International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 2–20, January 2011

Efficient Traffic Simulation Using Agents within the Global Cellular Automata Model

Christian Schäck, Rolf Hoffmann, Wolfgang Heenes
Technische Universität Darmstadt
FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, 64289 Darmstadt, Germany
{schaeck,hoffmann,heenes}@ra.informatik.tu-darmstadt.de

Received: June 30, 2010
Revised: October 30, 2010
Accepted: December 7, 2010
Communicated by Akihiro Fujiwara

Abstract

We present a mapping of the well known Nagel-Schreckenberg algorithm for traffic simulation onto the Global Cellular Automata (GCA) model using agents. The GCA model consists of a collection of cells which change their states synchronously depending on the states of their neighbors like in the classical CA (Cellular Automata) model. In contrast to the CA model the neighbors can be freely and dynamically selected at runtime. The vehicles are considered as agents that are modeled as GCA cells. An agent is linked to its agent in front, and an empty cell is linked to its agent behind. In the current generation t the position of an agent is already computed for the generation $t+2$. Thereby the agents movements and all cell updates can directly be calculated as defined by the cell rule. No searching of specific cells during the computation is necessary. Compared to an optimized CA algorithm (with searching for agents) the GCA algorithm executes significantly faster, especially for low traffic densities and high vehicle speeds. Simulating one lane with a density of 10% on an FPGA multiprocessor system resulted in a speed-up (measured in clock ticks) of 14.75 for a system with 16 NIOS II processors.

Keywords: Nagel-Schreckenberg, Global Cellular Automata, Multiprocessor Architecture, FPGA, multi-agent simulation

1 Introduction

We have described in [26] a general agent simulation system based on the GCA model (Global Cellular Automata) [11, 12] which can be used for a variety of different tasks to be solved with agents. The agent system is now used to model and simulate the Nagel-Schreckenberg [21] algorithm. Figure 1 shows the levels in the hierarchy of the agent simulation system: the agent system, the GCA model and the architecture. The object representation for each hierarchy level is shown in the right column of figure 1.

The agent simulation system consists of an agent system on top of the GCA model and on top of a certain GCA architecture (multiprocessor architecture (MPA) [25, 26, 27], a data parallel architecture (DPA) [8, 9, 10] or further architectures). The agent system uses GCA cells to model agents. Agents are mostly active entities moving and solving tasks while cell states are fixed at their

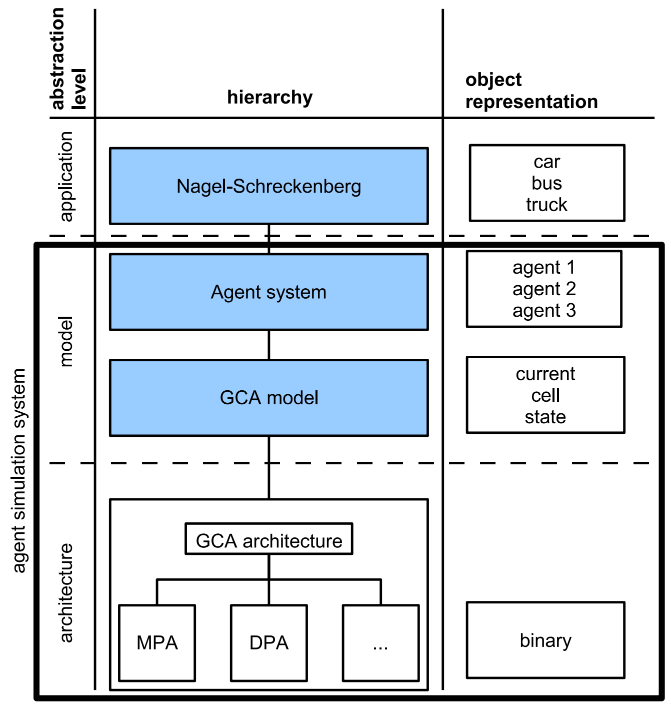

Figure 1: Abstraction levels: Application, agent system, GCA model, GCA architecture

location. An agent is modeled as a GCA cell state that can move in the GCA cell space. Moving of an agent is accomplished by deleting the cell state on the source position and creating it on the target position. Therefore, thinking about active agents (moving objects) is easier than thinking about copying cell states.

