DEAKIN UNIVERSITY

CONCURRENT AND DISTRIBUTED PROGRAMMING

ONTRACK SUBMISSION

Project

Submitted By:
Gia Phu TRAN
gptran
2022/10/12 13:33

Tutor:
Mohamed DARWEISH

Outcome	**Weight**
Evaluate	♦♦♦♦♦

(none)

October 12, 2022

REALTIME SELF-DRIVING CAR WITH PARALLEL APPLICATION

SIT315 Module 4

STUDENT ID: 220344366
NAME: GIA PHU TRAN

Name: Gia Phu Tran Student ID: 220344366 Unit: SIT315

Table of Contents

Introduction

The potential and future development opportunities of self-driving cars have attracted substantial attention from investors.

It is the product in human history which enables mobility to be more accessible for those who are unable to drive; including seniors, people with disabilities, to even children. It is currently one of the world's biggest Artificial Intelligence-related research and development discipline.

The two main components of a self-driving car is a good computer vision, and a seamless sensor fusion which helps to form perception of the environment. There are also several other supporting components which will be discussed in this section.

In real world practices, a camera is used in computer vision, which real-timely captures moving images whose data will be sent to the system to identify the existence of automobiles, people, highways, etc.

Sensor Fusion, on the other hand, integrates and combines data from various sensors, such as a radar and a lidar, to supplement what the camera sees. This allows the car to estimate the locations and speeds of the things it detects.

There is also localization, which is a system that dramatically increases the location accuracy and precision, as compared to a typical GPS.

Next up is a path planner, which uses the data from the first two steps to predict what vehicles, pedestrians, objects around them will do to generate trajectories from point A to point B.

Finally, controllers are used to actuate the vehicle.

In this project, it is the path planning step which we will be focusing on. A path planner allows a car to use the data that it has regarding the environment, along with its current position, to plan trajectories. To achieve this functionality, the car needs to be able to react to data in near real-time, and needs a high computational capability for quick and precise processing.

The simulated data which contains the position of the car and the perception of the environment will be obtained from the free game-making engine - Unity, which is profoundly known for its ease of use and its power in creating seamless visualization.

Four steps of self-driving car

The detected vehicle can do several things:

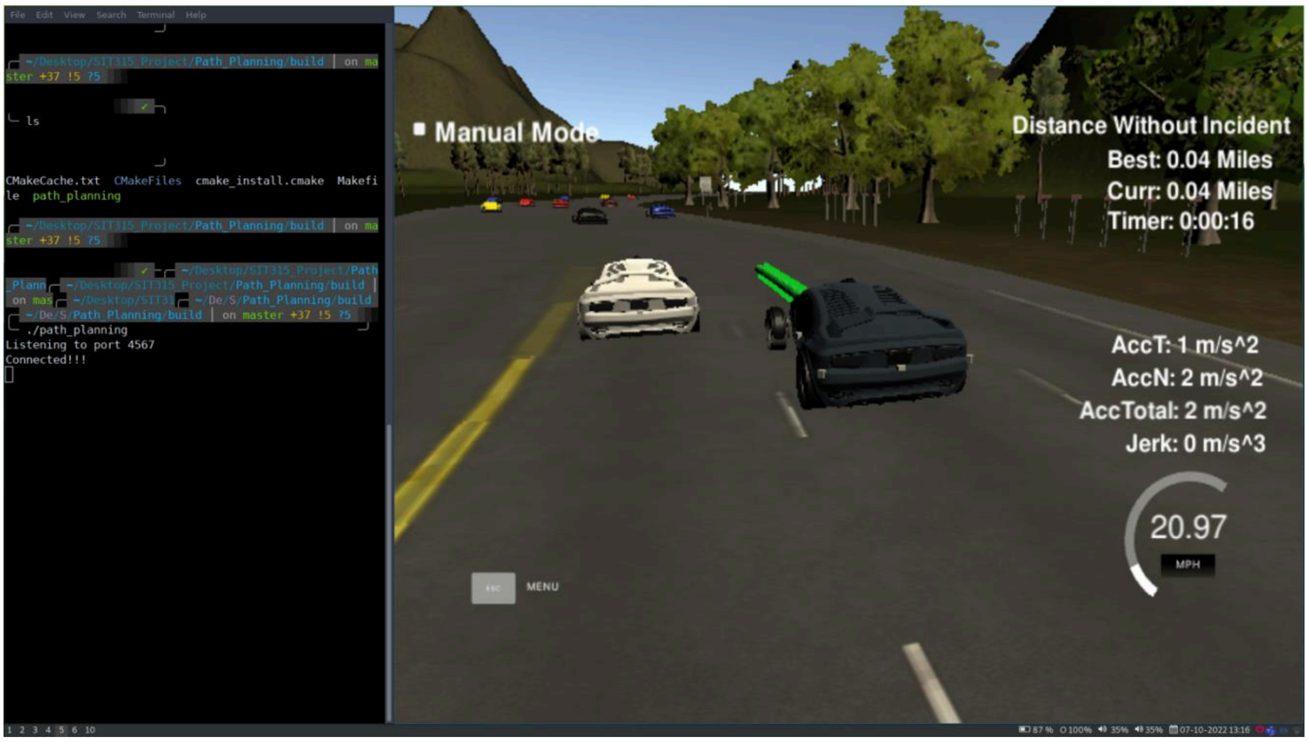
- Stay in its lane, which means:
    - Speed Up,
    - Slow Down,
    - Stay at constant speed
- Change lane

