ATS – An Autonomous Traffic Simulator

Bryant Pong, Derek Schultz, Matt Hancock

Rensselaer Polytechnic Institute

CSCI-4320 – Parallel Programming

***Abstract***: **In today’s world, autonomous cars are becoming more of a reality with each passing day. The prime example of such a vehicle is of Google’s autonomous car. Supporters of autonomous vehicles envision a future in which every day driving hassles, such as traffic jams and car accidents, become a thing of the past. They claim that since autonomous vehicles will have the ability to communicate with all other surrounding vehicles, one’s autonomous car will be able to “predict” the movements, actions, and responses of its neighbors to keep traffic moving safely and efficiently. While our team is supportive of the development and advancement of these autonomous vehicles, we are curious to see if such claims are true as the number of vehicles on a road increases. We created the Autonomous Traffic Simulator (*ATS*), a discrete-event simulation built upon the open source ROSS framework(1). Our simulation models a simple, yet common everyday occurrence: traffic lights at intersections. We will use this simulation to model how much time the ideal autonomous vehicle will take to traverse from a starting point to a destination point when having to obey traffic lights at each intersection.**

***Keywords* –autonomous, vehicle, simulator, traffic, lights, ROSS**

I – Introduction

Autonomous cars are not a recent invention; one only needs to look at science fiction movies and novels to see countless examples of such vehicles. Perhaps the most famous autonomous vehicle is Google’s Self-Driving Car (**Figure 1)**. Google uses a high-performance computer in conjunction with a variety of sensors, most notably a LIDAR (Light Detection and Range) sensor mounted on the roof.

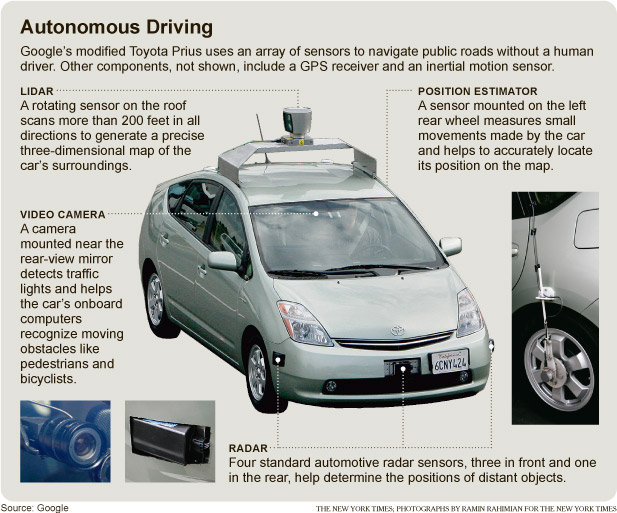
Individual Contributions:

**Bryant**: Designed and implemented ATS Traffic Light Event Handlers, wrote up this paper

**Derek:** Implemented Autonomous Vehicle Inter-Vehicle Event Handlers

**Matt**: Designed and implemented ATS Traffic Light Event Handlers, wrote up this paper

**\*\*\*Matt will be submitting our group’s code\*\*\***

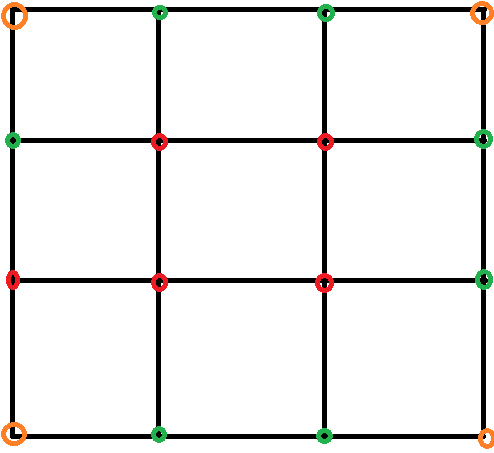


**Figure 1** – Google’s Autonomous Vehicle(2)

One of the goals of autonomous vehicles is to reduce daily commuting time to and from work. Proponents of autonomous vehicles claim that these vehicles will achieve this goal because the cars are able to communicate with each other and thus synchronize their movements to improve traffic conditions and ensure a smooth flow of traffic. However, our team was interested in the case where everyone is driving an autonomous car that is capable of communicating with all other cars. We decided to write up a simulator to see how much time the average autonomous vehicle takes to traverse from one point to another as the number of vehicles increases on a fixed world size and under the condition that in each intersection on the world, there exists a traffic light.

