:29:52

Comparing the run-times with 4 cores between optimization levels 0, 2, and 3 (rows 3, 4, and 8), there is a drop in execution times, of almost 50% between O0 and O3. The drop is also similar for the serial case when comparing optimization levels O0 with O3 (rows 2 and 5). Although the aggressive optimizations applied with the O3 setting are not guaranteed to speedup run-time due to the heuristic methods they use, they provide better speedup than optimization level O2 here.

Comparing the total run-times with increasing number of cores (1 to 8) with O3, rows 5-12 in Table III, show the expected asymptotic behaviour, reaching a minimum time with 4 cores or a speedup of 1.9x (row 8). However, as the number of cores increases beyond 4, the run-time increases. This is because beyond a certain number of cores, the communication latency between the cores cancels out and then surpasses the benefits of the multi-core speedup. Putting these two results together indicates that the optimal conditions for DynusT's overall speedup are optimization level 3 running on 4 cores.

On another note, the execution time of the initial simulation, goes down to only 16.5 min (2.5x speedup compared to serial) using 8 cores (has less critical sections). This saving in initial simulation time is essential in cases where 10's or 100's of initial simulations are required. For example, determining the optimal toll structure for a regional highway network requires multiple evaluations of dynamically assigning of vehicles into the network in response to the toll structure.

Finally, the run-times on Windows improve with higher optimization levels and an increasing number of cores (not shown in Table III), reaching the lowest times with 8 cores (shown in row 1). However, the total time is 1.7x longer and the initial simulation time is 2.4x longer than the times of Linux O3 - 4 Cores (row 8). This is due to 1. the manual optimizations made on the DynusT code after converting it to run on Linux (discussed in Section III-A), 2. the removal of extra visual DynusT pop-ups that are computationally intensive in Windows and 3. the higher control that Linux provides to multi-threaded programmes (rather than randomly pausing the computation of a core due to various Windows interrupts).

D. Variation with Network Size

In addition to analyzing the performance with compiler optimizations and parallelism on the GTHA network, we have examined DynusT's performance on the smaller GTA network (Figure 6), to determine whether the compiler optimizations and parallelism can provide the same improvements in run-time as the GTHA network. The same analysis as in the previous section was conducted on the GTA network, which is a subset of the GTHA network (see Table I).

Similar to Table III, Table IV shows the execution times for the GTA network. We see a considerable drop in execution time, of almost 50% between O0 and O3 with 4 cores (rows 3 and 8 in Table III). Moreover, we observe a speedup of 1.9x as the number of processors increases from 1 to 4, similar to the GTHA. However, the run-time is lowest for 5 (rather than 4) processors for the GTA, reaching a speedup of 2x. This is because the amount of data being communicated between processors for the GTA network (with 23% fewer vehicles) is less than that in the GTHA network, allowing the usage of 1 more processor before the communication latency cancels out and surpasses the multi-core speedup. This was also observed with a shorter period for writing vehicle info to files in the GTA network (not shown here) between the iterative simulation and dynamic assignment for example compared to the period around the 8-min mark in Figure 9, due to the lower number of vehicles in the network. In addition, the execution time of the initial simulation for the GTA, goes down to only 13 min (2.9x speedup) using 8 cores. This shows that reducing the amount of information being communicated in a parallellised DTA model enables getting further speedup using more cores. Alhough it is desirable to draw general conclusions about run-

Table IV: DynusT execution times on the GTA network

Row	Exec. Time (hh:mm:sec)	Initial Simulation	Iterative Simulation	Dynamic Assignment	Total (12-iterations)
1	Windows O3 - 8 Cores	00:17:00	00:09:14	00:09:40	04:05:51
2	Linux O0 - Serial	01:31:31	00:09:05	00:39:35	11:15:23
3	Linux O0 - 4 Cores	00:31:31	00:06:22	00:20:55	05:58:54
4	Linux O2 - 4 Cores	00:17:39	00:05:21	00:09:18	03:13:27
5	Linux O3 - Serial	00:37:36	00:06:15	00:19:34	05:46:46
6	Linux O3 - 2 Cores	00:14:17	00:06:17	00:09:01	03:17:50
7	Linux O3 - 3 Cores	00:14:05	00:05:49	00:09:05	03:12:53
8	Linux O3 - 4 Cores	00:14:02	00:05:32	00:09:02	03:08:44
9	Linux O3 - 5 Cores	00:13:43	00:04:41	00:09:05	02:58:59
10	Linux O3 - 6 Cores	00:13:13	00:04:54	00:09:26	03:05:13
11	Linux O3 - 7 Cores	00:13:00	00:04:59	00:09:25	03:05:40
12	Linux O3 - 8 Cores	00:13:08	00:05:10	00:09:08	03:04:54

time as a function of network size, it is important to note that the results presented here only reflect the size and the conditions of the GTHA and GTA networks. Therefore, a more thorough analysis using more test cases is required to reach such generalization.

