Parallel Vehicular Traffic Simulation using Reverse Computation-based Optimistic Execution

Srikanth B. Yoginath and Kalyan S. Perumalla
yoginathsb@ornl.gov, perumallaks@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Abstract

Vehicular traffic simulations are useful in applications such as emergency management and homeland security planning tools. High speed of traffic simulations translates directly to speed of response and level of resilience in those applications. Here, a parallel traffic simulation approach is presented that is aimed at reducing the time for simulating emergency vehicular traffic scenarios. Three unique aspects of this effort are: (1) exploration of optimistic simulation applied to vehicular traffic simulation (2) addressing reverse computation challenges specific to optimistic vehicular traffic simulation (3) achieving absolute (as opposed to self-relative) speedup with a sequential speed equal to that of a fast, de facto standard sequential simulator for emergency traffic. The design and development of the parallel simulation system is presented, along with a performance study that demonstrates excellent sequential performance as well as parallel performance.

1   Introduction

In applications such as emergency or evacuation planning[1, 2], a higher speed of simulation for traffic models translates into faster determination of critical metrics such as expected evacuation time. While sequential simulators exist for traffic simulation, scalable parallel simulations are few. Among the existing parallel traffic simulators, parallelism is realized either by functional parallelism[3] (parallelizing the steps such as trip planning, configuration generation, partitioning, etc.), or by synchronous parallel execution (time-stepped models)[4-7], or both. Our focus is to combine the speed of discrete event models with parallel execution of the actual simulation runtime. Additionally, despite speed concerns, our efforts are in staying at higher fidelity with entity-level models, as opposed to resorting to aggregate techniques such as fluid or network flow models.

With advances in parallel discrete event simulation modeling and the availability of multi-processor hardware, high-speed simulation of high fidelity models is appearing to be possible now. Based on this premise, we have designed and developed a parallel vehicular traffic simulation model called SCATTER-OPT, standing for an optimistic-parallel version of the SCATTER simulation system[8]. This simulator is capable of using either conservative or optimistic synchronization when executed on parallel platforms. Importantly, the parallel execution speedup achieved by this simulator is over and above some of the best performance achievable sequentially by vehicular traffic simulators today. The performance gains reported here, thus, are absolute and not simply self-relative.

In the rest of the article, the design, development and performance of SCATTER-OPT is described. This is started in Section 2 with the design overview of the transportation model, documenting the assumptions made in modeling the various components of the road network system. This is followed by a brief discussion on our parallelization approach. Then, we dwell into the specifics of the parallel simulation model discussing the realization of various vehicular traffic system characteristics like, congestion, routing etc. In Section 3, the reverse computing algorithms are discussed that are used for realizing optimistic synchronization. Section 4 is dedicated to a performance study, where first the sequential simulation runtime performance is compared with that of a de facto standard emergency management system called the Oak Ridge Emergency Management System (OREMS) [1, 9]. This is followed by a parallel performance study of SCATTER-OPT, evaluating both optimistic and conservative modes of synchronization for parallel execution. Section 5 summarizes our current work and alludes to future directions.

2   SCATTER-OPT System

Our simulation model is implemented in a simulator called SCATTER-OPT, which includes a discrete event model, and a parallel execution

framework for vehicular networks. Parallel discrete event simulation of vehicular network operation involves the development of a discrete-event model of the system coupled with a careful partitioning of the model on to multiple processors for optimum run-time performance. In our simulation model, the road network is modeled as a graph, in which road segments connect intersections. Each road segment is modeled with a few physical attributes (number of lanes and length of road segment), kinetic specifications (speed limit) and controllers (traffic lights). The traffic lights are synchronized in their operation and have a fixed period of GREEN time (when the vehicles are allowed to enter the road-segment) and a fixed period of RED time (when the vehicles are not allowed to enter). Traffic lights control the entry of vehicles for every road segment of the simulated road network.

An input file is used to specify parameters such as the GREEN time period and the RED time period, along with an initial-offset (i.e. the wait time to the first GREEN period at simulation start time). Each road-segment in the simulated transportation network contains this information.

An intersection, a point at which the road segments connect to each other, is considered as an indivisible, independent processing unit for parallel computing purposes. Every intersection is capable of generating new vehicle instances that are distinguished from each other by their unique identifiers. Each generated vehicle has its physical attributes (e.g., length of vehicle) and kinetic attributes (e.g., travel velocity and acceleration), in addition to its source and destination intersection information for its current trip. Every intersection in the road-network is capable of routing vehicles to their next hop toward their individual destinations.

A *VehicleEvent (VE)* signifies the arrival of a vehicle from one intersection to another. Events of this type are processed by each intersection to generate additional *VEs* that act as arrival events on this intersection's neighboring intersections. The dynamically chosen neighboring intersection is the next hop node toward the destination of the vehicle contained in a *VE*.

Information is provided in an input file about the road network layout (that encompasses intersection co-ordinates, connecting road-segment characteristics), traffic light information, source and destination intersections, traffic generation rate etc. This information is used to set up and initialize the simulation process.