II – The World

In ATS, the world is referring to an *X* by *Y* grid. **Figure 2** gives an example of a 3 x 3 World.



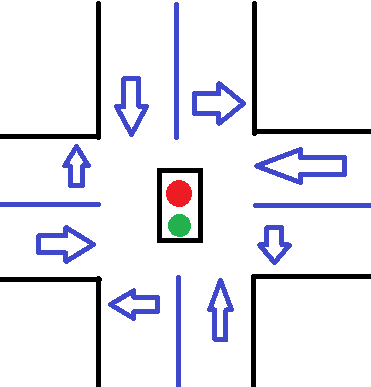
**Figure 2** – A Sample 3 x 3 World

In Figure 2, each black line segment refers to a road that autonomous vehicles can travel in. Each intersecting line (indicated by either a red, orange, or green circle) represents an intersection. At these intersections, a traffic light is placed. A red circle indicates a four-way intersection; a green circle indicates a three-way intersection, and an orange circle represents a two-way intersection.

In each road, there are two-lanes; one for each direction. For each direction, there is a left-turn lane and a straight lane.

An intersection is defined as a set of intersecting lines. A traffic light is placed at this intersection, and has two states: red or green. **Figure 3** shows an example of a four-way intersection with a traffic light.

In the world, there is an ending point defined that all cars are aiming to get to.



**Figure 3** – A four-way intersection with a traffic light

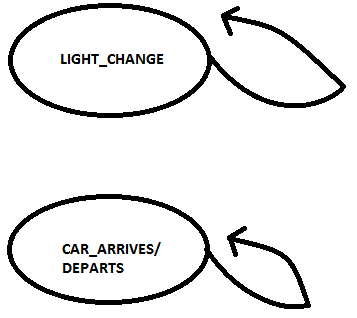
The arrows in Figure 3 show the possible directions that traffic can move in. It is important to note that our traffic lights have a ten-second counter. In addition, we assume that it takes one-second for a car to cross the intersection.

For our simulation, our world is a 1024 x 1024 square grid with 200,000 cars running. The destination coordinate point will be the same for all cars.

III – Traffic Light Event Handler Algorithm

Our team drew inspiration from the “traffic” simulation provided as part of ROSS(3).

We defined two separate events that our traffic light event handler needed to create. These events are: 1) **light\_change** (when does a light change?) and 2) **car\_arrives/departs** (handle when a car arrives/departs at the intersection). **Figure 4** shows a state table showing how the above two events interact with each other.



**Figure** **4 – State Diagram of Traffic Light Intersection Events**

When a car arrives at an intersection, the intersection calculates the number of cars arriving from each direction. To get from a starting point to an ending point, a car will first travel either north or south in the Y-Axis until the car and its destination point both have the same Y-coordinate. Next, the car will turn left or right and then travel to east or west until the car reaches the destination point. If the car has reached its destination, the event breaks and returns to main. The car arrives event calculates the next intersection and direction the car must travel to in the next iteration of the simulation. Finally, the CAR\_ARRIVES/DEPARTS event schedules another CAR\_DEPARTS/DEPARTS event.

The *LIGHT\_CHANGE* event, as indicated by Figure 4, is independent from the CAR\_ARRIVES/DEPARTS event. Similar to a real-world traffic light, our traffic light checks if the light timer has reached 0. If the time has expired,

The algorithm for the traffic light event handler is shown in **Listing** **1**:

