15-853: Spacial Decomposition and Parallelization in Autosim

Alexei Colin April 19, 2013

1 Introduction

Within the upcoming decades autonomous vehicles promise to revolutionize transportation. As the robotic technology behind the driverless operation matures, challenges at the next level of abstraction become apparent: How should large numbers of autonomous cars cooperate? How should they interact with legacy cars? What roles should be delegated to the road infrastructure?

A viable approach for investigating such questions is simulation of interacting collections of vehicles. Autosim is an open-source simulator (developed at CMU[1]) for the design and evaluation of behavioral protocols based on vehicular communication. The behaviors and state of the cars are controlled by models associated with each car, such as the mobility model, or the communication model. At each time step, the state of each car is updated by evaluating each of its models. Figure 1 shows the front-end of the simulator.

Autosim uses real-world maps annotated with higher-level data, such as lane markings, and road signs. The simulator supports hybrid mode in which simulated cars are combined with replicas of real cars, where the latter are based on measured data from the real world. Hence, it can be used for simulating the behavior of an autonomous vehicle among legacy cars in real-time.

Simulator performance is a key concern. Since city-wide simulations with dense traffic are of particular interest, computation cost should scale gracefully with the number of vehicles. Furthermore, performance is a functional requirement for hybrid simulations, in which data from real vehicles places real-time constraints on the simulated ones. The current version of Autosim is limited to small-scale simulations due to an unoptimized and sequential implementation. With the goal of enable larger simulations, in this project we applied spatial decomposition through a Quad-Tree and parallelization techniques through a map-reduce paradigm.

The code repository for this project can be accessed from within the CMU network at `git://rtml-drkpc1.ece.cmu.edu/autosim.git`.

[1]While it is a project in my research group, it is completely unrelated to my research, which is in power-aware CPU scheduling.

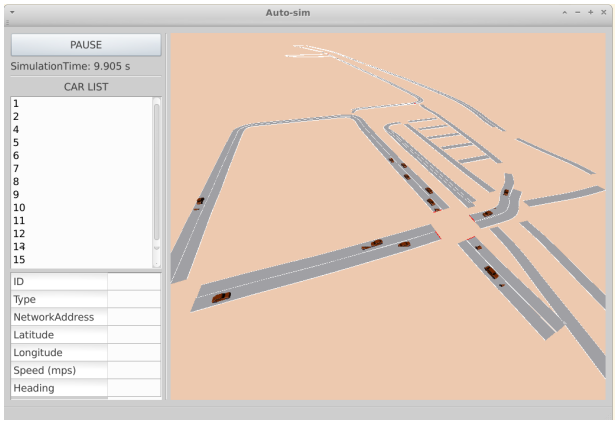

Figure 1: Front-end of the Autosim simulator

2 Approach

2.1 Spatial decomposition

Some computations performed at each simulation step concern only nearby cars within some region and, thus, naturally benefit from spacial decomposition. In this project we focused on the following computations:

- **message exchange**: cars communicate in order to coordinate intersection crossing or communal behaviors, such as follow-the-leader; each car broadcasts messages which reach only cars within a radius depending on the transmitter
- **collision detection**: each car updates its position based on a kinematic model without any regard for other cars; hence, in order to detect malfunction of interaction protocols, the simulator needs to check for collisions at each step as well as when a new car is inserted into the world

We replaced the $O(n^2)$ implementations for each of the above tasks with an efficient range-query algorithm based on a quad tree.

A quad tree was chosen over a simple grid-based cell structure because in real road networks traffic is clustered. In particular, intersections determine clustering. Choosing a good cell size in a grid is not generally applicable. A quad tree, on the other hand, adapts to the clusters. The question of how other spacial decomposition trees compare to quad trees is left as future work. In practice, it was found that even at moderate scale Autosim is very sensitive to constant factors, which would guide the choice.

We next describe the two major challenges that did not arise in the nearest-neighbor assignment and our approach to them: maintaining the data structure current in precense of point movement and range searching.

2.1.1 Incremental updates

Cars in the simulator move at each time step, which means that for the range searches to be accurate, the entries in the data structure needs to be updated each time step. One option is to recompute the tree from scratch. An option with lower overhead is to remove and re-insert the points that left their regions. However, it is possible to do slightly better by observing that

- a car is likely to move within the same leaf region
- adjacent leaf regions, where a car ends up if it does leave, are likely to be siblings or at least share an ancestor a small number of levels above

These observations lead to an algorithm for incremental updating of the tree. Instead of re-inserting the removed point from the root, we start at its exparent and traverse up the tree until a containing region is found, into which we perform the insert. The observation implies that much of the time, we would not need to traverse many levels upward. Since the removed point might be the last one, as we traverse upward, we coallesce (unsplit) the children of any parents as long as all of them are empty.

2.1.2 Range search

In the Autosim application, the prevalent query consists in finding the cars that are within a certain range of another car or an infrastructure element, such as a traffic light. This is known as a range search and is distinct from the nearest neighbor query. For our application, we are only interested in circular ranges.

As suggested in previous section, a quad tree is expensive to maintain. However, we argue that it is still profitable to do so. The cost of maintaining the quad tree is amortized over multiple range search queries within one simulation time step. Each radio transmitter requires a query with a different range. Two others are collision detection and proximity check for the newly generated simulated cars. The amortization aspect encourages extensions to Autosim that require range search.

2.2 Parallelization

Within a simulation step, several tasks carried out over the cars are independent. In particular, physical model state can be updated in parallel followed by a parallel collision check. We have parallelized both using a map-filter-reduce framework provided by `QtConcurrent`.

Map size (m)	1000×1000
Road segments	≈ 20
Intersections	15
Vehicles	≈ 450
Communication range (m)	5-10
Collision range (m)	¡3

Figure 2: Experimental parameters (a subset).