2.1 Parallel Decomposition

We use a simulation library to realize both conservative and reverse-computing based optimistic parallel discrete event simulation model. The library is a general-purpose parallel/distributed simulation kernel, which provides programming interfaces to develop models that could run on one or more machines. During parallel simulation, one instance of the application executable per processor is run on requested number of processors. Each processor can host multiple simulation logical processes. Logical processes are autonomous in the sense they hold and manage their own events and can be optimistic or conservative in their event processing. Efficient communication and virtual time synchronization are also provided across processors in shared and/or distributed memory platforms.

A challenge involved in any parallel discrete event model involves the issue of how to split the discrete event road-network model among the parallel processors. An efficient partitioning of the application process across different processors during parallel computation fetches better performance. Efficient partitioning involves recognition of modules of an application process that could run concurrently, with minimal, infrequent interaction between each other. In particular, zero lookahead interaction is to be avoided for parallel simulation efficiency.

Every lane in road-segment is a spatial resource that is occupied by vehicles. The vehicles infringe the road-segment lane space based on their physical dimensions. A traffic light for every road-segment controls the periodic entry of the vehicle into the corresponding road-segment lane. By having a traffic light for every road-segment, the periodic entry of vehicles into each road-segment lane is ensured.
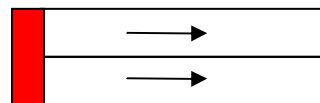

Figure 1: Outgoing road-segment with 2 lanes and a traffic signal component

We make an intersection (node) in the input road-network graph as a logical process. This would require grouping the road-segments connecting to each other via an intersection. There is a choice on how the segments are mapped relative to their intersection(s). Either incoming road-segments or out-going road-segment can be included as part of an intersection logical process. The former approach, namely, incoming segments mapped to their destination intersections, is used in the SCATTER tool. We use

the alternative approach in SCATTER-OPT, namely, outgoing segments mapped to same logical process as their source intersection. Thus, in SCATTER-OPT we consider the node of the road-network graph along with its outgoing road-segments as a logical process. Each intersection in the transportation model is a collection of outgoing segments plus the actual intersection space itself.
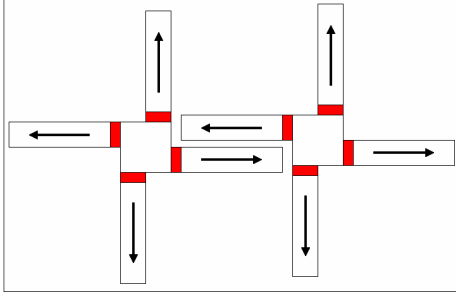

Figure 2: The signal controls the entry of vehicles into the road-segment

2.2 System Implementation

Traffic Generation

The SCATTER-OPT input file contains the information of the rate at which vehicles traveling to a particular destination to be generated. Note that a rate, rather than individual vehicle identity, is specified in the file. This is for convenience only; each vehicle is individually represented and simulated autonomously (e.g., a single vehicle could get stalled at a traffic light), and no aggregation is performed. Note also that the rate is on a per flow-basis, the rate can be different across different flows. Any desired traffic pattern can thus be generated in general.

A *CreateEvent* (*CE*) type is used for the purpose of creating a vehicle at each intersection based on the rate specified in the input file. A *CE*, when processed, creates a *VE* that is rescheduled on the same intersection, and also creates another *CE* that is scheduled to some random time *dt* in future, to continue chaining generation of vehicles at specified input rate. The processing of *VE* by an intersection remains same, regardless of its origin.

CreateEvent:
- *Create VE at this intersection*
- *Schedule it to the same intersection at (t + lookahead)*
- *Create and schedule CE at (t +dt) to this intersection*

Figure 3: *CreateEvent* processing block

Routing

After generation of a vehicle, each intersection ensures that a "vehicle-following" scheme is correctly followed. A vehicle-following scheme ensures vehicles follow first-in-first-out (FIFO) scheme while following their individual speeds and accelerations to the farthest possible distance. Routing of each vehicle toward its destination is based on Dijkstra's shortest path algorithm. The Boost's Graph Library (BGL) is used for the purpose of computing shortest paths. The road-network of the input file is converted into a BGL graph. For every node the shortest path to every other node in the network is determined, from which a vector of parents for each node is obtained. Each intersection uses this vector recursively to determine the next hop intersection of the vehicle toward its destination. Based on the next hop, the intersection determines out-going road-segment on which the incoming vehicle is routed.

Whenever a vehicle enters the intersection, the departure time (t_d) is calculated using transit time (t_t) and the vehicle's wait time at the traffic signal (t_w):
$$t_d = t_t + t_w.$$

(a) *Transit time* (t_t):

Transit time is calculated by solving the quadratic equation from the Newton's law of motion for time *t*. It takes the velocity and acceleration of the given vehicle into consideration, $S = ut + \frac{1}{2}at^2$, where, *S* is the distance traveled (road-segment length), *u* is the velocity of the vehicle; *a* is the acceleration of the vehicle, and *t* is the time to traverse the road-segment length. This value for departure time is retained only if the calculated value is greater than the departure time of the vehicle ahead of this vehicle. The departure time of the vehicle ahead of this could be obtained by looking into the *EventHistory* list held by the road-segment connecting to the vehicle's next-hop intersection.

