Comparing the Efficiency of Traffic Simulations Using Cellular Automata

Fernando Díaz-del-Río[1,2(✉)], David Ragel-Díaz-Jara[1], María-José Morón-Fernández[1,2], Daniel Cagigas-Muñiz[1], Daniel Cascado-Caballero[1], José-Luis Guisado-Lizar[1,2], and Gabriel Jimenez-Moreno[1,2]

[1] Department of Computer Architecture and Technology, Universidad de Sevilla, Avenida Reina Mercedes s/n, 41012 Sevilla, Spain
fdiaz@us.es

[2] Research Institute of Computer Engineering (I3US), Universidad de Sevilla, Avenida Reina Mercedes s/n, 41012 Sevilla, Spain

Abstract. The shift toward electric vehicles requires the development of an extensive public electric charging infrastructure. With the aim of simulating hundreds of configurations for charging stations, street directions, crossing, etc., we need to find the best solution in short periods of time to predict and prevent traffic congestion. Thus, we study different models to discretize and manage vehicle movements using a synchronous cellular automata, with an emphasis in reducing the amount of (frequently accessed) memory and execution time, and improving the thread parallelism. This is guided by the classical lemma of computer architecture "make the common case fast", thus optimizing those code sections where most of the execution time is spent. Experiments carried out for microscopic traffic simulations indicate that compiled languages increase run-time efficiency by more than 70×. Then several strategies are studied, such as storing future velocities of each vehicle so that neighbor vehicles can benefit from this information. Using a single 12-core PC, we get to a total run-time for a unidimensional simulation that is very close to that reached by supercomputers composed of thousands of cores that use interpreted languages. This may also greatly reduce the energy consumed. Although some performance degradation may occur when complex situations are introduced (crossroads, traffic lights, etc.), this degradation would not be significant if the length of the streets were large enough.

Keywords: traffic modeling · cellular automata · computer parallelism · microscopic traffic simulation

Supported by Grant TED2021-130825B-I00 funded by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR".

J.-L. Guisado-Lizar et al. (Eds.): SIMUtools 2023, LNICST 519, pp. 182–194, 2024.
https://doi.org/10.1007/978-3-031-57523-5_14

1 Introduction

The transportation sector is responsible for a substantial share of the total equivalents of carbon dioxide CO_2 released into the atmosphere (see data for the European Union in[1]). These greenhouse gas emissions are the cause behind the current global climate crisis, which poses a formidable challenge to humanity. Within transportation, road vehicles, which play a major role in our daily mobility, are a significant contributor to this phenomenon. In response to this pressing issue, a shift toward electric vehicles (EVs) is being carried out at a global level. This electrification of road transportation requires the development of an extensive public charging infrastructure. Forecasts for the year 2030 indicate a significant surge in the demand for public charging stations, far surpassing the current figures. For example, a study conducted by the International Energy Agency indicates that electric vehicles are expected to constitute approximately 55% of all transportation modalities in Europe by 2030 in the scenario of existing policies, taking into account different types of vehicles [2].

It is important to optimize the distribution of urban roads in a city to avoid them contributing to increase traffic congestion.

In this work, we study different models to discretize the movement of vehicles using a synchronous cellular automaton (CA), with an emphasis on the efficiency with respect to reducing both the amount of memory that is to be frequently accessed and the execution time, and to improve the thread parallelism. To do this, we take into consideration the classical lemma of computer architecture "make the common case fast", thus concentrating and optimizing those code sections where it is spent most of the execution time.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents several ways to simulate and represent vehicle movements and to understand how to make simulation efficient. Section 4 summarizes the definition of data structures to produce an efficient simulation. The results are presented and discussed in Sect. 5. Finally, Section 6 presents the conclusions and future research directions.

2 Related Works

Modern traffic simulation applications focus on microscopic models that simulate the movements of individual vehicles [1]. These models are capable of reflecting various phenomena observed in reality. However, the simulation of detailed models poses a performance challenge that can be solved by using HPC (High Performance Computing) systems or optimizing the part of the code where more than 90% of the execution time is spent.

