Parallelized Traffic Simulation

Denzel Farmer (df2817) Andrew Yang (asy2130)

Fall 2023

1 Introduction

A nearly universal experience as a driver is getting stuck in traffic. Congested highways often slow to a near stand-still while advertising speed limits of 60-80 mph. These bottlenecks can appear for seemingly no reason, and can prove highly frustrating to students returning from a relaxing Thanksgiving vacation.

With this in mind, we created a highway traffic simulator, which we used to simulate the flow of traffic on a highway, as well as including random events to make the simulation more dynamic. By taking advantage of existing driver behavior models, specifically the Intelligent Driver Model (IDM), we were able to construct a highly parallelized to allow fast simulation of scenarios involving thousands of vehicles across hundreds of lanes.

2 Objective

Our goal was to build a highly parallelizable traffic simulator, to observe the flow of highway traffic as it changes over some time period, based on configurable parameters.

We decided to simulate highway traffic microscopically, considering the behavior of each individual driver based on models meant to emulate human behavior. Based on this simulation, we were also able to observe how traffic flow responds to random stimuli (for example, a car briefly slowing down to avoid an obstacle) like in the real world.

We also had an idea of implementing lane changes between lanes, which would allow cars to switch between lanes. However, as discussed later, the performance of our implementation was highly inefficient and did not work well with our parallelized design, since it involved both time intensive sequential operations for each car in each lane (such as sorting) as well as broke the independence between lanes that had allowed for the speed up.

3 Implementation

We first constructed a sequential implementation of the simulator, using structures that would later be conducive to parallelization. Then, we expanded that implementation to include parallelization.

We split the application into a few distinct parts: code to read and parse configuration parameters, a generic simulation framework, traffic-specific data structures and operations, and code to render the output itself. Of these, we focused most of our development on the traffic modeling code

rather than optimizing other, less important components (like reading configuration or writing to disk).

3.1 Main Runner

The main runner code in `Main.hs` is responsible for all the `IO` code in our application. It performs argument and config parsing, evaluates the simulation result, and writes the output to a file.

To run a simulation, our code required a variety of parameters, both from the command line and from a JSON configuration file. We used the `optparse-applicative` library to read in the following arguments:

```
traffic-sim-exe (-c|--config FILENAME) (-o|--outputfile FILENAME)
                [-n|--iterations INT] [-x|--no-output BOOL]
```

The required arguments are a JSON configuration filepath to read further parameters and an output filepath for outputting the simulation result. We also include optional parameter `no-output`, which we use for certain evaluation runs; when set to true, the bulk of time-consuming simulation output generation is disabled (although we ensure that all components are still evaluated fully, confirmed with profiling).

A function `parseConfigFile :: String -> IO SimConfig` in `Config.h` reads the provided config file from disk and uses the `aeson` package to parse the result into a `SimConfig` object, which contains other configuration objects used by the simulator. This requires slightly ugly 'parsable' wrappers around various configuration objects, but since elegant configuration parsing wasn't our main goal, it works for our purposes.

With a configuration object, the main runner evaluates the pure simulation function `runSimLoop :: SimArgs -> SimConfig -> SimResult` defined in `Sim.h` and writes the output result to the provided output file.

3.2 Generic Simulation Structure

To evaluate the result of a simulation, we use the `runSimLoop` function. This function (and the `Sim.h` code in general) is mostly agnostic to the details of the simulation, and instead operates on `Scene` objects.

The simulation function lazily defines a 'generator' list using the following function:

```
buildByteResult :: Scene ->
                             (Scene -> (Scene, B.ByteString)) ->
                             [B.ByteString]
```

This helper is similar to the `iterate` function, and generates an infinite list of bytestrings representing the simulation result at each timestep using a provided update function. The simulation runner code passes a partially applied `updateScene` function, which takes in a current `Scene` of the simulation and generates a tuple containing the next `Scene` object and a bytestring describing the current scene.

We found experimentally that generating bytestring intermediate results with each update was significantly more efficient than our first naive solution, accumulating and converting a list of `Scene` objects (although both have significant disadvantages compared to a more standard streaming approach).

After taking the requested number of bytestrings from the generator, `runSimLoop` wraps the result in a `SimResult` and returns it.

3.3 Traffic Modeling Structure

To implement the traffic simulation model, we use a few data key data structures: `Scene`, `Lane`, and `Car`. Each is implemented in its own `.hs` file, and they have related configuration objects defining their static configuration.

3.3.1 Scene

The `Scene` object and its related configuration objects are the interface between the previously described simulation code and the traffic simulation model. A `Scene` represents the state of the entire simulation at a single timestep, and consists of a list of `Lane` objects.

The most important function operating on the `Scene` data is `updateScene`:

```
updateScene :: SceneConfig ->
                SpawnerConfig -> Bool -> Scene ->
                (Scene, B.ByteString)
```

This function takes in a `Scene` (along with configurations) and computes both the next scene in the simulation, and the bytestring representation of the current string. To do this, it maps the `updateLane` function over the contained list of `Lane` objects, and merges the results.

In the lane-parallelized version of our code, we use the following `ParMap` rather than a simple `map`:

```
(nextLanes, currLaneBytes) = unzip $ parMap
    rdeepseq updateLaneFunc currLanes
```

This computes the update for each lane in parallel by fully evaluating the result of the `updateLaneFunc` function.

In our prototypes which implemented lane-switching, updating a scene also included a pass over the lanes which performed switching calculations.