Given this, there are 4 alternative actions that the vehicle can take in the virtual highway placement

The sensors operate in near real time by continuously providing the location and the speed of the vehicle. The real-time component in this case allows a fast and precise decision if a vehicle should change gears or change lanes.

The goal of this project, is to allow the car to safely maneuver around a simulated double-directions highway with speed limit of 50MPH. On the highway, there will also be numerous other cars whose speed ranges from 40MPH to 60MPH, the speed limit of the road is 50MPH. Our main car will be provided with the car's localization and sensor fusion data which mean the perception about the other cars on the road. The perspective of our car is to maintain at the speed limit as close as possible to 50MPH speed limit, so that mean we would have to change lane if we must. For safety reasons, the car cannot make a so sudden move because that could also cause surprise for drivers in real life so our main car can not experience total acceleration over 10m/s^2 and not too much jerk over 10m/s^3 also.

For the simulation to not excel itself. The map of the highway is in data/highway_map.txt. Each waypoint in the list contains [x,y,s,dx,dy] values, x and y are the waypoint's map coordinate position, the s value is the distance along the road to get to that waypoint in meters, the dx and dy values define the unit normal vector pointing outward of the highway loop. he car should be able to make one complete loop around the 6945.554m highway.


```
╭─ ~/Desktop/SIT315_Project/Path_Planning/build │ on ma
ster +37 !5 ?5
                                        ✓ ─╮
╰─ ls
                                           ─╯
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefi
le  path_planning

╭─ ~/Desktop/SIT315_Project/Path_Planning/build │ on ma
ster +37 !5 ?5
                                        ✓ ─╮ ~/Desktop/SIT315_Project/Path
_Plann╭─ ~/Desktop/SIT315_Project/Path_Planning/build │
 on mas╭─ ~/Desktop/SIT31╭─ ~/De/S/Path_Planning/build
╭─ ~/De/S/Path_Planning/build │ on master +37 !5 ?5
╰─ ./path_planning                                  ─╯
Listening to port 4567
Connected!!!
```

The simulation environment (on the right) and program running (on the left)

Real Time

The simulation will feed us constantly through HTTP about the main car's localization data with

- ["x"] The car's x position in map coordinates
- ["y"] The car's y position in map coordinates
- ["s"] The car's s position in frenet coordinates
- ["d"] The car's d position in frenet coordinates
- ["yaw"] The car's yaw angle in the map
- ["speed"] The car's speed in MPH