One of the most well-known microscopic models is the Nagel-Schreckenberg model [6], implemented for parallel execution. The authors tuned the algorithms

[1] https://www.europarl.europa.eu/news/en/headlines/society/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics).

for particular processor architectures, achieving good results for different specific computers [5].

Other important findings on parallelization can be found in [10]. The performance analysis of TRANSIM [9] identifies some issues that need to be addressed to achieve good scalability on a large number of processors when information is exchanged between spaces after each time step. GPU implementations are a more recent approach. The problem is that such simulations may not be well suited to GPU or SIMD kernels, due to the inherent random memory accesses, as some authors point out [11].

Other works that parallelize the traffic simulator are: [4], where authors reached a speedup of about 3 using a distributed and synchronized package on four PC computers; [7], where speedup was about 5 using 16 processors; and [3], where a 16 computing nodes (12 cores each) maintain a good speedup up to six nodes, but get stuck for more nodes due to intensive communication.

The above parallel simulators have been written in C, C++ and Java, using OpenMP or native distribution protocols. Only a few papers were written using other languages such as Erlang [12]), which is designed for massively concurrent and asynchronous applications, thus achieving very good scalability on large HPC systems.

3 Representing Vehicle Movements

The classical Nagel-Schreckengberg (Na-Sch) model reproduces the movement of vehicles using a synchronous cellular automata (CA), where space and time are discretized. A common way to simplify the representation of a two-dimensional map is to render and compact it into a one-dimensional vector, where each element (or CA cell) represents an area of a few meters. The vector must contain some special cells to represent crossroads and bifurcations (which must point to at least two cells). However, the number of crossroads and bifurcations is usually much lower than the total number of cells, which implies that speeding up the movement of vehicles along a one-dimensional vector is a critical task.

This model Na-Sch defines the vehicles variables with two pairs of values: The current and next positions of vehicle i ($1 <= i <= N$, N is the number of CA cells) are $x_i(t)$ and $x_i(t+1)$, and its current and next velocities $v_i(t)$ and $v_i(t+1)$. Each magnitude corresponds to time steps (t) and (t + 1).

The CA dynamic can be run in parallel and is determined according to the four following rules [6] for a certain vehicle i:

1. Acceleration: $v_i(t+1) = min(v_i(t)+1, v_{max})$;
2. Deceleration: $v_i(t+1) = min(d_i, v_i(t+1))$;
3. Randomization: $v_i(t+1) = max(v_i(t+1)-1, 0)$ (braking reduces velocity in one unit with probability P_b);
4. Movement: $x_i(t+1) = x_i(t) + v_i(t+1)$;

where v_{max} is the maximum speed that i vehicle can reach and d_i is the number of empty cells in front of the vehicle.

Note that these equations are obtained for a simple discretization of a moving particle with a bounded acceleration of one unit, where the third step (randomization) introduces a certain stochastic behavior in the model. Concretely, the most difficult and run-time consuming computing piece of the algorithm is the calculus of the ahead "free" distance d_i. This requires the search for empty cells in front of each vehicle, which implies a loop that necessarily scans the contiguous forward cells at least up to the reachable velocity of the vehicle.

In addition, special attention must be paid when using random numbers. Calling a good random function generator is very time-consuming: in fact, this call usually lasts more than the rest of the code devoted to each cell. High-quality uniform random numbers are not crucial for this kind of simulations: note that movement discretization is actually coarse, and the randomization of the third step is artificial. Therefore, it is preferable to generate previously a vector with a random number for each cell and simply access to one different element for each cell and time step during the simulation (see the explanation later).

The previous equations contain several variables that must be mapped into data structures with the double aim of being efficient and flexible enough to allow the introduction of several functionalities in the simulations. In this section, we proceed with a progressive optimization of the required structures.

First of all, one can part directly from declaring the most simple structures that allow us to run all the CA cell in parallel, and which correspond to the values that equations hold: two pairs of vectors for the current and next positions x_{curr} and x_{next}, and for the velocities v_{curr} and v_{next}, whose index represents the position on the map. As each cell needs to know the current state of its neighborhood to compute its next values, x_{curr} and x_{next} cannot be condensed into a single scalar vector.