**Listing 1 – Traffic Light Event Handler**

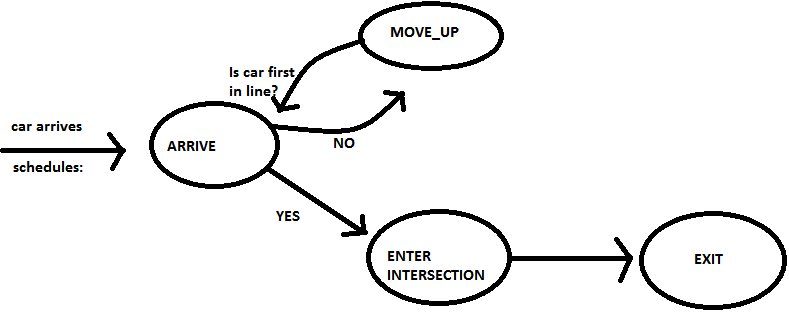
IV – Autonomous Vehicle Event Handler Algorithms

The autonomous vehicle event handler (AVEH) manages how the actual vehicles move optimally and concurrently through an intersection. We defined four events for this handler: 1) **arrive**; 2) **move\_up**; 3) **enter\_intersection**; and 4) **exit**.

When a car arrives into an intersection, the AVEH schedules an *arrive* event. This event places the car into a queue of cars. If this car is the only car in its lane, the AVEH next schedules an *enter\_intersection* event, in which the car now begins to drive into the intersection. When the car is about to leave the intersection, AVEH schedules an *exit* event.

The above case only handles the case in which the car arriving is the first and only car in the lane. In the case when an arriving car is not the first car in line, the fourth event, *move\_up* is scheduled. Move\_up calculates the amount of time the car will spend waiting in line until it is at the front of the line. AVEH will continue to schedule move\_up events until the car reaches the front of the line.

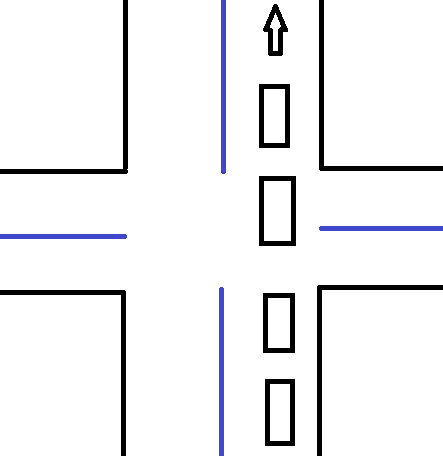
**Figure 5** shows a state diagram of how the aforementioned four events are scheduled by the AVEH.



**Figure 5** – State Diagram for the Autonomous

Vehicle Event Handler

By default, the enter\_intersection event performs a round-robin algorithm to allow cars to travel across the intersection; for instance, a car from the south lane will go, followed by a car in the west lane, then one from the north, and finally one from the east. However, to further optimize the intersection, in the enter\_intersection event, AVEH checks to see how many cars are in the opposite directions. If no cars exist, multiple cars may cross the intersection. **Figure 6** demonstrates an example of when multiple cars may enter the intersection.



**Figure 6 –** Optimizing how many cars can travel across an intersection by AVEH

In Figure 6, since there are no cars arriving in either the east-west or south left-turn directions,

multiple cars will schedule event\_intersection events and proceed across the intersection.

Another optimization is when multiple cars wish to make a right-turn from different directions. The AVEH will allow multiple cars to make these turns as long as the cars do no conflict with each other while turning.

By using these optimizations, our team would like to see if autonomous vehicles truly will decrease the commute time for everyone.

|  |  |  |
| --- | --- | --- |
| **Number Of Cores** | **Average Runtime per Car (in seconds)** | **Total Simulation Runtime (in seconds)** |
| 1 | 648 | 10.1064 |
| 64 | 644.0625 | 0.3413 |
| 128 | 645.1171875 | 0.1869 |
| 256 | 1471 | 0.0967 |
| 512 | 537.234375 | 0.126 |
| 1024 | 380.831 | 0.2749 |
|  |  |  |

**Table 1 –** Results from Naïve Algorithm – Sequential and Parallel Runs

IV – Results of Simulation

Our team ran a series of four separate simulations on the Blue Gene Q. We created two simulations that modeled a naïve navigation algorithm for the cars navigating through a traffic light without the aforementioned traffic optimization algorithm. The other two simulations include the AVEH and its intersection optimization algorithms.