3.3.2 Lane

A `Lane` object contains an ordered list of `Car` objects, and represents the current state of a single lane of traffic at a single time step:

```
    data Lane = Lane {
    lnCars :: [Car],
    lnStep :: Int,
    lnMaxCarId :: Int,
    lnSeed :: Int,
    lnID :: Int
} deriving (Show, Generic, NFData)
```

In addition to the list of cars, each `Lane` includes the current simulation step, the maximum ID assigned to any `Car` object (since car IDs are per-lane), a random seed, and a numeric identifier.

As with the `Scene` object, there is an `updateLane` function, which takes in configuration parameters and a current `Lane` and outputs the next lane and the byte representation of the current lane:

```
updateLane :: Bool -> SceneConfig ->
                SpawnerConfig -> Lane ->
                (Lane, B.ByteString)
```

Updating a lane involves a filtering away cars which have passed beyond the maximum boundary, potentially spawning a new car based on the `spawnRate` parameter, and updating the acceleration, velocity, and position of each car (based on its current state and that state of the car immediately in front of it). Each of these actions is handled by a separate helper function.

In the chunk-parallelized implementation, we perform a bit more work, splitting the car list into chunks with the `generateChunks` function. We then perform a `ParMap` over those chunks with the `processChunks` helper, which does the following:

```
cRes = parMap rdeepseq (processChunk nDisp dT) chunks
(cars, byteStrings) = unzip chunkResults
```

3.3.3 Car

The most important data structure is the `Car` itself. Each car has a car ID, which is important for pairing each car with a consistent color within the Python rendering script.

To store the physical motion attributes, each car contains a position, velocity and acceleration Float value. Each car also keeps track of its desired speed, which is important for making the car enforce a current speed, especially if it is at the head of the line and could theoretically accelerate forever. This is also important for realism, as cars in the real world clearly cannot accelerate forever. Cars also have a maximum deceleration and acceleration, to simulate real world conditions, considering that cars cannot halt immediately nor can they accelerate at an infinite rate.

Each car has a minimum gap parameter that serves as the minimum permissible gap between a car and its immediate neighbor. Each car also has a desired time gap between itself and the car preceding it, which is represented as a time instead of a physical gap because the gap increases when speed increases, just like in the real world.

```
data Car = Car
  { cID :: Int,
    cPos :: Float,
    cVel :: Float,
    cAccel :: Float,
    cDesiredSpeed :: Float,
    cMinGap :: Float,
    cDesiredTimeGap :: Float,
    cMaxDecel :: Float,
    cMaxAccel :: Float
  } deriving (Generic, NFData, Show)
```

There are two main update functions that act on the `Car` object. The `updateCarAccel` function performs the IDM/ACC acceleration calculation described in the next section, determining how a particular car reacts based on its state and the state of its leading car (or a dummy car, if it has no

leader). The `updateCarPosVel` applies simple equations of motion to update a single car's position based on its velocity and velocity based on its acceleration.

3.3.4 Driver-Model Algorithm

We used a car-following driver model that takes in position and speed information about a car and the car directly in front of it, and decides how much to accelerate/decelerate at each time step to maintain a safe following distance.

In particular, we implemented the Intelligent Driver Model described in chapter 11 of Traffic Flow Dynamics [1].

This model describes driver behavior as a choice of acceleration based on the following equation:

$$\frac{dv}{dt} = a\left[1 - \left(\frac{v}{v_0}\right)^4 - \left(\frac{s^*(v, \Delta\ v)}{s}\right)^2\right] \tag{1}$$

The first term in this equation, $1 - \left(\frac{v}{v_0}\right)^4$, represents the fraction of maximum acceleration a a driver would choose if there were no leading car, based on the car's current velocity v and the driver's desired velocity v_0.

The second term takes into consideration the car directly in front, and represents 'breaking' the driver does in response to a leading car. This calculated with a ratio of the actual gap s and the 'desired' gap s^*. The desired gap is calculated as follows:

$$s^* = s_0 + max\left[0, 1 - \left(vT + \frac{v\Delta v}{2\sqrt{2ab}}\right)\right] \tag{2}$$

This is based on the driver's minimum acceptable stopped gap s_0, the current velocity v, the driver's desired time gap T, the difference between the current car's velocity and the velocity of the leading car Δv, the maximum acceleration a, and the maximum deceleration b.

To implement this model, for each simulation step we recalculate the desired acceleration for each car at time t based on its state, its leader's state, and that car's 'profile' of the constant parameters a, v_0, s_0, T and b. Then, we recalculate position and velocity for the next timestep $t + \Delta t$ assuming constant acceleration over the time step. If the calculated velocity would be negative, we coerce it to 0.

3.3.5 Configuration Parameters

To configure the application, we allow a number of configuration parameters. These are passed in JSON format, and affect how the simulation proceeds. In particular, these include:

- simSceneConfig: a JSON object configuring the initial scene
  - Max Position: the boundary position (in meters) after which cars will despawn
  - Step Time (s): the duration of time in each time step when updating position/velocity
  - Lanes: the number of lanes in the scene
  - Initial Car Count: the number of cars to begin with (evenly spaced from 0 to boundary)
  - Initial Car Config: JSON object describing the driver profile for initial cars (contents not shown)

- simSpawnerConfig: a JSON object configuring the continous car spawner
  - Random Seed: a seed to randomize driver profile choice
  - Spawn Rate (steps): the number of steps in between spawning new cars
  - Spawn Speed (m/s): the speed at which spawned cars should start
  - Config Frequencies: a list of floats of the same length as Car Configs, providing a weight to use when randomly selecting driver profiles
  - Car Configs: a list of JSON objects describing the possible driver configurations to randomly select from when spawning a new car

Each car configuration contains the components of the driver profile described in the previous section:

- Desired Speed (m/s)
- Minimum Gap (m)
- Desired Time Gap (s)
- Max Deceleration (m/s^2)
- Max Acceleration (m/s^2)

3.4 Rendering

We were able to evaluate that our simulation worked through visualizing our output file using a Python script that took line separated JSON formatted values that represented each "timestamp." We were then able to generate static images for each frame, before stitching them together into an MP4 video. We were able to verify that our code met our expected output, as we were able to see the cars move across the screen evenly, as well as see the random variation in acceleration that we generated.