Figure 1 represents 3 vehicles in a unidimensional CA, where the calculus of d_i for the two first vehicles and their resultant movements are depicted with arrows. Here, we assume that random braking does not occur. The upper table represents the simple solution that builds the two pairs of vectors.

Although a similar pair of velocity vectors can be the most straightforward solution, both can be condensed into one if we realize that next velocity can be stored in a local variable for each cell, which is stored at the end of a step can be as a last stage. That is, considering $v(t+1)$ as a local variable (note that the subindex i has been suppressed) and adding a new fifth rule that deletes the previous velocity and stores the new one. Additionally, the position vector only requires a few bits to indicate the presence of a vehicle in a cell. A mark must exist for occupied cells of position vectors, whereas the velocity magnitudes must reside in the velocity vectors. This new representation is depicted in the lower table of Fig. 1.

Going further, an obvious optimization in the classical model is the condensation of the 3 vectors into only two vectors containing current and next velocities, so that non-null cells in current velocity vector are also used as marks to compute d_i. Thus, Fig. 2 represents the same situation as Fig. 1 but using only two

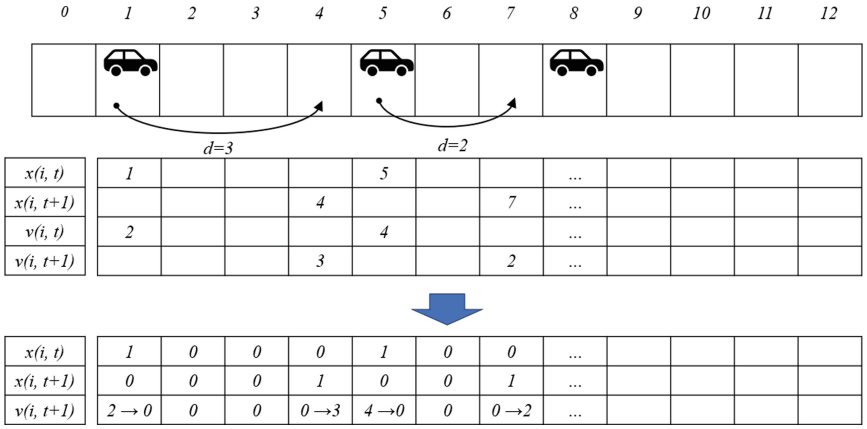

Fig. 1. Example of magnitudes to be computed in the classical Nagel Schreckenberg model for two vehicles.

vectors. Note that the mark in the free cell is now the value -1, because the value 0 must be used for vehicles with null velocity.

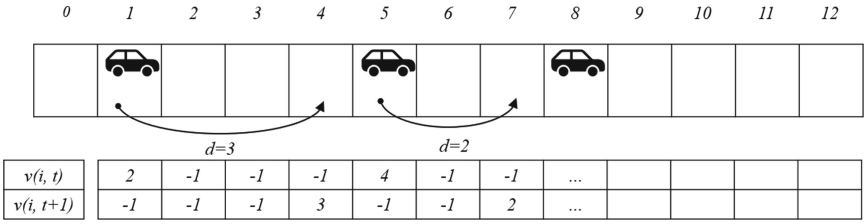

Fig. 2. Example of magnitudes to be computed in the optimized Nagel Schreckenberg model for two vehicles.

Another possibility of implementing this model comes from the use of a list that contains (at least) the positions of vehicles. However, the loop that searches for empty cells to compute d_i always implies the necessity of a vector with marks representing the occupied cells, because we cannot guarantee that the vehicle list will be ordered, and the search in a disordered list would be very inefficient.

Obviously, the combination of a list of vehicle coordinates plus a vector of free/occupied cells may be beneficial if the number of vehicles is very small. This is to be analyzed in Sect. 5.