This transit time has to be reconciled with the vehicle-following constraints, namely, slow the vehicle down if one or more vehicles ahead of this vehicle prevent this vehicle from reaching at the computed transit time. A sum of departure time of the vehicle ahead (t_{dva}) and time needed to cover a distance of vehicle length (*vlen*) at a speed specified by speed limit (*s*), is used as the departure time to ensure the vehicle following. $t_d = t_{dva} + (vlen/s)$.

(b) *Wait time at the Traffic signal* (t_w)

As mentioned earlier every road-segment has GREEN time period during which it allows the flow of traffic through it and RED time period during which no vehicle is allowed to enter the road-segment. At any

given simulation time, every intersection is capable of knowing the GREEN time period or RED time period of any road-segments in the road-network and hence the wait period on traffic signal for a particular vehicle is calculated determining when it enters any road-segment. The departure time for every vehicle is then calculated as: $t_d = t_t + t_w$.

Each road-segment lane maintains a list named *EventHistory* that keeps track of the *arrival_time*, *departure_time* and *vehicle_id* of every vehicle that entered and left the intersection.

```
VehicleEvent:
If(enough space to hold veh in road-segment)
{
  - compute transit time (tt)
  - compute wait time (tw)
  - calculate departure time (td)
  - send vehicle to next hop intersection at td
  - record the event in EventHistory
}
```

Figure 4: *VehicleEvent* processing block

Perfect Reversibility Considerations

In what follows, an overriding consideration behind the modeling approaches and consequential data structures is to enable perfectly reversible computation of state changes.

Congestion

As mentioned earlier every road-segment keeps track of incoming and out-going vehicle events in an *EventList*. This is referred by the intersection to determine the occupancy of any of its road-segments at a given time. On vehicle arrival, the intersection looks into the occupancy of the road-segment that connects to the vehicle's next-hop intersection. The vehicle is scheduled to depart the intersection only if enough space in the road-segment is ascertained to exist at that point of time. On the other hand if the associated road-segment is completely occupied, then a congestion behavior in the road-network is emulated. To realize this, the vehicle is held in the current intersection as long as enough space is available in the corresponding out-going road-segment. The additional time delay (time spent in waiting for availability of enough space) that the vehicle experiences in the intersection before being scheduled for departure directly corresponds to the congestion in the road network. In the following paragraphs we discuss different approaches that we have implemented to realize this behavior.

(a) *Simple rescheduling*

In this approach, on detecting non-availability of space in the road-segment the arriving vehicle is re-scheduled to the same road-segment at the next GREEN time period (when this road-segment allows vehicle to pass through it). With this simple technique the vehicle is (logically) retained in the intersection as long as there is enough space in the outgoing road segment to accommodate it, also ensuring that the vehicle leaves the intersection in the order of its arrival.

This technique works well for minor congestion in the transportation network. However, if the arrival rate of the vehicles is far greater than the departure rate, then the simulation crawls toward its end. This is due to the need for regular polling required for rescheduling the departures; as a result of this polling, every vehicle suffers multiple reschedules on a single intersection, this in-turn gives rise to equivalent number of events.

(b) *Rescheduling with initial time adjustment*

Needless polling and rescheduling of *VEs* degrades the performance in the previous strategy. If it were possible to ascertain the number of vehicles "in limbo" that are ahead of any new arriving vehicle, we can calculate the time taken for that many vehicles to depart and schedule the vehicle arrival at that time in future. To this end, we maintain a variable named *nlimbo* that keeps track of the number of limbo vehicles in an intersection that have logically moved on beyond the road-segment. This variable is used to calculate the new departure time as $t_{new\_d} = \text{nlimbo} \times (vlen / s) + t_d$, where t_{new_d} is the new departure time calculated; $vlen$ is the average vehicle length; s, is the speed limit on the road-segment and t_d is the departure time calculated based on *transit time* and *wait time*.

However, the *nlimbo* variable could not be used for rescheduling the already rescheduled events, since with this variable the order of arrival of vehicles is not taken into consideration. Hence, the time calculated using the *nlimbo* variable could be used only to schedule the newly arrived vehicles from peer intersections and not to reschedule the vehicles withheld previously due to congestion. While we could get better speed-up pushing the initial re-scheduling time of the arriving vehicle much farther in future, the later schedules of the same vehicle would be of constant time; hence this suffers from the same problem of the former model. Better run-time performance could be achieved for smaller periods of congestion, but for longer periods of congestion the

run-time performance would be similar to the previous model.