The Python script works by accepting a header line that contains important information about the configuration, such as the number of iterations, number of lanes, and a bound on the x-axis so that the renderer knows what the scale should be.

Using the number of iterations read through the header, it reads through that many lines. At each stage, it draws a rectangle onto the canvas, using the interpolated x and y position calculated from the Car data it read in as well as using the lane it is in.

4 Parallelization

Our approach towards parallelization underwent several iterations, starting with a "naive" approach that parallelized every single operation, then parallelizing chunks of operations that would be calculated sequentially, before eventually parallelizing computation across multiple lanes that are functionally independent.

4.1 No Chunks

In our first iteration, we did not have any chunks at all, to see if there would be any speedup. For this, we paired every single car with the car in front of it, placed those in a list, and then called **parMap** on the list, using the rdeepseq strategy, and using our acceleration, velocity, and position functions on each pair, which would return a list of the first cars in each pair.

We were able to see that not having any chunks had a negative effect on our computation speed. Given that the individual kinematic equations were comprised of a small number of simple arithmetical calculations, it made sense that the overhead of creating and executing each thread dominated the time spent calculating the equations themselves. As a result, we decided to follow a chunking approach in order to ensure that the time spent on overhead was not wasteful.

4.2 Chunking

In our next iteration, we implemented a chunking algorithm that split the original list of cars into chunks of any arbitrary size. However, because of the reliance on consecutive cars in order to calculate our motion equations, simply partitioning the list would not work. Specifically, the last car in every chunk would not be able to calculate its acceleration on its immediate predecessor, which would lead it to accelerate as if there was no vehicle in front of it. To remediate this, we had to create new chunks of length two that would account for the boundary between the n sized chunks.

After testing this however, we realized that the unevenly sized chunks were not conducive for effective parallelization, as it could be possible for a huge variation in chunk size (for example the chunks of size 2 could be computing alongside chunks of size 1000) which subsequently resulted in dramatic imbalances in the operations that each core was calculating this.

To remedy this, we restructured the chunking algorithm such that every chunk would be the same size. We were able to do this by having a one element overlap between each chunk, meaning that every car would have a proper calculation. We would then drop the calculated value from each chunk, since it would consider the last value as having no predecessor, which is not true.

4.3 Lane Parallelization

Beyond this, we looked beyond processing a single lane in a parallel fashion. This matches well with the real world, since in most situations, a single highway will have multiple lanes of traffic going at once. In addition, since we did not implement any lane changes, the individual lanes are functionally independent, a prime opportunity for parallelism.

To do this, we called **parMap** on a list of lanes. In our implementation, it was expected that each lane would have a roughly equal amount of cars in the scene at all times, as we set a fixed rate for spawning cars between all lanes, and each lane started with the same number of cars. In theory, since cars have the ability to stop/slow down randomly, it could be possible over time that each lane would have drastically differing lengths. However, in our testing, the number of cars in each lane was about the same.

At this scale, with potentially hundreds if not thousands of cars per lane, and a theoretically unbounded number of lanes, the power of parallelization became especially apparent. The issues that we had initially with regards to calculations that were too fast were addressed, as the time needed per calculation became larger on several orders of magnitude in cases where large numbers of cars would be present in each lane at once. This meant that the overhead that was initially

a waste of time was a relatively small amount of time compared to the time spent on actually "productive" calculations.

4.4 Combination Approach

Ultimately, we found that a combination of chunking and parallelization between lanes yielded the best speed up in our use case. This was because the amount of operations needed to calculate the values for each lane had grown large enough that it made sense to sacrifice a bit of computation time generating chunks in order to be able to calculate the list values more effectively.

4.5 Limitations

As is the case for most parallel programs, we were restricted by Amdahl's Law, which meant that any improvement in the parallel calculations would not be able to speed up the inevitable sequential portions. Fortunately, the sequential operations in our program were not extremely time intensive, although they took a non-negligible amount of time.

The sequential part of our code was mainly limited to constant time operations, with the exception of generating chunks, which was necessarily an O(n) operation.

Another limitation was the fact that our timesteps needed to be synced. As a result, if any chunk had finished computing earlier than the others, it would need to wait. However, given the even sized chunks and theoretically consistent calculation time needed for each chunk to be processed, we expect that this was not a signficant limitation.

When writing mode was enabled, there was also a significant amount of sequential time spent writing to disk, which is why we allowed the user to turn that off in order to benchmark actual parallel code.

5 Evaluation

We ended up evaluating our code by comparing our multithreaded version with the single threaded version. Our test machine for this project was a M1 MacBook Pro, with 8 cores, 8 threads, and 16 GB of RAM.

We appended the GNU "time" command at the front of every executions, and used the provided time as the source of truth for our comparisons.

6 Results

In the following chart, we can see how the speedup peaks when 4 cores are used, but is slower with larger and smaller numbers of cores.