Finally, the re-ordination of two iterations of the rules of the classical model may produce a faster execution (see Sect. 5). Since the most time-consuming part is the search lop that computes d_i, we can reduce the number of iterations of this loop by storing the future velocity of each vehicle. In addition, the future velocity stored in each cell is also the safety distance that must be fulfilled

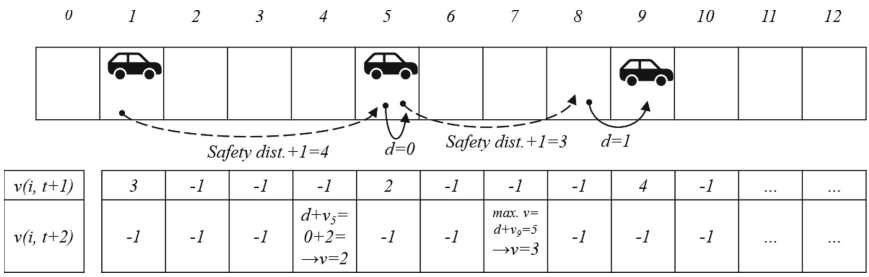

Fig. 3. Example of magnitudes to be computed in the novel model for two vehicles.

between vehicles. That is, we must ensure that the stored velocity value implies that the number of cells that are free in front of the vehicle is at least this same value. Thus, the search for the empty cell ahead can begin in the cell situated at the stored velocity plus 1.

Figure 3 represents this case for two vehicles: the first begins its search from cell 5 (because its stored safety distance is 3), where it encounters (in the first iteration) an occupied cell with a (safety distance) velocity of 2. Then its next velocity can be: the amount of cells (d in the figure) where the ahead vehicle was found plus the stored velocity of this ahead vehicle, that is, 0+2 = 2. The second vehicle can begin its search from cell 8 (because its stored safety distance is 2), and it encounters (in the second iteration) an occupied cell with a (safety distance) velocity of 4. Then its next velocity can be: the amount of empty ahead cells (d=1) plus the stored velocity of the vehicle ahead, that is, 1 + 4 = 5. However, the acceleration bound of the model must reduce this new velocity to 3 (one unit more than the previous one).

In summary, the CA dynamic rules for this novel model for a certain vehicle i are as follows:

1. Search (Deceleration) and Bounded Acceleration: $v(t+2)[i + v(t+1)[i]] = min(d_i^* + v(t+1)[i + v(t+1)[i] + d_i^*], v_{max})$;
2. Randomization: $v(t+2)[i] = max(v(t+2)[i] - 1, 0)$ (braking reduces velocity in one unit with probability P_b);

Where d_i^* is the number of empty cells ahead after the safety distance ($v(t+1)[i]$).

Note that we have used the notation $[i]$ to indicate the element $i-th$ of the velocity vector, while the two (current and next) velocities are expressed with $(t+1)$ and $(t+2)$.

4 Efficient Definition of Data Structures

According to the classical Na-Sch model, a complete description of the CA simulation requires up to four vectors with dimensions equal to the number of cells:

$v_i(t+1), v_i(t), x_i(t+1), x_i(t)$. Other quantities do not need to be stored in large data vectors because they are global (v_{max} or can be computed specifically for each cell (d_i, P_b). Besides, when developing a simulation with more complex behavior, e.g. crossroads, different characteristic of vehicles, traffic lights, routing preferences, etc., additional data structures must be inserted.

However, special care must be taken both to maintain the previous fast kernel of simulation and to avoid that complex structures play a significant role in the CA evolution. This is because the memory access locality is usually degraded when using complex structures. In the same way, it is preferable to compact the set of cells into a unique unidimensional vector (which favors memory locality and effective caching if neighboring streets were allocated in adjacent addresses) instead of scattering cells among dynamic lists (which may be allocated in disperse memory addresses). The reason for this decision, as demonstrated in Sect. 5, is that cell vectors would play the most important role in the simulation and probably consume most of the execution time. Thus, we are taking into consideration the lemma "make the common case fast".

Although cell vectors are believed to be a memory waste if the ratio between the number of vehicles and cells is low (around a 10% is generally considered to simulate dense and demanding traffic conditions), note that a complete set of cells simplifies the search for adjacent vehicles. This issue is a crucial, very common, and repetitive task to simulate vehicle movements. In contrast, a search for a smaller but disordered list of cells would be very time-consuming because the search has to be done for the complete disordered list and for each vehicle. Using a cell vector, only the indexes to the new structures (that is, acting like pointers) must be added and updated (as explained below).