And we also need the previous path data given to the Planner

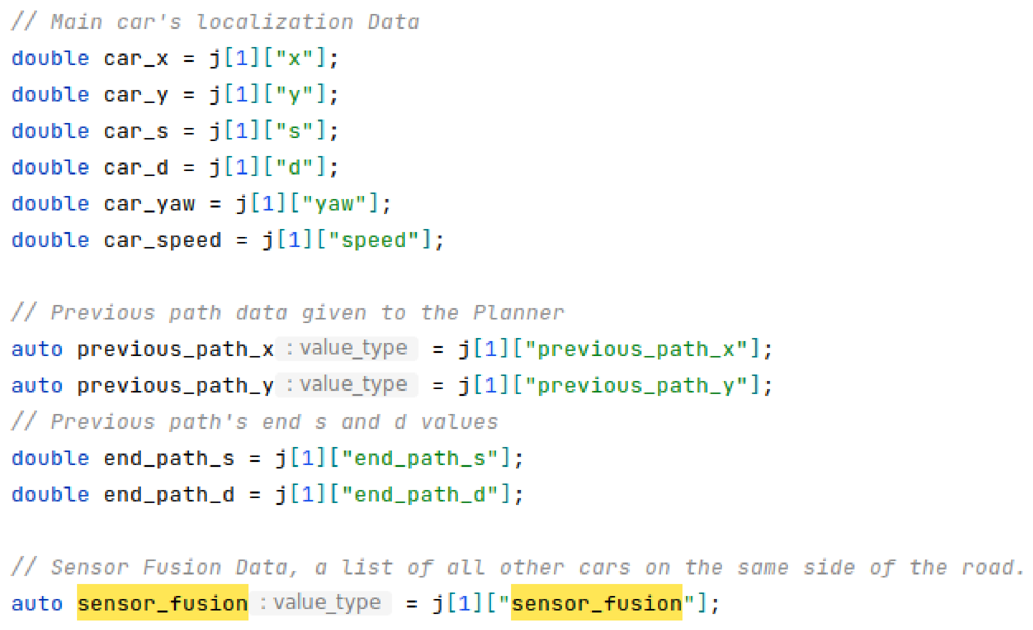
- ["previous_path_x"] The previous list of x points previously given to the simulator
- ["previous_path_y"] The previous list of y points previously given to the simulator

Previous path's end s and d values:

- ["end_path_s"] The previous list's last point's frenet s value
- ["end_path_d"] The previous list's last point's frenet d value

Finally, sensor Fusion Data, a list of all other cars on the same side of the road.

- ["sensor_fusion"] A 2d vector of cars and then that car's [car's unique ID, car's x position in map coordinates, car's y position in map coordinates, car's x velocity in m/s, car's y velocity in m/s, car's s position in frenet coordinates, car's d position in frenet coordinates].



```
// Main car's localization Data
double car_x = j[1]["x"];
double car_y = j[1]["y"];
double car_s = j[1]["s"];
double car_d = j[1]["d"];
double car_yaw = j[1]["yaw"];
double car_speed = j[1]["speed"];

// Previous path data given to the Planner
auto previous_path_x : value_type = j[1]["previous_path_x"];
auto previous_path_y : value_type = j[1]["previous_path_y"];
// Previous path's end s and d values
double end_path_s = j[1]["end_path_s"];
double end_path_d = j[1]["end_path_d"];

// Sensor Fusion Data, a list of all other cars on the same side of the road.
auto sensor_fusion : value_type = j[1]["sensor_fusion"];
```

The information sent through HTTP and how we received it

The automobile has a perfect controller and visits every (x,y) place in the list every.02 seconds. The units for the (x,y) points are metres, and the spacing between the points sets the car's speed. The angle of the automobile is determined by the vector that connects one place to the next in the list. Acceleration is measured in both the tangential and normal directions, and will be represent as the jerk, which is the rate of change of overall acceleration.