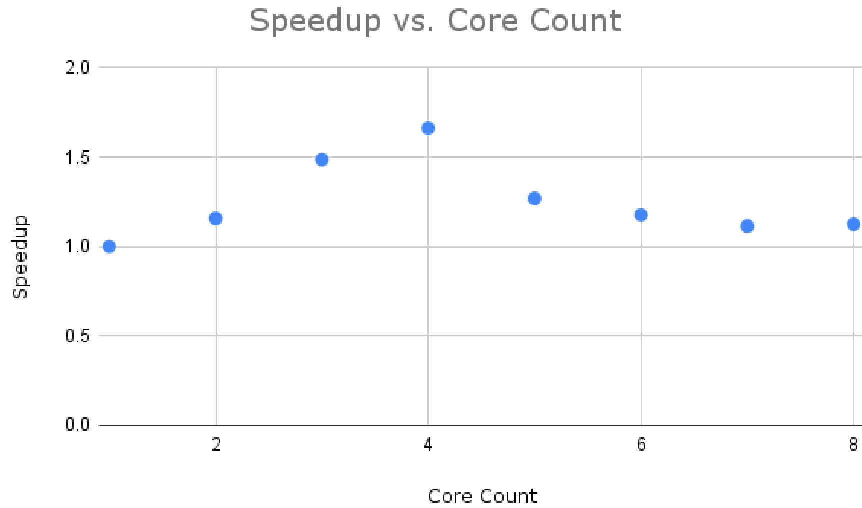

Figure 1: Speedup vs. Core Count over 50 iterations, 10,000 initial cars, 10,000 max scene size

Below, we can see that as we increase the chunk size to approach the number of cars, the total computation time continues to approach 10 seconds. With small chunk sizes, the time elapsed gets significantly larger, which makes sense as the overhead to computation speedup ratio gets exceptionally high with small chunk sizes.

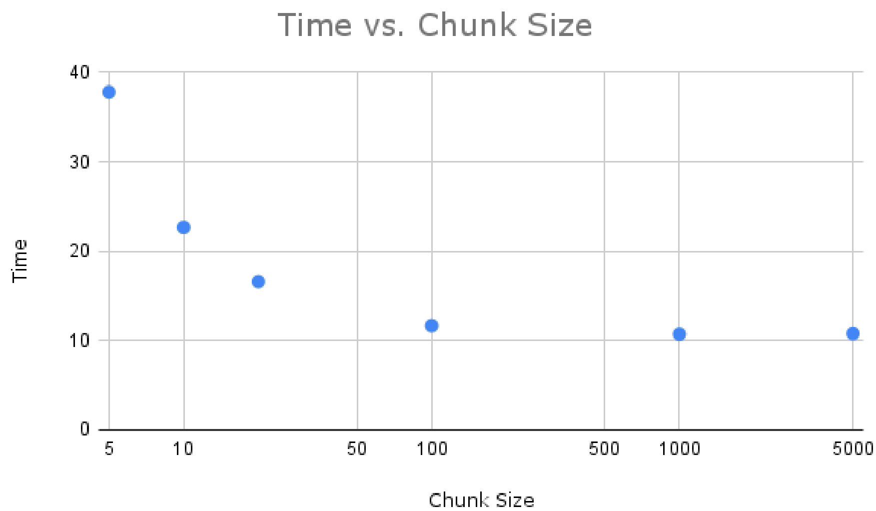

Figure 2: Time vs. Chunk Size over 50 iterations, 10,000 initial cars, 10,000 max scene size

When comparing four cores vs one core, we see that at lower values, the amount of speedup with the parallel approach is slightly lower than when it is at higher values. However, the factor differential is still relatively stable.

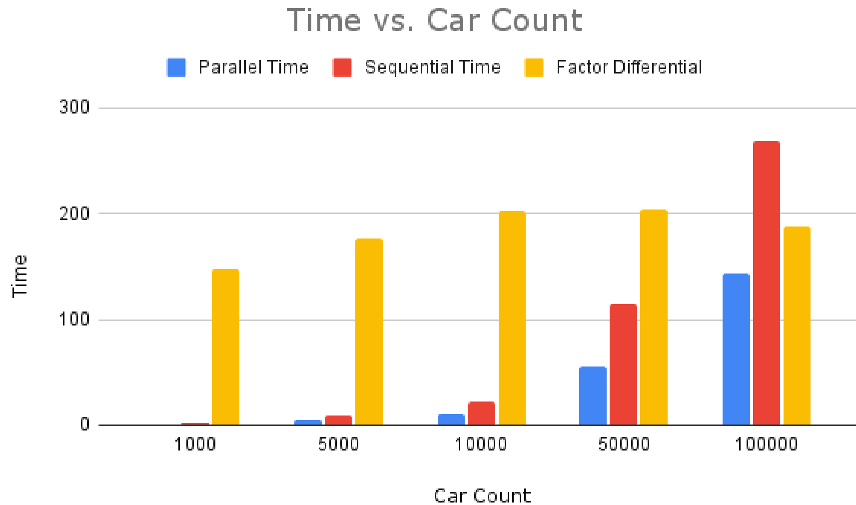

Figure 3: Time vs. Car Count over 50 iterations, 10,000 initial cars, 100,000 max scene size

6.1 Future Improvements

6.2 Lane Switching

While we made a few prototypes which implemented basic lane switching, their sequential performance penalty was too high for reasonable use. We believe this is due to our choice of data structure representing a lane, and as such could be improved significantly.

Doing so would have significant implications for parallelism, because it would create computationally intensive tasks that must be completed in sequence after each step calculation.