In addition to cell vectors, other structures are needed to store the characteristics of vehicles and the shape of the streets that correspond to the city to be simulated. These structures can contain at least the following information to define vehicle and street data. For each vehicle: its routing preferences, its type, the cell where it is at the current step, and other necessary to emulate more complex behaviors, like its length, maximum speed, etc. For each street: the cell indexes where it begins and ends (actually, linear indexes to the cell vector), its maximum allowed speed, a set of indexes to relate this street with its neighboring ones), if it has priority when entering a crossroad, and other necessary to emulate more complex behaviors, like its type, priority changes due to traffic lights, and so on.

In summary, the basic vectors that control the speed of vehicles are two $(v_i(t+1), v_i(t))$; the other two $(x_i(t+1), x_i(t))$ are not necessary because the cell index is, in fact, the position $x_i(t)$. Instead of position, a pointer to the vehicle list (namely, $p_{vehlist}(t)$) for each cell must be updated at the end of each simulation step. Note that two vectors for the current (t) and next steps $(t+1)$ are not necessary to control vehicle movements. Thus, a minimal set of pointers that relate cell indexes with vehicle and street indexes is also required. Figure 4 represents the minimal set of structures and its relation using indexes among them. Note that Fig. 3 contains the minimal set of structures required

for efficient simulation, thus reducing the amount of memory that is frequently accessed.

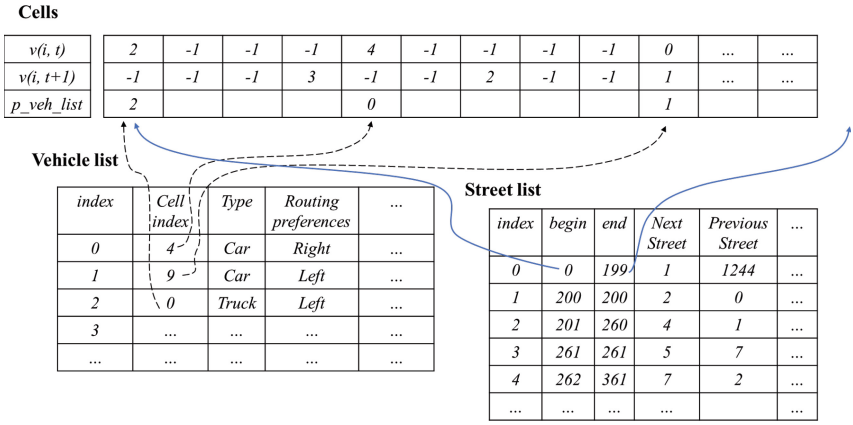

Fig. 4. Minimal set of structures required for an efficient simulation and their main relations.

In the next Section we compare the run-time execution of the different algorithms to get to that with better results. Additionally, we introduce and discuss additional features in the baseline model, to measure the influence of these features on the final execution time.

5 Results and Discussion

The results have been conducted thoroughly to conclude which performance improvement can be reached when changing the CA traffic model. We consider the execution of the classical Na-Sch model as the baseline time, and we proceed to determine the acceleration when parallelizing several cores and when comparing it with different optimizations and additional features.

Before trying to reach extreme performance and making optimizations, a first comparison was made between languages for the classical Na-Sch model. Being Python one of the most common languages nowadays, it is discovered that the algorithm written in this language is around 70× slower than the same algorithm written in C++ for a smaller number of cells (64 Ki cells[2]) when running in only one core. Exactly, a total of $2.89639e+07$ movements can be run in a second for the C version in contrast to $4.14538e+05$ reached in Python. The situation is even worse for larger sizes (80× slower for 128 Ki cells). Going further, speedup for the version in C++ is easily achieved using OpenMP, then accelerating a number of times near the number of cores (see next results). This result is even greater than that found as the mean for scientific codes (around 40× slower; see [8]).

[2] Ki is the multiplier for 1024×.

Different configurations were also analyzed for the several tests in two stages. In a first stage, we determine the most important parameters and fast models using a simple simulation, mainly consisting of unidimensional vectors. Values for a medium-sized city of a million inhabitants have been used: 5000 streets of around 200 m each, that is, a total of 10^6 m, approx. A common selection for the size of the simulated cell is 4 m, which results in a number of cells of 25×10^4 cells.