(c) *Using a "limbo list"*

The best performance is obtained by keeping track of more information, namely, by maintaining a list (*limbo-list*) instead of a single *nlimbo* variable. Using this scheme, we were able obtain a faster simulation. The FIFO *limbo-list* preserves the order of arrival and obviates rescheduling congested vehicle events on the same intersection. Whenever a new vehicle arrives at the intersection, it is put into the *limbo-list*. The intersection ensures that enough space in the outgoing road segment is available before removing the vehicle from the *limbo-list*. If enough space is available, the intersection schedules the vehicle at the end of the *limbo-list* to ensure the FIFO order in traffic flow.

```
VehicleEvent:
- Insert the incoming vehicle in limbo-list
While (limbo-list not empty)
{
  if (space for veh in road-segment){
    - get the first veh from limbo-list
    - compute transit time(tt)
    - compute wait time (tw).
    - calculate departure time (td)
    - send VE to next hop intersection at td
    - record the event in EventHistory
  } else{
    - create a SelfUpdateEvent (SE)
    - schedule SE at tx
    - break from loop
  }
}
```

Figure 5: Altered *VehicleEvent* processing algorithm to accommodate congestion behavior

If enough space to hold the vehicle is not available on the outgoing road segment at that point of time, the vehicle is retained in the *limbo-list*. The removal of the vehicle from the *limbo-list* should be dependent on the availability of space on the out-going road-segment. To ensure this, a *SelfUpdateEvent* (*SE*) is scheduled to a future time, when enough space to hold the vehicle in the outgoing road segment lane is available. This future time, t_x, can be approximated to the departure time of the second vehicle at the farthest end of the corresponding outgoing road segment lane, which ensures spatial availability to hold at least one vehicle in the road-segment lane.

Thus, by maintaining the additional *limbo-list* and with *SE*, we were able to emulate congestion in the road-network and as expected, better runtime performance was also observed. Figure 5 gives the altered algorithm to process *VE*, to accommodate modeling the congestion behavior in the road-network. Absence of the first step, i.e. the insertion of vehicle into *limbo-list*, is the only change in the algorithm used to process *SE* arrival.

Memory management

As described above each road-segment maintains a list to keep track of the events arrived and departed from that road-segment, we refer to this list as the *EventHistory*. As the simulation progresses with time, the length of the *EventHistory* grows. This not only increases the memory requirement for the simulation run but also degrades the performance, since the list is used for calculating "number of vehicles in road-segment" that results in a search with $O(n)$ complexity. Hence, constant cleanup of the *EventHistory* is needed to overcome the memory and performance inefficiencies. To ensure safe cleanup, only the elements with vehicle arrival time less than the safe global time (Lower Bound on Time Stamp – LBTS) are purged.

(a) *Using a periodic timer*

If we were to use a timer that would schedule an event periodically to clear the *EventHistory*, it would solve the performance as well as memory problem. But, events for clearing the *EventHistory* will be generated even when the event history is not big enough. Also an inactive intersection (to which vehicles haven't arrived yet) ends up needlessly performing this clearing operation. Hence, we generate and process reclaimable events while using a timer; and this in-turn affects the performance of the simulation.

(b) *Keeping track of length of the EventHistory*

Another strategy is to keep track of the length of *EventHistory* to initiate the cleanup process. This strategy works fine but poses a problem when the vehicle arrival rate is very high. In this case, each arrival event looks into the length of *EventHistory* and schedules a cleanup event. If a check on previously initiated clean-up process is not made before initiating a new process, redundant events are created and processed. This would go on until the point where the very first arrival actually completes the cleanup process and updates the length of the *EventHistory*.

This problem of redundant event generation is overcome by making the scheduling of the clean-up process mutually exclusive, *i.e.*, if an event has already scheduled a cleanup process, no other event would schedule one, until the initiated one completes. By doing so, we reduce many redundant events thus enhancing the runtime performance of the simulation

model. The cleanup process of *EventHistory* in SCATTER-OPT is realized through an event type called *FlushEvent*.

3 Synchronization

Broadly speaking, multiple synchronization techniques exist to perform parallel execution, namely, conservative and optimistic. SCATTER-OPT is currently tested to work in both conservative and optimistic synchronization modes. Reverse computation technique is used to realize rollback in optimistic synchronization.

3.1 Reverse Computation (RC)

Optimistic federates differ from their conservative counterparts in that they do not discard events after processing them. Instead they keep the events around, and also maintain copies of simulation states before modifying them as part of event processing. Since optimistic federates do not rely on lookahead, they execute their events without blocking for safety. Thus a federate will have to roll back its computation if/when it later receives events whose timestamp is less than its current simulation time. There are two main parts to such rollback (1) undo local computation by restoring the state prior to erroneous event processing (2) undo all events erroneously sent to other federates. While the parallel simulation library performs these rollbacks at the library level, reversal code to restore the application data structure state, in the event of rollback, needs to be written and provided by the application.

3.2 RC Algorithm

To recap from previous section, four events are used to realize the transportation model. They are: *CreateEvent (CE)*, *VehicleEvent (VE)*, *SelfUpdateEvent (SE)* and *FlushEvent (FE)*. The *FEs* are utilized for optimization purposes in a safe manner (reclaiming memory that strictly belongs to past that cannot be rolled back) and hence do not impact the correctness of simulation, hence the reversal of *FE* can be ignored. The *CE* is used to generate the traffic at the specified rate. The only state variable they alter is the *VehicleID* counter, which is incremented as vehicles are generated. Rollback of a *CE* involves decrementing this counter. In the experimental road network that we have considered, source intersections do not have any incoming events from any other intersections, thus eliminating the possibility of causality error occurrences. Hence, in this experimental setup *CE* reversal code is never utilized.