6.2.1 Streaming-Based Output

As it currently stands, we build up bytestring in memory and write to disk at the end. However, this means that it takes a significant amount of sequential computation at the end of execution. To make this sequential work faster, one idea we had was to interleave the writing to the file with the actual computation of each timestamp. To do this, a possible solution could be to use a streaming package to stream to disk during the computation process. Another potential alternative is to reduce the frequency of writes by only recording values periodically instead of every time stamp. However, this could cause issues with the rendering of the MP4 as frames would necessarily be dropped.

References

[1] Treiber, Martin, et al. "Chapter 11: Car-Following Models Based on Driving Strategies." *Traffic Flow Dynamics: Data, Models and Simulation*, Springer, Berlin, 2013.

[2] Kesting, Arne, et al. "General Lane-changing model Mobil for car-following models." *Transportation Research Record: Journal of the Transportation Research Board*, vol. 1999, no. 1, 2007, pp. 86–94, https://doi.org/10.3141/1999-10.

[3] `https://github.com/movsim/traffic-simulation-de/tree/master`

7 Source Code

7.1 Main.hs

```
module Main (main) where

import Config (parseConfigFile,
               SimConfig)

import Sim (runSimLoop, simResultToByteString, SimArgs(..))

import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BSC

import System.Exit (exitFailure)
import Control.Exception (catch, IOException)
import Options.Applicative (Parser, strOption, long, short, metavar, help,
                            option, auto, showDefault, value, execParser,
                            info, (<**>), helper, fullDesc, progDesc, header)

data TrafficSimArgs = TrafficSimArgs
  { configFile :: String,
    outFile :: String,
    numIterations :: Int,
    tsNoDisplay :: Bool
  } deriving Show

arguments :: Parser TrafficSimArgs
arguments = TrafficSimArgs
  <$> strOption
      ( long "config"
     <> short 'c'
     <> metavar "FILENAME"
     <> help "Path to the configuration file"
      )
  <*> strOption
      ( long "outputfile"
     <> short 'o'
     <> metavar "FILENAME"
     <> help "Path to place output file"
      )
  <*> option auto
      ( long "iterations"
     <> short 'n'
     <> metavar "INT"
     <> help "Number of iterations to run the simulator"
     <> showDefault
     <> value 10 -- Default to 10 iterations
      )
  <*> option auto
    (
      long "no-output"
      <> short 'x'
      <> metavar "BOOL"
      <> help "Don't save output (for benchmarking)"
```

```
    <> showDefault
    <> value False
  )

main :: IO ()
main = runTrafficSim =<< execParser opts
  where
    opts = info (arguments <**> helper)
                ( fullDesc
                <> progDesc "Traffic Simulator Application"
                <> header "Traffic Simulator — a tool for simulating traffic" )


runTrafficSim :: TrafficSimArgs -> IO ()
runTrafficSim args
        | numIterations args <= 0 = do
            putStrLn "Error: Number of iterations must be a positive integer."
            exitFailure
        | otherwise = do
            simConfig <- parseConfigFile (configFile args) `catch` handleConfigParseError

            let simArgs = SimArgs {noDisplay = (tsNoDisplay args),
                                   numIters = (numIterations args)}

            let simResult = runSimLoop simArgs simConfig
            let simResultBytes = simResultToByteString simResult
            let numStepBytes = BSC.pack ("{Steps = " ++ (show (numIterations args)) ++ "}")

            B.writeFile (outFile args) $ numStepBytes `BSC.append` simResultBytes

handleConfigParseError :: IOException -> IO SimConfig
handleConfigParseError e = do
  putStrLn $ "Error parsing config file: " ++ show e
  exitFailure
```