In the second stage, we concentrate on the model that yields the best results and compare its performance with that of baseline time for a realistic simulation and for different values of the most important parameters, which were selected in the first stage. These parameters are mainly: *a*) ratio of the number of vehicles to the number of cells; *b*) maximum speed.

Throughout the simulations, the different values of the parameters have been chosen to be similar to those in the current literature, except when indicated. An 8% for the ratio of the number of vehicles by the number of cells; for the maximum velocity values, the values are 4, 8 and 12 cells per simulation step. Note that the maximum velocity for cities is around 50 Km/h (15 m/s). If a simulation step counts for a second, this means 4 cells per step. Bigger simulated velocities would mean a finer description of movement, that is, fewer meters per cell. Random braking probability is fixed to 0.10; a common value in most Na-Sch simulations.

We also proceed with two computers (one laptop and a modern desktop PC) to demonstrate whether the machine has an influence on the results. The computers are an Intel Core i7-10750H, 2.60 GHz, 16.0 GB; and a 12th Intel Core i7-12700K, 3.60 GHz, 32.0 GB.

The codes were written in C++ and compiled with Microsoft Visual Studio 2022, allowing all speed optimizations. Each test is repeated ten times, and the minimum times are collected, because the first execution is always slower since the operating system is reallocating the executable (actually the mean times vary less than 5% than the minimum ones).

The main models to be considered are those represented in the three Figures of Sect. 3, that is, the classical Nagel-Schreckenberg, the optimized Nagel-Schreckenberg, and the novel model proposed here.

The main loop that controls the vehicle movement of these models can be two-fold. Obviously, the most basic loop executes one iteration for each cell. However, a second option consists of a loop with one iteration for each vehicle. In this case, we need to go through the vehicle list structure, read the cell index, which points to the corresponding cell where each vehicle is placed, where finally the movement is to be computed.

The main result is obtained by comparing the single-threaded execution with the parallel execution obtained with OpenMP. Speed-up values are very satisfactory even for the laptop, because they are close to the number of physical cores that the machine has. Consequently, and from now on, only results for the parallel OpenMP versions are shown.

Figure 5 shows the number of movements per second for the classical Nagel-Schreckenberg and the novel models when varying the number of cells from 128 Ki to 1024 Ki, and for different maximum velocities (4, 8 and 12 cells per step). Here, the probability of random braking is fixed to 0.1.

It is clear that the number of cells managed for medium cities has very little influence on the sizes that allow the cells to be allocated to the L3 cache. Of course, there is a small performance reduction for much larger sizes. Similar results are encountered for the other models.

Besides, running the same algorithm for different number of steps (bigger than 10) implies little variations in performance. This is evident because each time step is independent from the rest, and once the data vectors have been cached, each execution step does not suffer from important variations.

Therefore, from now on the number of cells to be simulated is that of a medium city (around 256 Ki) and the number of steps is fixed to 160 (a sensible number used, for example, in [12]). Because the achieved speedups are usually near the number of physical cores, OpenMP is active in all the tests.

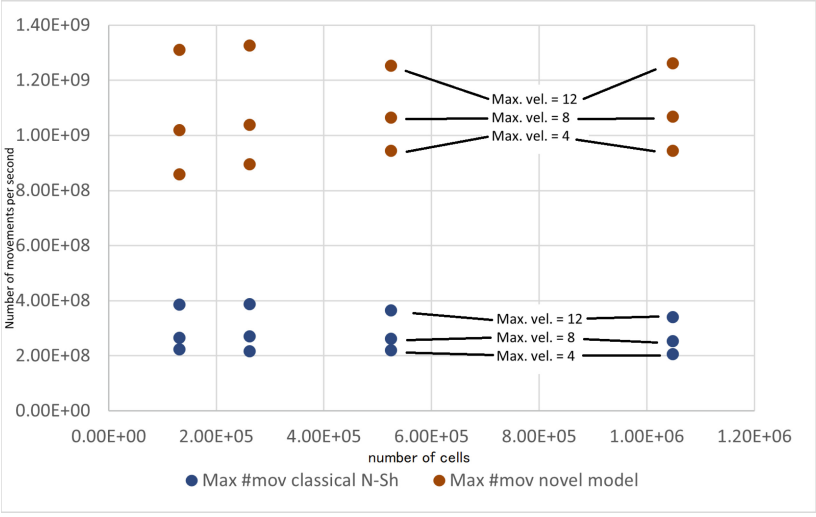

Fig. 5. Number of movements per second for the classical Nagel Schreckenberg and the optimized Nagel Schreckenberg models when varying the number of cells from 128 K to 1024 K. Maximum velocities are 4, 8 and 12 cells per step.