We performed a strong – scaling analysis whereby our world is a grid of 128 x 128 units and cars randomly spawn in the world but share a common ending destination coordinate point.

The two simulations from both versions result from running each simulation in a sequential manner (i.e. using ROSS’s synch = 1 for a serial run and synch = 3 for a parallel run).

In **Figure 7, Figure 8, and Table 1**, our team’s run results for a sequential run for the *traffic* (no AVEH and intersection optimization) simulation is shown. Table 1 additionally shows the data from our parallel tests.

As Table 1 indicates, the average car takes an average of 648 seconds to reach the ending destination, while the entire simulation runs in 10.1064 seconds for the naïve sequential implementation. The average runtime for the sequential run can be seen in Figure 7; the total simulation runtime for the sequential run can be viewed in Figure 8.

However, as Figure 1 indicates, in general, as the number of cores increases, the average car takes less time to reach the ending destination.

**Figure 7** - Naïve Sequential Run – Average time a car spends traveling to

Its ending destination

**Figure 8** – Naïve Sequential Run – Total Simulation Time

In **Figure 9** and **Figure 10**, our group’s naïve parallel are shown.

One of the more interesting points of discussion of the parallel implementation is that the average runtime per car increases as the number of cores rises from 64 to 256, then dramatically drops down. Our team speculates that the reason behind this temporary increase is due to the MPI Sends/Receives utilized by ROSS. Unfortunately, as ROSS handles the interprocess communication between MPI ranks, our team was not able to determine nor provide actual hard-data to backup our speculation. However, we came up with this conclusion after recalling that in prior homework assignments, the performance of other parallel applications is reduced at a certain number of cores (in our naïve simulation, this number is 256 cores). In addition, our code for both the naïve implementation and the optimized implementation is a series of *if-blocks*; thus, our team believes that the computation time is in constant O(1) time. As a result, the resulting time spent must be dedicated to ROSS’ internal MPI communication.

However, as the number of cores increases, the final runtime for the naïve parallel implementation is a stunning 380.831 seconds for the average car.

Even though our team only used a maximum of 1024 cores, Figure 9’s trend seems to indicate that as the number of cores increases, the average car will have a decreased travel time.

**Figure 9** – Naïve Parallel Run – Average time spent traveling by the average car

**Figure 10 –** Naïve Parallel Run – Total Simulation Runtime

Figure 10 shows our naïve parallel run’s total simulation time. It is interesting to note that while Figure 9 showed that the total time spent by the average car traveling was the largest at 256, in Figure 10 at 256 cores, the total simulation runtime is at its lowest! Naturally, as the number of cores increases, the total simulation time will increase, since ROSS will spend more time coordinating communication between more MPI ranks.

The total simulation runtime was lowest at 256 cores, with a total runtime of 0.0967 seconds taken. Conversely, the total simulation runtime was the highest

V – Future Research

The currently developed world is simplified compared to a real world setting. Further research can be done using actual map data from Google Maps. This can accurately determine travel time between lights, the maximum number of cars between intersections, the speed limit and many more factors. In addition, adding more than one lane to an intersection creates a more realistic map. Some intersections have a combination of one to three lanes with left turning lights. Lastly, the current model only has red and green lights. Adding the yellow light will have new effects on how cars move from intersection to intersection. Most drivers tend to speed up to get through a yellow light. The newer model can mimic the same behavior.

References

1. “ROSS Wiki Main Page”, *Rensselaer Polytechnic Institute*. <http://odin.cs.rpi.edu/ross/index.php/Main_Page>,

2013.

2. “Google-self-driven-car.jpeg”, *Alecdifrawi*.

<http://www.alecdifrawi.com/wp-content/uploads/2011/08/google-self-driven-car.jpeg>, 2013.

3. “Traffic ROSS Simulation”, *Rensselaer Computer Science*. <http://www.cs.rpi.edu/websvn/filedetails.php?repname=rossnet&path=%2Ftrunk%2Fross%2Fmodels%2Ftraffic%2FIntersection.c>, 2013.