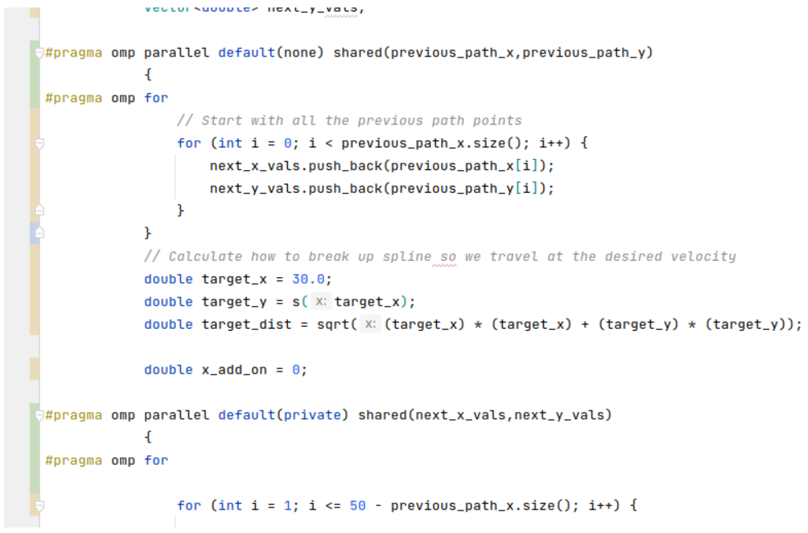
The overall acceleration of the (x,y) point pathways received by the planner should not exceed 10 m/s2, and the jerk should not exceed 10 m/s3.

So, in short, the simulation feeds us with real-time data, and we provide them with real-time instructions

Parallel

The hardest part of the program is to doing a lot of calculations and still have to return the instruction at the right time. So, in order to do that we need to parallel the biggest calculations part which is

- going through the list of all other cars on the same side of the road and calculate if it is too close to our main car
- Going through all the previous path and calculating the trajectories we will using



```
        vector<double> next_y_vals;

#pragma omp parallel default(none) shared(previous_path_x,previous_path_y)
        {
#pragma omp for
            // Start with all the previous path points
            for (int i = 0; i < previous_path_x.size(); i++) {
                next_x_vals.push_back(previous_path_x[i]);
                next_y_vals.push_back(previous_path_y[i]);
            }
        }
        // Calculate how to break up spline so we travel at the desired velocity
        double target_x = 30.0;
        double target_y = s( x: target_x);
        double target_dist = sqrt( x: (target_x) * (target_x) + (target_y) * (target_y));

        double x_add_on = 0;

#pragma omp parallel default(private) shared(next_x_vals,next_y_vals)
        {
#pragma omp for

            for (int i = 1; i <= 50 - previous_path_x.size(); i++) {
```

An example of applying parallel for faster calculation

There will be some delay between the simulator running and the path planner delivering a path; But to avoid our calculation still not fast enough however, with optimized code, this is normally only 1-3 time steps. During this period, the simulator will continue to utilize the last points it was given; thus, it is a good idea to save the last points you used so that the transition is seamless.