VII. Conclusion

This paper focused on the speedup of DTA simulation models because of the needs to have numerous evaluation runs for optimization and to enable real-time traffic management applications. The methodologies included using both compiler optimizations and parallelism. DynusT as a widely used DTA model was evaluated as a test case, while its results could be generalized because we have used real-networks and calibrated them using real data sets in the Greater Toronto and Hamilton Area (GTHA). Extensive testing was performed to evaluate various dimensions for speed-up including: network size, number of processors, various optimization levels and operating systems. The results showed that compiler optimizations and parallelism allowed the execution time for a 12-iteration simulation run and an initial simulation to be reduced using around 4 cores by around 50% and the 60%, respectively. Moreover, they provided an important insight that speed-up with increasing number of cores is achievable but up to a certain point, depending on the communication latency between the cores. Therefore, reducing the amount of information being communicated in a parallellised DTA model enables getting further speedup using more cores. An added contribution of this work is that the Linux version of DynusT works on more high-performance clusters, and can thus be run using the fastest available processors. Future work includes using the modified DynusT to run a congestion pricing ITS application and a robust DTA algorithm for the GTA network.

ACKNOWLEDGMENTS

This research was funded by the Ontario Research Fund grant on Connected Vehicles and Smart Transportation [22], traffic data was provided by the One-ITS networking platform [23] and the GTA network was developed by Islam Kamel at the University of Toronto.

References

[1] M. Florian, M. Mahut, and N. Tremblay, "Application of a simulation-based dynamic traffic assignment model," *European Journal of Operational Research*, vol. 189, no. 3, pp. 1381 – 1392, 2008.

[2] "Quadstone Paramics: Traffic and Pedestrian Simulation, Analysis and Design Software," http://www.paramics-online.com/.

[3] "AIMSUN: Traffic Modeling Without Boundaries," http://www.aimsun.com/wp/.

[4] "VisSim: The smarter, faster way for model-based system development," http://www.vissim.com/.

[5] N. Taylor, "The contram dynamic traffic assignment model," *Networks and Spatial Economics*, vol. 3, no. 3, pp. 297–322, 2003.

[6] H. Mahmassani, A. A. N. Huynh, X. Zhou, Y. Chiu, and K. Abdelghany, "Dynasmart-p (version 0.926) user's guide," Technical Report STO67-85-PIII, Center for Transportation Research, University of Texas at Austin, Tech. Rep., 2001.

[7] "DynusT: Dynamic Urban System in Transportation," http://dynust.net/wikibin/doku.php/.

[8] S. Peeta and A. Ziliaskopoulos, "Foundations of dynamic traffic assignment: The past, the present and the future," *Networks and Spatial Economics*, vol. 1, no. 3-4, pp. 233–265, 2001.

[9] Y. Wen, "Scalability of dynamic traffic assignment," Ph.D. dissertation, Massachusetts Institute of Technology, Boston, USA, 2008.

[10] R. B. et. al., *Dynamic Traffic Assignment, A Primer*, Transportation Research Board, 500 Fifth Street, NW Washington, DC 20001, 2011.

[11] H. Aydt, Y. Xu, M. Lees, and A. Knoll, "A multi-threaded execution model for the agent-based semsim traffic simulation," in *AsiaSim 2013*, ser. Communications in Computer and Information Science, 2013, vol. 402, pp. 1–12.

[12] Y. Xu, G. Tan, X. Li, and X. Song, "Mesoscopic traffic simulation on cpu/gpu," in *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, 2014, pp. 39–50.

[13] Y.-C. Chiu, J. Villalobos, and P. Mirchandani, "Numerical performance of the spatially and temporally scalable dynamic traffic simulation and assignment system malta," in *Transportation Research Board 87th Annual Meeting*, 2009.

[14] H. Abdelgawad and B. Abdulhai, "Managing large-scale multimodal emergency evacuations," *Journal of Transportation Safety & Security*, vol. 2, no. 2, pp. 122–151, 2010.

[15] M. Kutz, *Handbook of transportation engineering*. McGraw-Hill, 2011.

[16] Y.-C. Chiu, L. Zhou, and H. Song, "Development and calibration of the anisotropic mesoscopic simulation model for uninterrupted flow facilities," *Transportation Research Part B: Methodological*, vol. 44, no. 1, pp. 152 – 174, 2010.

[17] R. Kumar and P. Singh, "An approach for compiler optimization to exploit instruction level parallelism," in *Advanced Computing, Networking and Informatics- Volume 2*, ser. Smart Innovation, Systems and Technologies, 2014, vol. 28, pp. 509–516.

[18] R. Kirner and W. Haas, "Optimizing compilation with preservation of structural code coverage metrics to support software testing," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 184–218, 2014.

[19] "Quick-reference guide to optimization with intel compilers version 12," University of Toronto SciNet support guides, Tech. Rep., 2010.

[20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, Sunnyvale, California, 1967.

[21] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov 2010, pp. 159–168.

[22] "Connected Vehicles and Smart Transportation," http://cvstproject.com.

[23] "The One-ITS Networking Platform," www.one-its.net.