The *VE* and the *SE* events are responsible for routing the vehicles from one intersection to another. Doing so, they alter the states of *EventHistory* and

limbo-list data-structures at the application level. Hence, during rollback, reversal of the data structures to their previous states is necessary for correct rollback. Further, as discussed earlier, both *VE* and *SE* processing use the same algorithm; the only difference being that the former inserts the arriving vehicle into the *limbo-list*.

In the following, the reversal procedures for *VE* and *SE* processing are similar, unless otherwise noted.

Each VE_i may generate one or many VE_{ij}, (where $j=1,2,3,...$) and, may or may not create an *SE*, based on the congestion in the network. Hence, for reversal we should first find out if *VEs* were generated and if so, how many were generated.

```
VehicleEvent (VE) /SelfUpdateEvent(SE)
If(VEs generated)
{
  loop(Number of generated VEs){
    - remove event-record from EventHistory
    - insert Veh into limbo-list (from front)
  }
  if (this event is VE){
    - remove last Veh. from limbo-list
  }
}
```

Figure 6: Algorithm for application level *VehicleEvent* and *SelfUpdateEvent* reversal

We know that when an intersection generates an out-going *VE*, it records that event in the *EventHistory* list. Hence, after finding the number of *VEs* generated, we need to remove the record for each event generated in the *EventHistory* list. Further, the vehicle object in that *VE* should be extracted and pushed back into *limbo-list* from the end through which it was removed. While, the *SE* reversal procedure completes here, the *VE* reversal goes a step further and removes only the last vehicle that was inserted in the *limbo-list* to complete its reversal procedure. This takes back the intersection road segment to the correct state prior to the processing of this *VE*.

After arriving at the reversal algorithm, implementation was found to be a challenge with the reverse execution interface. The reverse computation algorithm in the previous sub-section needs to identify the events that generated specific events and extract objects from the generated events. Modifications to the library had to be made to expose the event's causal list data structure, so that the application could iterate through the copy of events sent, to find the necessary events.

4   Performance Study

4.1   Sequential Performance

Experimental Setup

In this section, we refer to road networks to be grids of size $N \times N$. The experimental setup contains $N \times N$ intersections, with $2 \times N$ sources injecting traffic from either sides (right and left), to 8 destinations equally distributed on the top and bottom ends of the grid. Hence, our $N \times N$ grid scenario consists of $(N \times N)+(2 \times N)+8$ intersections.   Figure 7 shows a 10×10 road network grid, containing 10 sources on left and right, and 4 destinations at top and bottom, in addition to 100 interior intersections.
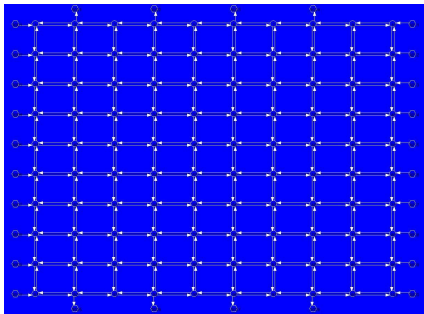
Figure 7: 10×10 road-network grid (128 intersections) snapshot of OREMS graphical interface

Also, the traffic lights are specified with an 8-second cycle time. This cycle time is equally shared between GREEN and RED time periods.   The intersection transit time is set to 1 second.

SCATTER-OPT, tested sequentially for getting an idea of its raw sequential speed. To make sure it is close to the best sequential performance available today, it is compared with OREMS (Evacuation Modeling System). OREMS is an aggregate, *fluid-based* model, very fast in computing and is used in actual emergency operations.

Road networks of dimension 10×10, 12×12, 14×14 and 16×16 were used for comparison. The intersections are connected to their neighbors through single-lane road-segments of length 1600 meters (1 mile). The lengths of the road-segments are same throughout the test network, except for the road segments connecting the sources and sinks, which are of length 10 meters. For each road network, sources generate traffic at a rate r vehicles/hour/destination, where, r = 400, 500, 600, 800 and 900. Larger grid sizes of the network were not considered for study since OREMS input format prevents it from executing beyond a 16×16-sized grid.

Hardware

The sequential performance comparison of SCATTER-OPT with OREMS was carried out on an Intel® Core™2 Duo CPU T7700 at 2.4 GHz, with 2GB of memory running Microsoft Windows XP Professional SP2.