Previous Fig. 5 gives a first idea of the speedup obtained for the models of Sect. 3, which counts for different parameters. The next set of simulations compares the speedup obtained for the models in Sect. 3 for different parameters.

Table 1 shows the number of vehicle movements per second and the speed-up (which is the reference for the classic Na-Sch algorithm) when simulating on the modern PC a total of 160 steps and 256 Ki cells with a ratio of the number of

Table 1. Comparative speed-up using the number of vehicle movements per second for the Classic Na Sch, its optimized version, and the novel model proposed here when varying maximum velocity.

	Number of vehicle movements per second			Speedup	
Max. Veloc	Classic	Optim	Novel	Classic vs Optim	Classic vs Novel
4	2.28177e+08	4.68137e+08	7.09059e+08	2.05	3.11
8	2.17847e+08	4.41291e+08	7.60969e+08	2.03	3.49
12	2.06842e+08	4.06585e+08	8.67613e+08	1.97	4.19

Table 2. Number of vehicle movements per second for different models. r.r. means the reorder ratio of the vehicle list.

Ratio veh/cells	Na-Sch classic	Na-Sch optim	Novel model
5	3.90486e+08	7.58989e+08	1.32079e+09
10	2.73786e+08	5.42995e+08	8.68227e+08
20	1.96049e+08	3.82318e+08	5.9374e+08

Ratio veh/cells	Na-Sch classic using vehicle list (r.r. =0.1)	Na-Sch classic using vehicle list (r.r. =0.2)	Na-Sch classic using vehicle list (r.r. =0.8)
5	5.61475e+08	4.35744e+08	2.83068e+08
10	4.68071e+08	3.60048e+08	2.75381e+08
20	3.84744e+08	2.87258e+08	2.33214e+08

vehicles by the number of cells of 7% and different velocities. Similar speedups are encountered for the laptop. It is clear that the novel model outperforms the rest and that the larger the velocity, the more speedup is reached. This can be understood because the proposed novel algorithm scans for far fewer cells to find each vehicle ahead. In fact, if the vehicles formed a platoon, exactly the next cell (ahead of the stored future velocity) would contain a vehicle, and the scanning would have only one iteration (see Fig. 3). This is clearly more obvious when maximum velocities are greater. On the other hand, classic or optimized Na-Sch models need to scan at least as many cells as the current velocity.

The next interesting effect occurs when simulating different ratios between the number of vehicles and cells: 1 vehicle of 5, 10 and 20 cells is simulated and the results are presented in Table 2. Around a 10% is generally considered to simulate dense and demanding traffic conditions; 1 out of 5 is very dense traffic and 1 out of 20 emulates relaxed traffic conditions.

This table shows the number of movements per second for the following models: the classical Na-Sch, its optimized version, the novel model proposed here, and three tests that implement the classical Na-Sch model with a list of vehicles, so that the main loop is executed (and parallelized) for this list. These three tests vary the reorder ratio (r.r.) of the vehicle list (values r.r. =0.1, 0.2, 0.8), that is, before the simulation, the indexes where the vehicles reside are reordered to emulate the real case: Each entry in the vehicle list may point to

disperse positions of the cell vector. The rest of parameters are the same as in the previous table (except the maximum velocity fixed to 4 cell/step).

Several interesting conclusions are drawn from this table, which are enumerated in the next section.