The GCA model as such has the advantage that it can easily be used to describe parallel algorithms [14] and that it can also be efficiently mapped onto different parallel architectures, e.g. DPA [8, 9, 10] or MPA [25, 26, 27]. A digital laboratory of agent-based highway traffic model has been investigated in [20] using the GCA model.

1.1 Related Work

Multi-agent modeling can be applied to a variety of disciplines, e.g. traffic simulation [17, 21, 22], evacuation simulation [4, 33, 36], autonomous aircrafts [16], simulation of biologic systems [29, 32] or ecosystems [5]. In previous work [26] we have shown how agent simulation can be described and simulated using the GCA (Global Cellular Automata) model [11, 12]. Meanwhile we have shown in a current work [28] how the architectures can further be improved to accelerate the simulation speed by the factor 4.8 for a different test application. Other investigations have incorporated the characteristics of CAs (Cellular Automata) to simulate HIV-Immune Interaction Dynamics [37].

Multi-agent simulations have also been realized on GPUs (Graphics Processing Unit) using the CUDA (Compute Unified Device Architecture) framework [30, 31]. The authors of [6] present a new technique to simulate agent-based models on GPUs. They reached a high simulation performance using sugarscape as a test application. But the authors criticized the bad programmability and denote it as counterintuitive. As there is a wide range of multi-agent applications, a more specialized and agent dedicated platform is desirable. As the agent behavior is mostly fine grained approaches using MPI (Message Passing Interface) are expected to generate too much overhead. Thus, approaches with low communication overhead with easy programmability are needed.

Using the GCA model the programmability can be kept intuitive. New applications can then be developed with low effort. The application developer does not need to care about parallelism or parallel execution. Parallelism is implicitly given by the amount of processors within the architecture. Many of the above mentioned applications (e.g. [4, 5, 21]) have either been developed using the CA model or can easily be mapped onto it due to its similarities. A dedicated multi-agent simulation platform based on the CA/GCA-model can therefore simulate all these applications efficiently and benefit from past researches based on cellular automata. Our architectures are implemented on FPGAs (Field Programmable Gate Array). Therefore the absolute performance will not be able to keep up with the highly advanced GPUs. Developing new architectures allows us to investigate specially designed commonly required agent functions. The most advanced architecture can then be build as an ASIC (Application Specific Integrated Circuit) to serve as an agent simulation platform for various applications.

The authors of [7] use an FPGA based cellular automaton to simulate crowd evacuation models. They state that the FPGA is advantageous in terms of low-cost, high-speed, compactness and portability features. They further mention, that the CA-model offers an easy-to-implement framework, providing a cell-centered programming style that forces the programmer to solve conflicts between concurrent agent's actions.

1.2 The Global Cellular Automata Model

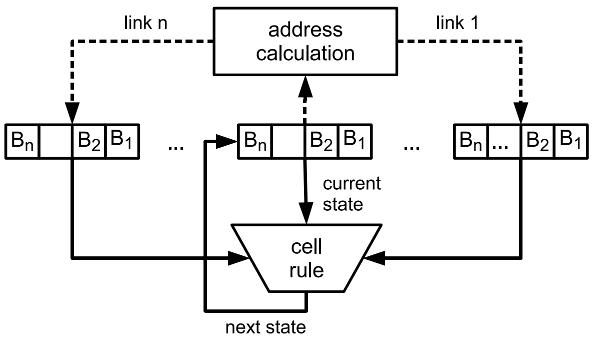

Figure 2: The GCA operation principle

The GCA model is a generalization of the Cellular Automata (CA) model using dynamic global links. The GCA model consists of a set of cells that update their state synchronously in parallel according to a local rule stored in each cell. Each cell can hold multiple data and link fields. Here we will not distinguish between data and link fields and call them *blocks*. Each cell consists of a configurable number of blocks. The term block describes a memory location holding any kind of data (data or link information). The interpretation of a block is determined by the application. For each cell a local cell rule is applied calculating the next block states. All cell states at a certain time step t constitute a so called *generation*. In the GCA model each cell has only read access to any other cell. Write conflicts cannot occur, therefore the model can easily be supported by hardware for a large number of cells. Figure 2 shows the operations principle for n blocks. The amount of blocks per cell can be adjusted as necessary.

