Parallel Programming, CSCI 4320
Project Report: City Traffic Discrete Event Handler
Due Friday April 26th, 2019

Alex Garten (gartea@rpi.edu) and Shoshana Malfatto (malfas@rpi.edu)

1 Introduction

Our goal was to implement a parallel discrete event simulator which models vehicle traffic patterns to reveal which traffic rules result in the shortest trips between a location A and location B, in a large grid modeled after a city. Though implementations of traffic pattern DES already exist, we will use this as an opportunity to learn how to create a DES, since they have many applications.

2 Background

Our knowledge of parallel discrete event simulators (PDES) comes originally from the course lecture. Parallel DES help represent large-scale systems that are hard to understand, and which may require long run times if represented in real-time. A parallel DES should significantly reduce the run time. For instance, a simulation of air traffic could take a few minutes instead of a several years by looking at only specific events that are deemed important, and looking at them in parallel when possible. This could be applied to many other topics like communication networks or biology.

In class we talked about Rensselaers Optimistic Simulation System (ROSS) which runs Time Warp, a protocol that increases parallelism through rollback and anti-message mechanisms. This protocol scales very well in performance tests on RPI's Blue Gene/Q. We could have chosen to use ROSS to build our implementation, but we decided to create our simulator from scratch.

3 Implementation

We used a conservative approach to build our parallel DES, where cars can stop at a discrete number of locations, and at most one movement can occur within a tick. In our large city, which is a grid of streets going either north-south or east-west, cars only need to know about the cars near them, so traffic can run somewhat independently in different regions. However, synchronization is required to accurately move cars across the borders of their region.

For parallelization we used MPI, which we preferred to relying on threads after seeing the performance of both parallezat in Assignment 4/5. We used a side length for our grid of 32,768 blocks. For the full number of rows, this is multiplied by 2 so that even rows correspond to east-west streets, and odd rows correspond to north-south streets. The rows get divided up by the number of ranks, so that there needs to be communication with "ghost rows" like in homework 4/5.

Each street is a struct that contains an array for the lane going in one direction (east or south) and the lane going in the opposite direction (west or north). Each lane can hold 4 cars at one time. This resembles a queue, which is a commonly seen data structure in discrete event simulators.

There are intersection structs for each place where four streets should meet. This makes moving cars cleaner for the programmer to implement and read.

The beginning of the simulation starts with cars being randomly generated at locations on the grid, with random destinations. These locations are represented by a row index, a column index, and a street index which corresponds the eastern or southern direction.

At each step in time, or 'tick', ghost rows are sent between ranks. Then intersections are updated according to the specified rules. Cars try to turn in a direction that will get them closer to their destination, but they must follow the rules of the road and not crash into other cars. For instance, if the car at the northern side of the intersection goes straight south, the car on the eastern side of the intersection can't go any direction but right within that tick of time.

4 Performance Results

ROSS user documentation: https://ross-org.github.io/

5 Analysis of Performance

6 Conclusion

7 References

ROSS simulator paper: https://www.cs.rpi.edu/ chrisc/COURSES/PARALLEL/SPRING-2019/papers/sequoia-ross-pads-2013.pdf