While the incremental updates to the quad tree remain a sequential bottleneck, we observe that this cost is overshadowed by physical model updates and collision checks, both of which where successfully parallelized (data not presented here).

3 Evaluation

3.1 Experimental setup

The experiments where conducted in Autosim. Cars are continuously created and inserted into the world at times according to a Poisson distribution. A car stays in the world until it completes its path. The workload during the simulation consisted of the intersection crossing negotiation under the default protocols, default path planning, collision detection, and message exchanges between cars. Both experiments yielded similar results. We used a road network based on data collected from the Pittsburgh area (cloned and concatenated to form a larger map) and a synthetic grid road network. Results are similar, and we present only the results from the latter. Experimental parameters are given in Table 2.

Autosim configuration was extended to choose the car registry implementation variant and to control the number of threads in the threadpool used by `QtConcurrent`. We have also varied the number of leaf nodes in the quad tree.

We have measured the CPU-clock and the wall-clock time needed to simulate 120 s of virtual time. The cpu-clock time was done at thread-level for sequantial code and at process level for parallel code. Under parallel execution, the CPU time gives an indication of the work performed. Also, we counted the number of distance calculations made.

The measurements where performed using `clock_gettime` family of POSIX functions 3.4GHz/8GB machine running GNU/Linux. To ensure there are no extraneous bottlenecks, the

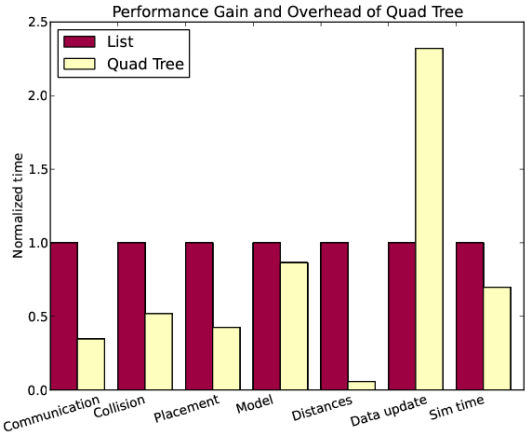

Figure 3: Performance gain and overhead of the range search via quad tree

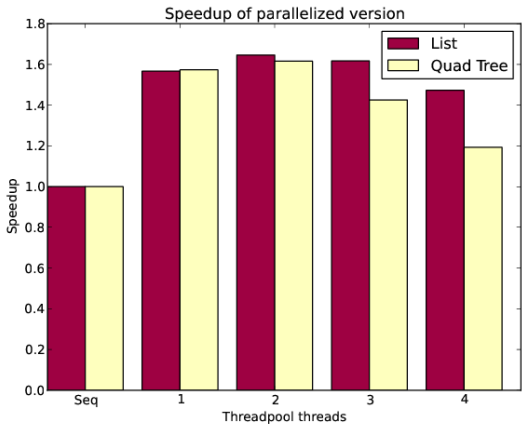

Figure 4: Speedup achieved by parallel implementation

visualization code was completely decoupled and a CLI-only compilation of Autosim was created for this evaluation.

3.2 Spacial decomposition

Figure 3 shows the benefit from a quad tree as well as its overhead as compared to the basic implementation using a generic container. It shows the measured execution times for the respective parts of the simulation workload.

The first three bars (Communication, Collision, Placement) correspond to the three tasks that where changed to use the range search query instead of an $O(n^2)$ loops. There objective of decrease execution time of these tasks was achieved.

'Model' bar corresponds to the physical model state updates for each car, which do not require range searching. It is shown to illustrate the subtasks for which execution time did not significantly changed. 'Distances' is a count of pair-wise distances that had to be computed (proportional to time). As expected, the quad tree is able to dramatically reduce the number of pair-wise comparissons.

The overhead of the quad tree is captured by 'Data update' bar. Compared to the cost of keeping a list up to date, a quad tree is more than twice more expensive. However, the proportion of time spent in maintainance of the data structure is small (e.g. less than 1% than the time spent in collision detection). Hence, it does not prevent the total simulation time (last bar) to improve.

3.3 Speedup

We next investigated the benefit of parallelization. Figure 4 shows the speedup achieved by the implementation where some independent subparts where parallelized as compared to its sequential algorithm. The major tasks parallelized are the updating of physical models for each car and the collision check.

Note that the same parallelized code utilizes the range search of List and Ouad Tree implementations of the car set. That is, in Figure 4 the emphasis is on comparing the speedups within one algorithm (instead of comparing List to Quad Tree).

Note that the thread counts do not include the main thread, but only the threadpool threads. Hence, the speed up for threadpool of thread one includes work done by the master thread.

The graph shows that for our implementation and the underlying Qt concurrency library, after two threads, more threads come with prohibitive overhead, which has a negative impact on the speedup. The same can be seen in the measured CPU time (which counts workload even when it is executed in parallel): cpu time increases fafter two cores (data not presented here).

We also note that both algorithms bnefit about equally from the parallelism because what is parallelized is largedy orthogonal to range search

4 Conclusion

Simulation of traffic is indispensible for development of next-generation vehicular coordination pro-

tocols. For example, the Autosim simulator has been successfully used for evaluating intersection protocols for autonomous vehicles. The original implementation, however, was limited by its reliance on $O(n^2)$ range searches and sequencial code.

We have addressed both limitations in this project. Range queries are now serviced using a spacial decomposition of the world map. An algorithm for incrementally updating the tree was developed and implemented. For further performance gain, collision detection and model updates have been parallelized and the overhead at higher thread counts was exposed.

Future work may include a investigating whether efficiency gain from a more advanced spacial decomposition tree can be realized.