Benchmarks

Simulation runtime in *seconds* to evacuate traffic generated at rates as specified in the experimental setup is plotted against *number of vehicles evacuated* for both OREMS and SCATTER-OPT.
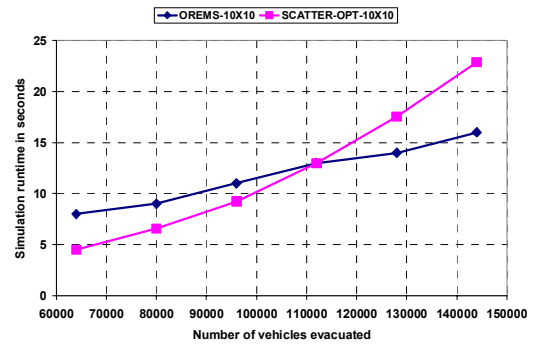

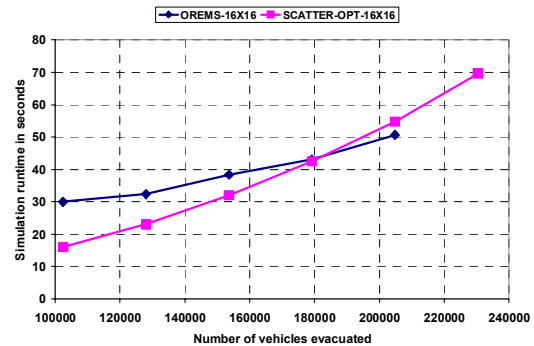Figure 8(a): Simulation runtime of OREMS and SCATTER-OPT against number of vehicles evacuated plot, for 10×10 grid

Figure 8(b): Simulation runtime of OREMS and SCATTER-OPT against number of vehicles evacuated plot, for 16×16 grid

As seen from figure 8, in both 10×10 and 16×16 networks, the discrete event-based SCATTER-OPT model runtime is smaller than that of OREMS at lower input traffic rate, but SCATTER-OPT' runtime slowly overtakes OREMS' as the input traffic rate increases. Similar pattern is observed for grid sizes 12×12 and 14×14. However, the increase in simulation time of SCATTER-OPT on larger networks is negligible when compared to runtimes of equivalent high-fidelity

(vehicle-level) simulators. In separate experiments, we benchmarked the same networks with MITSIM and TRANSIMS and obtained runtimes that were at least one order of magnitude higher (i.e., simulations were 10× slower than OREMS and SCATTER-OPT).

4.2   Parallel Performance

Absolute Speedup

To demonstrate the absolute speedup of SCATTER-OPT, we present the simulation runtime comparison of OREMS and SCATTER-OPT (with one and two processors) on the same 16×16 road network scenario considered for sequential runs in Figure 9.
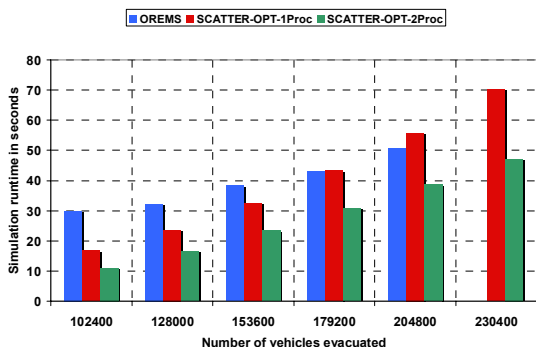
Figure 9: Simulation runtime of OREMS and SCATTER-OPT (using one and two processors) against number of vehicles evacuated, for 16×16 grid

The serial and parallel SCATTER-OPT runs were carried out again on the same hardware, but running Mac OS X 10.4.11 operating system. Note that the runtimes of single processor runs carried out on Mac OS X in figure 9 closely correspond to the one taken on Windows XP in figure 8(b). The reduction in the simulation runtime with the increase in the number of processors, demonstrate the absolute gain in speedup of SCATTER-OPT.

Larger Road-network Scenario

Previously, we considered a constant time of 1 second as the time required for a vehicle to cross any intersection (the space that connects road-segments). This is used as the minimum lookahead time period in conservative synchronization and this time period is constant across all intersections. Realistically, the transit time to cross the intersection space could be either lesser than or greater than 1 second, but is never constant. In conservative mode, if the lookahead is large, it is good for the runtime performance since the simulation can take long strides. On the other hand, smaller values of lookahead could make the simulation run slower. The simulation performance would be worsened for a broad lookahead range, since the minima of these lookahead values is taken into consideration while calculating the global virtual time. With the vehicular traffic modeling, we are bound to get a range of lookahead values. Hence, the study of the performance of the simulation model with decrease in lookahead becomes significant for both conservative and optimistic synchronization based models.
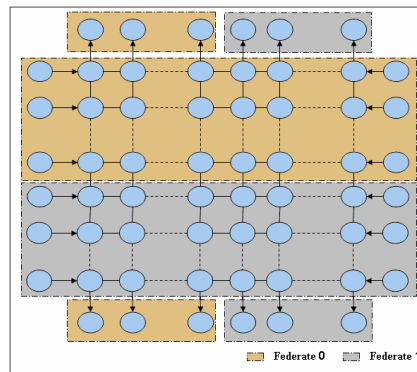
Figure 10: Distribution of intersections across processors or federates

Experimental Setup

The experimental setup contains $N \times N$ intersections, with $2 \times N$ sources injecting traffic from either sides (right and left), to $2 \times N$ destinations equally distributed on the top and bottom ends of the grid. Hence, our $N \times N$ grid scenario consists of $(N \times N) + (2 \times N) + (2 \times N)$ intersections. For the optimistic and conservative performance comparison purposes, we have considered a 64×64 network grid, with 128 sources generating traffic at 400 vehicles/hour/destination rate, toward 128 destinations.