6 Conclusions and Future Work

Experiments carried out for microscopic traffic simulations indicate that compiled languages increase run-time efficiency by more than 70×. In addition, a novel discretized vehicle movement model is proposed that reduces execution time by more than 3× when compared with the classical Nagel-Schreckenberg model. Putting all together, we get to a total run-time for a unidimensional simulation using a 12-core PC that is very close to that reached by supercomputers composed of thousands of cores that use interpreted languages. Other several interesting conclusions that yield from result tables are:

1. If the automata simulates less vehicles, the number of vehicle movements per second is lower for any model that manage cells. This can be understood because the main loop must go through the complete vector cell. If it contained fewer vehicles, there would be an overhead time spent in empty cells. Obviously, this effect is not so significant for the model that includes a vehicle list.
2. Simulations using a vehicle list behave poorer than optimized cases, even for low traffic densities. The effect of sparse accesses to the computer memory hierarchy introduces a very negative effect on lists.
3. As expected, the model using a vehicle list gets much worse when the list is reordered to emulate the above commented real case.
4. The novel model reaches an impressive performance (1.32079e+09 vehicle movements per second) in saturated traffic conditions (due to its reduced scanning task, as explained above). In this sense, we can remark that using only a common PC with 12 cores, we are reaching a performance that is very close to that of [12], which involves a huge amount of 19,200 computing cores. To be exact, in this paper, a total of 160 steps for 11.5M cars were simulated in one second, which means a total of $160 \times 11.5M = 1.84e + 09$ vehicle movements per second, similar to the results of Table 2. Therefore, the waste of energy when using a top HPC system instead of a common PC to get similar performance is enormous. We are aware that some performance degradation is going to occur when introducing more complex situations in the simulations (crossroads, traffic lights, etc.). However, one can estimate that this degradation would not be significant because the ratio between cells and crossing is usually high. For example, in [12] a total of 240K crossroads and 144M cells are simulated, which means $144M/240K = 600$ cells for each crossing.

Future work is twofold. First, making simulations even faster by comprising data cell information into a more reduced number of bits. This can promote

efficiency in other platforms like SIMD instructions, GPUs, etc. Second, studying different models of crossroads to make them also time-efficient.

References

1. Brilon, W., Wu, N.: Evaluation of cellular automata for traffic flow simulation on freeway and urban streets. In: Brilon, W., Huber, F., Schreckenberg, M., Wallentowitz, H. (eds.) Traffic and Mobility, pp. 163–180. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60236-8_11
2. International Energy Agency: Global EV Outlook 2023 (2023). https://www.iea.org/reports/global-ev-outlook-2023
3. Kanezashi, H., Suzumura, T.: Performance optimization for agent-based traffic simulation by dynamic agent assignment. In: 2015 Winter Simulation Conference (WSC), pp. 757–766. IEEE (2015)
4. Klefstad, R., Zhang, Y., Lai, M., Jayakrishnan, R., Lavanya, R.: A distributed, scalable, and synchronized framework for large-scale microscopic traffic simulation. In: Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005, pp. 813–818. IEEE (2005)
5. Nagel, K., Schleicher, A.: Microscopic traffic modeling on parallel high performance computers. Parallel Comput. **20**(1), 125–146 (1994)
6. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. J. Phys. I France **2**, 2221–2229 (1992). https://doi.org/10.1051/jp1:1992277
7. O'Cearbhaill, E.A., O'Mahony, M.: Parallel implementation of a transportation network model. J. Parallel Distrib. Comput. **65**(1), 1–14 (2005)
8. Pereira, R., et al.: Ranking programming languages by energy efficiency. Sci. Comput. Program. **205**, 102609 (2021). https://doi.org/10.1016/j.scico.2021.102609. https://www.sciencedirect.com/science/article/pii/S0167642321000022
9. Raney, B., Voellmy, A., Cetin, N., Vrtic, M., Nagel, K.: Towards a microscopic traffic simulation of all of Switzerland. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2329, pp. 371–380. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46043-8_37
10. Rickert, M., Nagel, K.: Dynamic traffic assignment on parallel computers in transims. Future Gener. Comput. Syst. **17**(5), 637–648 (2001)
11. Strippgen, D., Nagel, K.: Using common graphics hardware for multi-agent traffic simulation with cuda. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, pp. 1–8 (2009)
12. Turek, W.: Erlang-based desynchronized urban traffic simulation for high-performance computing systems. Future Gener. Comput. Syst. **79**, 645–652 (2018). https://doi.org/10.1016/j.future.2017.06.003. https://www.sciencedirect.com/science/article/pii/S0167739X17311810