7.2 Car.hs

```
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module Car (Car(..), CarConfig(..), CarState(..), initCar, initCarState,
moveCarStateForward, calcPos, calcVel, updateCarPosVel, carWithinBound,
updateCarAccel, calcAccel, calcFreeAccel, calcBreakingAccel) where

import Control.Parallel.Strategies (NFData)
import GHC.Generics (Generic)
import qualified Data.ByteString as B

data CarState = CarState
  {
    csPos :: Float,
    csVel :: Float,
    csAcc :: Float
  } deriving (Show, Eq)

-- Initialize a car state to the given position
-- with 0 acceleration and velocity
initCarState :: Float -> Float -> CarState
initCarState pos vel = CarState pos vel 0

-- Constants for a single car
data CarConfig = CarConfig
  {
      ccDesiredSpeed :: Float,
      ccMinGap :: Float,
      ccDesiredTimeGap :: Float,
      ccMaxDecel :: Float,
      ccMaxAccel :: Float
  } deriving (Show, Eq)


data Car = Car
  { cID :: Int,
    cPos :: Float,
    cVel :: Float,
    cAccel :: Float,
    cDesiredSpeed :: Float,
    cMinGap :: Float,
    cDesiredTimeGap :: Float,
    cMaxDecel :: Float,
    cMaxAccel :: Float
  } deriving (Generic, NFData, Show)

initCar :: Int -> CarConfig -> CarState -> Car
initCar carID carConfig carState =
        Car { cID           = carID
        , cPos              = csPos carState
        , cVel              = csVel carState
        , cAccel            = csAcc carState
        , cDesiredSpeed     = ccDesiredSpeed carConfig
        , cMinGap           = ccMinGap carConfig
```

```
          , cDesiredTimeGap = ccDesiredTimeGap carConfig
          , cMaxDecel       = ccMaxDecel carConfig
          , cMaxAccel        = ccMaxAccel carConfig
          }

-- Allow sorting cars by location initially
instance Ord Car where
  compare a b = compare (cPos a) (cPos b)

instance Eq Car where
    a == b = cPos a == cPos b

moveCarStateForward :: Float -> CarState -> CarState
moveCarStateForward dist carstate = carstate {csPos = forwardPos}
 where
  forwardPos = (csPos carstate) + dist

carWithinBound ::  Float -> Car -> Bool
carWithinBound bound car = (cPos car) < bound

updateCarAccel :: Car -> Car -> Car
updateCarAccel car next = car {cAccel = calcAccel curV deltaV
curGap car}
  where
    curV = cVel car
    deltaV = curV - (cVel next)
    curGap = (cPos next) - (cPos car)

-- Calculate the actual acceleration chosen by the driver based on
current speed, gap, delta-V, and characteristics
calcAccel :: Float -> Float -> Float -> Car -> Float
calcAccel curV deltaV curGap car = maxAccel * (freeDesAcc - breakingAccel)
 where
  maxAccel = cMaxAccel car
  freeDesAcc = calcFreeAccel curV car
  breakingAccel = calcBreakingAccel curV deltaV curGap car

-- Calculate the 'free desired' acceleration; fraction of max the driver would accelerate
-- on an unobstructed road based on current speed and desired speed
calcFreeAccel :: Float -> Car -> Float
calcFreeAccel curV car = 1 - (curV/desV) ^ (4 :: Int)
 where
  desV = cDesiredSpeed car


-- Calculate the 'breaking' acceleration; fraction of max acceleration driver doesn't
-- use because they are getting too close to the car in front
calcBreakingAccel :: Float -> Float -> Float -> Car -> Float
calcBreakingAccel curV deltaV curGap car = (dynamicDesiredGap / curGap) ^ (2 :: Int)
 where
  dynamicDesiredGap =  (cMinGap car) + (max 0 (ownSpeedFac + relSpeedFac))
  ownSpeedFac = curV * (cDesiredTimeGap car)
  relSpeedFac = (curV * deltaV) / ((2) * (sqrt (maxAccel * maxDecel)))
  maxAccel = cMaxAccel car
```

```
  maxDecel = cMaxDecel car

updateCarPosVel :: Float -> Car -> Car
updateCarPosVel deltaT car = car {cPos = (calcPos deltaT car), cVel = (calcVel deltaT car)}

calcPos :: Float -> Car -> Float
calcPos deltaT car = x + v*deltaT + (0.5)*(a)*(deltaT^(2 :: Int))
  where
    x = cPos car
    v = cVel car
    a = cAccel car

calcVel :: Float -> Car -> Float
calcVel deltaT car = max 0 (v + a*deltaT)
 where
  v = cVel car
  a = cAccel car
```

7.3 Config.hs

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}

module Config (SimConfig(..),
               parseConfigFile,
               getSceneConfig,
               setSceneConfig,
               getSpawnerConfig
               ) where

import Control.Exception (Exception, throwIO)

import Car (CarConfig(..), CarState(..))
import Scene (SceneConfig(..), SpawnerConfig(..))

import GHC.Generics (Generic)
import Data.Aeson (FromJSON, decode, parseJSON, withObject, (.:))
import qualified Data.ByteString.Lazy as B

-- "Parsable" Wrapper types to allow JSON Parsing --

-- State of a single car
newtype ParsCarState = ParsCarState {unwrapCarState :: CarState}
  deriving (Show, Eq)

instance FromJSON ParsCarState where
  parseJSON = withObject "Car-State" $ \v -> fmap ParsCarState $ CarState
    <$> v .: "Position"
    <*> v .: "Velocity"
    <*> v .: "Acceleration"

-- Base config for a car
newtype ParsCarConfig = ParsCarConfig {unwrapCarConfig :: CarConfig}
  deriving (Show, Eq)

instance FromJSON ParsCarConfig where
  parseJSON = withObject "Car-Config" $ \v -> fmap ParsCarConfig $ CarConfig
    <$> v .: "Desired-Speed-(m/s)"
    <*> v .: "Minimum-Gap-(m)"
    <*> v .: "Desired-Time-Gap-(s)"
    <*> v .: "Max-Deceleration-(m/s^2)"
    <*> v .: "Max-Acceleration-(m/s^2)"

-- Car spawning configuration
newtype ParsSpawnerConfig = ParsSpawnerConfig {unwrapSpawnerConfig :: SpawnerConfig}
    deriving (Show, Eq)

instance FromJSON ParsSpawnerConfig where
    parseJSON = withObject "Spawner-Config" $ \v -> do
        pRandomSeed <- v.: "Random-Seed"
        pSpawnRate <- v.: "Spawn-Rate-(steps)"
        pSpawnSpeed <- v.: "Spawn-Speed-(m/s)"
        configFreqs <- v.: "Config-Frequencies"
```