Event Load Distribution

The input network grid is divided into blocks of rows, and intersections (including the sources), falling in each block run on one federate, the destination intersections are equally divided among all federates. For example: a 64×64 road-network grid, when divided among two federates, each federate gets 64×32 intersections and 64 sources (32 left and 32 right) and 64 (32 top and 32 bottom) sinks. Figure 10, shows the distribution of intersections among two federates.

Road-segment Length Fixation

It was observed as the distance between the intersections increases, the number of rollbacks reduced significantly. Further, no rollbacks were recorded when the road-segment length was 1600 meters. Hence, to study the effect of reverse computing on the system, the distance between all the intersections was fixed to 1000 meters (1 kilometer).

Hardware

The parallel runs to study the conservative and optimistic runtime performances were carried out on Oak Ridge Institutional Clusters (OIC). The OIC cluster consists of a unique bladed architecture from *Ciara Technologies* called *VXRACK*. The *VXRACK* contains 80 usable nodes. Each node has *Dual Intel® 3.4GHz Xeon EM64T* processors, *4GB* of memory and dual *Gigabit Ethernet* interconnects. All nodes run *Red Hat Linux Enterprise WS v4* operating system.

Benchmarks

Here, we discuss the performance of the model that simulates the evacuation of around 6.5 million vehicles generated from 128 sources, through 4096 intersections toward 128 destinations. Figure 11(a) and 11(b) presents observed simulation runtimes (in hours) for parallel runs across number of processors used. The two curves seen in these figures pertain to the runtimes of the parallel runs using conservative and optimistic synchronization.

With a lookahead of 1second, a higher runtime in the model with optimistic synchronization is seen and the conservative mode performs better than optimistic. The degradation in the performance of the model using optimistic synchronization is attributed to the reversals. The performance gain expected due to optimistic synchronization is lost due to the higher number (of order 10^6) of reversals. However, with 32 processors the runtime of the optimistic curve significantly recedes due to very small reversal counts. The bar graph in figure11(c) shows the reversal counts recorded for simulation runs with varying number of processors. Note that the undo counts plotted here correspond to *VE* and *SE* reversals only. The *FE* reversal-count is not considered, since no code is invoked for its reversal.

As we reduce the lookahead by a factor of 10, the simulation runtime curve for conservative mode starts quickly increasing with increase in number of processors; this can be attributed to the frequent synchronization requirement. With no reversals during optimistic synchronization the simulation runtime decreases with increase in the number of processors.

From figure 11(b), we see that with a lookahead of 0.1, the 16 processors simulation runtime for optimistic mode is around 14 hours; the corresponding conservative mode value is around 18 hours that increased from its lowest of 16.5 hours with 8 processors. Hence, using optimistic mode a drop of around 2.5 hours (15%) is achieved using 16 processors over the best simulation conservative mode runtime, when the lookahead is 0.1. Similar

observation with a lookahead of 0.1 can also be made in 128×128 grid scenario that models the evacuation of around 6.5 million vehicles from 256 sources toward 256 destinations through 16,384 intersections, as shown in figure 12. Thus, in modeling vehicular traffic network, where the lookahead is not fixed, optimistic synchronization (reverse-computing) provides a better promise for timely simulation results.
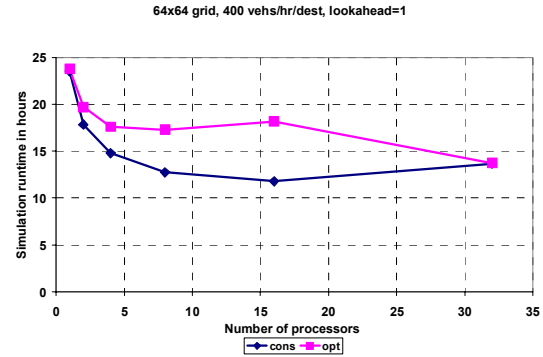

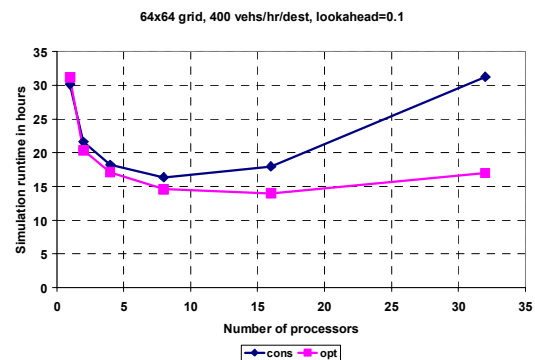Figure 11(a): Simulation runtime against number of processors, with lookahead 1