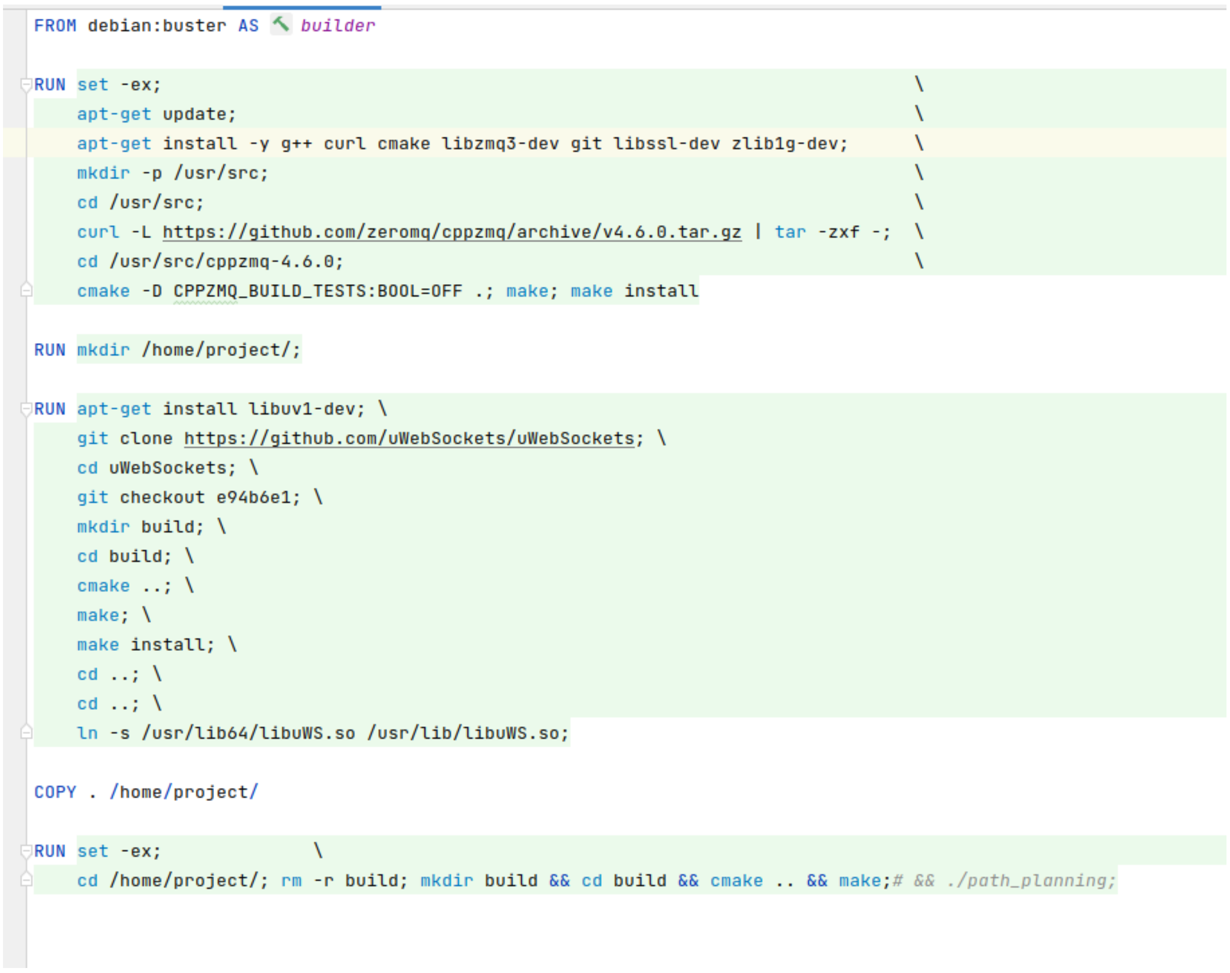
Previous path x and previous path y might be useful for this transition since they display the last points sent to the simulator controller without the processed points. You would either return a path that extends the existing path or build a new path that transitions well with the prior path.

Spline lib was an extremely helpful resource for accomplishing this project and making smooth trajectories; the spline function is in a single hearer file and is very simple to use.

Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

In order to provide a better understanding of the whole project so I wrap up my code in a container.