```
        parsCarConfigs <- v.: "Car-Configs"

        let carConfigs = map (unwrapCarConfig) parsCarConfigs

        let spawnerConfig = SpawnerConfig pRandomSeed pSpawnRate
        pSpawnSpeed configFreqs carConfigs

        return $ ParsSpawnerConfig $ spawnerConfig

-- Initial scene confguration
newtype ParsSceneConfig = ParsSceneConfig { unwrapSceneConfig :: SceneConfig }
  deriving (Show, Eq)

instance FromJSON ParsSceneConfig where
  parseJSON = withObject "Initial-Scene" $ \v -> do
    pMaxPosition <- v.: "Max-Position"
    pStepTime <- v .: "Step-Time-(s)"
    pLanes <- v.: "Lanes"
    initCarConfig <- v.: "Initial-Car-Config"
    initCarCount <- v.: "Initial-Car-Count"

    let carConfig = unwrapCarConfig initCarConfig
    let sceneConfig = SceneConfig pMaxPosition pStepTime pLanes carConfig initCarCount

    return $ ParsSceneConfig $ sceneConfig

-- Overall simulation configuration (no need for non-parsable version)
data SimConfig = SimConfig {
  simSceneConfig :: ParsSceneConfig,
  simSpawnerConfig :: ParsSpawnerConfig
} deriving (Show, Eq, Generic)

instance FromJSON SimConfig

-- Set the SceneConfig attribute in a SimConfig
setSceneConfig :: SimConfig -> SceneConfig -> SimConfig
setSceneConfig simConfig sceneConfig = simConfig {simSceneConfig = newParsScConfig}
 where
  newParsScConfig = ParsSceneConfig sceneConfig

-- Retrieve the SceneConfig from a SimConfig
getSceneConfig :: SimConfig -> SceneConfig
getSceneConfig simConfig = unwrapSceneConfig $ simSceneConfig simConfig

-- Retrieve the SpawnerConfig from a SimConfig
getSpawnerConfig :: SimConfig -> SpawnerConfig
getSpawnerConfig simConfig = unwrapSpawnerConfig $ simSpawnerConfig simConfig


data ParseError = ParseError String deriving (Show)
instance Exception ParseError

-- Parse the config file
parseConfigFile :: String -> IO SimConfig
```

```
parseConfigFile filePath = do
  fileContent <- B.readFile filePath
  case decode fileContent of
    Just config -> return config
    Nothing -> throwIO $ ParseError $ "Invalid JSON format in: " ++ (show filePath)
```

7.4 Scene.hs

```
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module Scene (Scene, SceneConfig(..), SpawnerConfig(..),
generateChunks, updateScene,
updateConsecPairsSkipLast,updateCarsChunk, initScene) where

import Car (Car(..), CarConfig(..), initCar, initCarState,
updateCarPosVel, updateCarAccel, carWithinBound)

import Data.List (sort)
import Control.Parallel.Strategies (parMap, NFData, rdeepseq)
import GHC.Generics (Generic)
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BSC
import System.Random (randomR, mkStdGen)

-- Configuration for car spawner
data SpawnerConfig = SpawnerConfig {
   spRandomSeed :: Int,
   spSpawnRate :: Int,
   spSpawnSpeed :: Float,
   spConfigFreqs :: [Float],
   spCarConfigs :: [CarConfig]
} deriving (Show, Eq)

type SceneChunk = [Car]

-- Configuration for initial scene construction
data SceneConfig = SceneConfig {
     scBound :: Float,
     scStepTime :: Float,
     scLanes :: Int,
     scInitConfig :: CarConfig,
     scInitCarCount :: Int
     } deriving (Show, Eq)

-- Datatype representing a lane of cars processed up until a given timestep
data Lane = Lane {
     lnCars :: [Car],
     lnStep :: Int,
     lnMaxCarId :: Int, -- ID's are per-lane
     lnSeed :: Int,
     lnID :: Int
} deriving (Show, Generic, NFData)

-- Datatype representing the full scene
data Scene = Scene {
     sLanes :: [Lane]
 } deriving (Show, Generic, NFData)

-- Initialize a lane with given boundary, car count, and initial car configuration
initLane :: Int -> Int -> Float -> Int -> CarConfig -> Lane
initLane laneID startSeed bound carCount carConfig = Lane initCars 0 carCount startSeed laneID
```

```
 where
    initCars = sort $ map (initCarFunc) [1..carCount]
    initCarFunc index = initCar index carConfig (initCarState (posStep * (fromIntegral index))
    posStep = bound / (fromIntegral carCount)

initScene :: Int -> SceneConfig -> Scene
initScene startSeed sceneConfig = Scene $ map genLane [0..numLanes]
 where
    numLanes = (scLanes sceneConfig)
    genLane laneID = initLane laneID (startSeed*laneID) (scBound
    sceneConfig) (scInitCarCount sceneConfig) (scInitConfig
    sceneConfig)

updateScene :: SceneConfig -> SpawnerConfig -> Bool -> Scene -> (Scene, B.ByteString)
updateScene sceneConfig spawnerConfig noDisplay scene = (nextScene, currBytes)
 where
    nextScene = Scene nextLanes
    currBytes = BSC.concat currLaneBytes

    (nextLanes, currLaneBytes) = unzip $ parMap rdeepseq updateLaneFunc currLanes

    updateLaneFunc = updateLane noDisplay sceneConfig spawnerConfig
    currLanes = sLanes scene

-- Updates a single lane and returns the bytestring representation of the current lane
-- and the next lane
updateLane :: Bool -> SceneConfig -> SpawnerConfig -> Lane -> (Lane, B.ByteString)
updateLane noDisplay sceneConfig spawnerConfig lane = (nextLane, currLaneBytes)
 where
    nextLane = Lane nextCars nextStep nextMaxID nextSeed (lnID lane)
    nextStep = (lnStep lane) + 1

    currLaneBytes =  laneIDBytes `BSC.append` currLaneBytesRaw
    laneIDBytes = BSC.pack ("\n{LaneID = " ++ (show (lnID lane)) ++
    ", Step = " ++ (show (lnStep lane)) ++ "}")

    (nextCars, currLaneBytesRaw) = processChunks noDisplay deltaT chunks
    deltaT = scStepTime sceneConfig

    chunks = generateChunks 150 $ newCars ++ [dummyCar]
    dummyCar = initCar (-1) (CarConfig 0 0 0 0 0) (initCarState  100
    10000000.0) -- really big so definetly out of the way