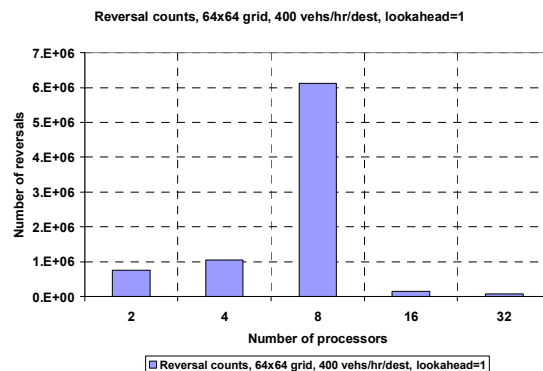Figure 11(b): Simulation runtime against number of processors, with lookahead 0.1

Figure 11(c): Number of reversals against number of processors, in 64×64 grid when the lookahead is 1

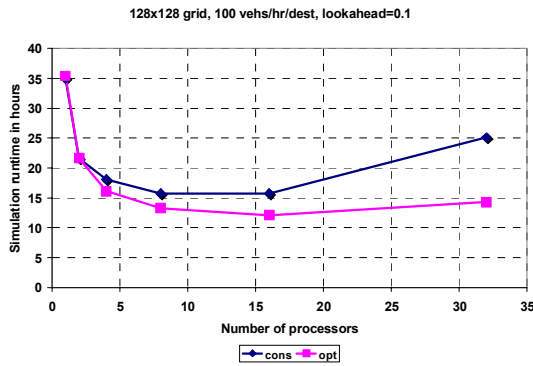**128x128 grid, 100 vehs/hr/dest, lookahead=0.1**

Figure 12: Simulation runtime against number of processors, with lookahead of 0.1 on a 128×128 grid

The simulation results in both optimistic and conservative execution have been verified to be correct and evacuation time results are repeatable, for both sequential and parallel runs (evacuation time shows negligible variation among runs with varying number of processors, varying flush event periods, and so on).

5 Summary and Conclusion

In this paper, we discussed the design, development and performance of a parallel discrete event vehicular traffic simulation model. We ensured its sequential performance compares to the best of the available transportation model (OREMS). We incorporated the reverse-computing based optimistic synchronization for the PDES model. We compared the parallel performance of models using optimistic and conservative synchronization techniques fixing the input traffic generation rate and varying lookahead values. To our knowledge, this is the first attempt at applying optimistic simulation techniques to parallel vehicular network simulation. The perfectly reversible formulation of the model is also novel that enables reverse computation. Additionally, the performance improvement is challenging due to the requirement of low parallel computation overhead needed to compare favorably with an extant, fast sequential simulator (OREMS). In that vein, the absolute speedup (i.e., speedup compared to OREMS), rather than self-relative speedup (speedup compared to SCATTER-OPT on 1-processor) is an additional strength.

5.1 Future Work

While current implementation is limited to a single lane per direction per road segment, support for multiple lanes needs to be added. Performance on generalized networks remains to be evaluated, although we expect the challenges to only lie in optimizing intersection-to-processor mapping for performance; the

software is capable of delivering correct results with any arbitrary assignment. Performance of rollback using reverse computation could be compared to that of state saving.

An important direction we are currently pursuing is in testing the system with much larger networks than are presently used in existing systems such as OREMS. The current limitation is due to the input network format of OREMS, which limits the size of the networks to be with in 100×100. The data structures and the parallel simulation library do not have any non-scalable components as such; hence we believe it will scale well to larger networks (e.g., 1000×1000).

Acknowledgements

References

1. Franzese, O. and L. Han. *A Methodology for the Assessment of Traffic Management Strategies for Large-scale Emergency Evacuations*. in *11th Annual Meeting of ITS America*. 2001.
2. Perumalla, K.S. and B. Bhaduri. *On Accounting for the Interplay of Kinetic and Non-kinetic Aspects in Population Mobility Models*. in *European Modeling and Simulation Symposium*. 2006.
3. Fisher, K.M., *Transims is Coming!* Public Roads, 2000. **63**(5): p. 49-51.
4. Laboratory, L.A.N. *TRANSIMS*. 2001; http://transims.tsasa.lanl.gov/.
5. Meister, K., et al. *A Comprehensive Scheduler for a Large-scale Multi-agent Transportation Simulation*. in *International Conference on Travel Behaviour Research*. 2006.
6. Innovative Transportation Concepts, I. *VISSIM Simulation Tool*. 2001; http://www.itc-world.com/VISSIMinfo.htm.
7. Cameron, G.D.B. and G.I.D. Duncan, *PARAMICS, Parallel Microscopic Simulation of Road Traffic.* Journal of Supercomputing, 1996. **10**(1): p. 25-53.
8. Perumalla, K.S. *A Systems Approach to Scalable Transportation Network Modeling*. in *Winter Simulation Conference*. 2006. IEEE.
9. Bhaduri, B., C. Liu, and O. Franzese. *Oak Ridge Evacuation Modeling System (OREMS): A PC-Based Computer Tool for Emergency Evacuation Planning*. in *Symposium on GIS for Transportation*. 2006.