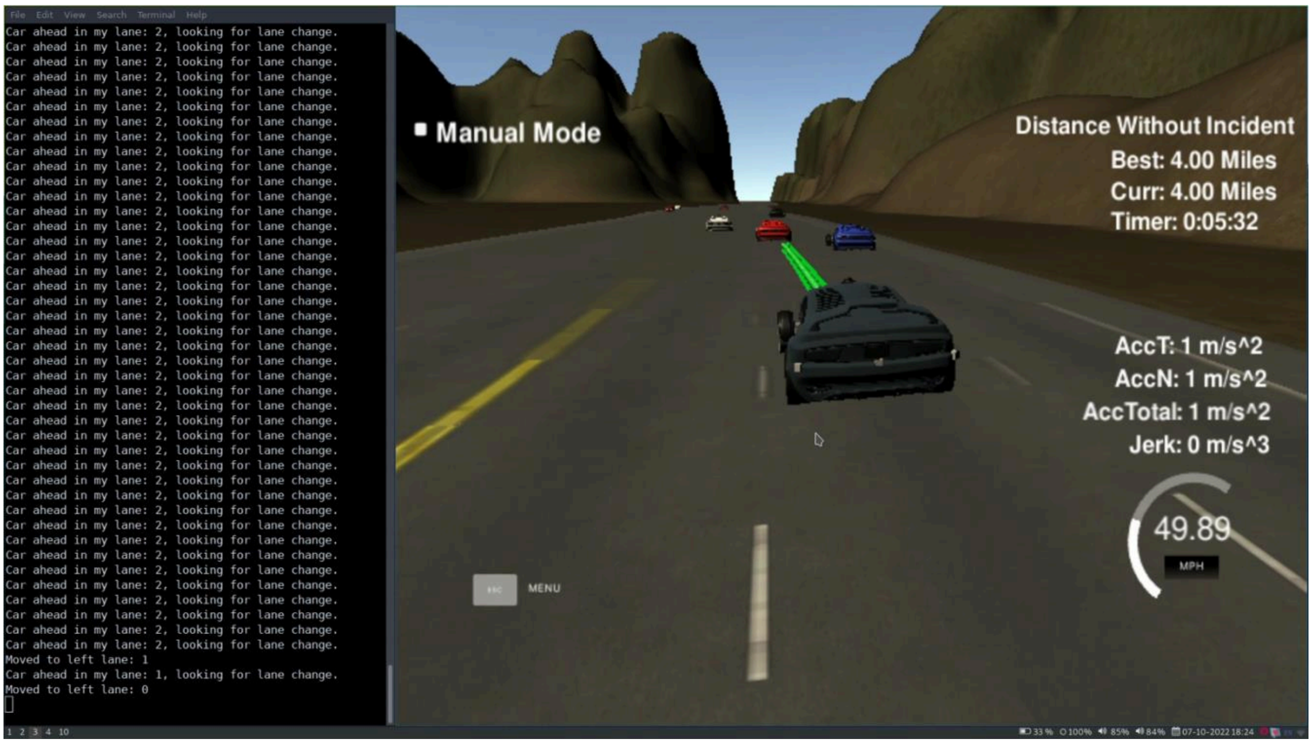


```
FROM debian:buster AS builder

RUN set -ex;                                                             \
    apt-get update;                                                      \
    apt-get install -y g++ curl cmake libzmq3-dev git libssl-dev zlib1g-dev;     \
    mkdir -p /usr/src;                                                   \
    cd /usr/src;                                                         \
    curl -L https://github.com/zeromq/cppzmq/archive/v4.6.0.tar.gz | tar -zxf -;  \
    cd /usr/src/cppzmq-4.6.0;                                            \
    cmake -D CPPZMQ_BUILD_TESTS:BOOL=OFF .; make; make install

RUN mkdir /home/project/;

RUN apt-get install libuv1-dev; \
    git clone https://github.com/uWebSockets/uWebSockets; \
    cd uWebSockets; \
    git checkout e94b6e1; \
    mkdir build; \
    cd build; \
    cmake ..; \
    make; \
    make install; \
    cd ..; \
    cd ..; \
    ln -s /usr/lib64/libuWS.so /usr/lib/libuWS.so;

COPY . /home/project/

RUN set -ex;              \
    cd /home/project/; rm -r build; mkdir build && cd build && cmake .. && make;# && ./path_planning;
```

Dockerfile using for deployment

Evaluation

The result turns out to be surprisingly well, in 7 minutes of runtime our main car doesn't cause any collision. It did make some incidents only by going over the speed limit of 50MPH a bit. If we change the limit to lower the results can be much better



```
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Car ahead in my lane: 2, looking for lane change.
Moved to left lane: 1
Car ahead in my lane: 1, looking for lane change.
Moved to left lane: 0
```

A runtime example of 5 minutes

Comparing with the when we doing autonomous without parallel the car clearly have more collusions the reaction of the car seem slower and the Jerk always seem to be high because the slow instruction.

	Distance without Incident	**Highest Acceleration**	**Jerk warning**
Without OMP	1.82 miles	9 m/s^2	3 times
With OMP	4.07 miles	8m/s^2	0 times

Table compare the different when using OMP and without

So clearly with OMP we can give a much faster response so we give our passenger a safer, smoother journey.

Source code on GitHub

I have provided my source code on GitHub with Dockerfile, docker-compose.yml and CMake file include to compile:

https://github.com/phulelouch/SIT315_Parallel_Path_Planing_Docker

Video Demonstration

https://www.youtube.com/watch?v=02Z2GojQh8s

Reference

https://github.com/udacity/self-driving-car

https://github.com/udacity/self-driving-car-sim

```
Source code on GitHub
I have provided my source code on GitHub with Dockerfile, docker-compose.yml and
↪  CMake file include to compile:
https://github.com/phulelouch/SIT315_Parallel_Path_Planing_Docker


```

https://www.youtube.com/watch?v=02Z2GojQh8s