    (nextMaxID, newCars)
        | shouldSpawn = spawnNewCar (lnMaxCarId lane) nextSeed
        (spSpawnSpeed spawnerConfig) (spCarConfigs spawnerConfig)
        (spConfigFreqs spawnerConfig) filteredCars
        | otherwise = (lnMaxCarId lane, filteredCars)
    nextSeed = (lnSeed lane) + 1
    shouldSpawn = nextStep `mod` (spSpawnRate spawnerConfig) == 0
    filteredCars = filterCars (scBound sceneConfig) (lnCars lane)


-- Spawn new car, chosing config based on random seed
```

```haskell
-- Returns new list and new car's ID
spawnNewCar :: Int -> Int -> Float -> [CarConfig] -> [Float] -> [Car] -> (Int, [Car])
spawnNewCar currMaxID seed startSpeed carConfigs weights cars
        | null carConfigs || null weights || length carConfigs /=
        length weights  = (currMaxID, cars)
        | otherwise = (nextID, newCar : cars)
         where
            nextID = currMaxID + 1
            newCar = initCar nextID randConfig (initCarState
            startSpeed 0)

            randConfig = selectByWeight rand weights carConfigs

            (rand, _) = randomR (0, totalWeight) gen -- Generate a
            random number within the range of total weights
            gen = mkStdGen seed
            totalWeight = sum weights

-- TODO test these
-- Helper function to select an element based on the random number and weights
selectByWeight :: Float -> [Float] -> [a] -> a
selectByWeight rand weights options = selectByWeightHelper rand (scanl1 (+) weights) options

selectByWeightHelper :: Float -> [Float] -> [a] -> a
selectByWeightHelper _ _ [] = error "Empty list, this should not happen"
selectByWeightHelper rand (w:ws) (x:xs)
  | rand <= w  = x
  | otherwise = selectByWeightHelper rand ws xs



-- Take in a bound and filter all cars beyond that bound
filterCars :: Float -> [Car] -> [Car]
filterCars bound cars = filter (carWithinBound bound) cars

processChunks :: Bool -> Float -> [SceneChunk] -> ([Car], B.ByteString)
processChunks noDisplay deltaT chunks = (concat cars, B.concat byteStrings)
  where
    chunkResults = parMap rdeepseq (processChunk noDisplay deltaT) chunks
    (cars, byteStrings) = unzip chunkResults


-- Returns a chunk with one fewer cars
processChunk :: Bool -> Float -> SceneChunk -> (SceneChunk, B.ByteString)
processChunk noDisplay deltaT cars = (nextCars, currBytes)
 where
    nextCars = updateCarsChunk deltaT cars
    currBytes
        | noDisplay = BSC.pack []
        | otherwise = BSC.pack $ show (init cars)

updateCarsChunk :: Float -> [Car] -> [Car]
updateCarsChunk deltaT chunk = map (updateCarPosVel deltaT)
(updateConsecPairsSkipLast updateCarAccel chunk)
```

```
updateConsecPairsSkipLast :: (a -> a -> b) -> [a] -> [b]
updateConsecPairsSkipLast _ [] = []
updateConsecPairsSkipLast _ [_] = []
updateConsecPairsSkipLast update (x:xs) = update x (head xs) :
updateConsecPairsSkipLast update xs


-- Generate chunks of size n from a list, with one element overlap between each chunk.
generateChunks :: Int -> [a] -> [[a]]
generateChunks _ [] = []
generateChunks n xs
    | length xs > n = take n xs : generateChunks n (drop (n - 1) xs)
    | otherwise = [xs]
```

7.5 Sim.hs

```
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module Sim (runSimLoop, simResultToByteString, SimResult(..), SimArgs(..)) where
import Scene (Scene, SpawnerConfig(..), updateScene, initScene)

import Control.Parallel.Strategies (NFData)

import Config (SimConfig(..), getSceneConfig, setSceneConfig, getSpawnerConfig)
import Control.DeepSeq (rnf)
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BSC

-- Arguments passed in via command line (not JSON)
data SimArgs = SimArgs {
  numIters :: Int,
  noDisplay :: Bool
}

-- The result of a full simulation
data SimResult = SimResult SimConfig [B.ByteString]

-- Convert simulation result to a bytestring for file writing
simResultToByteString :: SimResult -> B.ByteString
simResultToByteString (SimResult config sceneBytes) = (BSC.pack (show config)) `BSC.append` (BSC

instance NFData SimResult where
    rnf (SimResult _ scenes) = rnf scenes


-- Runs simulation and returns printable result
runSimLoop :: SimArgs -> SimConfig -> SimResult
runSimLoop simArgs simConfig = SimResult simConfig byteStringResult
    where
        byteStringResult =  take (numIters simArgs) sceneGenerator

        sceneGenerator = buildByteResult initialScene updateFunc

        initialScene = initScene (spRandomSeed spawnerConfig) sceneConfig
        updateFunc = updateScene sceneConfig spawnerConfig (noDisplay simArgs)

        sceneConfig = getSceneConfig simConfig
        spawnerConfig = getSpawnerConfig simConfig


-- Builds infinite list of simulation byte result
-- Takes update function that generates the next scene and converts the current scene to bytes
buildByteResult :: Scene -> (Scene -> (Scene, B.ByteString)) -> [B.ByteString]
buildByteResult currScene updateFunc = currBytes : (buildByteResult nextScene updateFunc)
 where
    (nextScene, currBytes) = updateFunc currScene
```