1.3 Nagel-Schreckenberg Algorithm

The Nagel-Scheckenberg algorithm was introduced in [21]. The authors describe a stochastic discrete automaton model to simulate freeway traffic for a single lane. Since then, extensions for this model as well as other approaches have been published such as two-lane traffic models [17, 22], different driver behaviors [24] or 4-Way intersections [19]. In contrast to our work, [31] uses graphics processing units (GPU) and NVIDIAs CUDA framework to accelerate traffic simulation. As a generalization they also use the term *agent*.

The steps of the Nagel-Schreckenberg algorithm [21, p. 2] are:

1. If the maximum speed of a vehicle is not yet reached, increment the speed by one. The incremented speed is called *new speed*.
2. If the gap Δ to the next vehicle in front is less than the new speed, reduce it to the size of the gap.
3. Reduce the new speed by one with probability π.
4. Move all vehicles according to their new speed.

This algorithm is probabilistic, thus random numbers are needed for its implementation. To provide different and random behavior for the agents in the agent system, we have to define how random numbers can be covered by the GCA model. The movement of an agent to a random destination equals a copy and delete process in the GCA model. The source cell state has to be copied to the destination cell state and the source cell state has to be deleted afterwards. Thus, two cells have to behave consistently.

1.4 Random Numbers in the GCA Model

In order to move the cell state of a cell to another randomly selected cell, both cells need knowledge about the same random number. A global random number, accessible by all cells does not satisfy the need of most applications as it does not allow different random behaviors for each cell. To be able to have different random numbers for each cell, as well as global access to every random number, it is necessary that each cell holds its own random number. This random number has to be generated one generation in advance of its usage, because global access is only allowed for cell state variables, not for local variables being modified in the current generation.

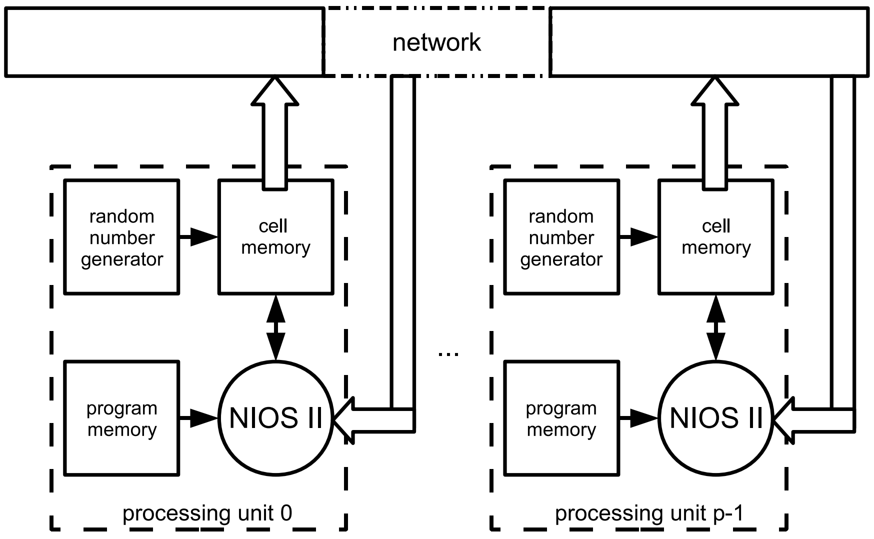

Figure 3: General GCA system architecture with data path and random number generator

2 Multiprocessor Architecture

2.1 Overview

The GCA multiprocessor architecture already presented in [26] was enhanced and will be used here for the evaluation of the speed-up of the Nagel-Schreckenberg algorithm. It consists of p NIOS II softcores and different selectable interconnection networks. It turned out that for agent simulations the network with dynamic bus arbitration performed best for agent simulations [26]. So it will be used here, too. Our system design was taking into account related work like bus architectures in [23], and another multiprocessor architecture with NIOS II softcores [18].

Each of the p NIOS II processors is supplied with a program memory and a cell memory (*Processing Unit* (PU)). The program memory holds the programmable cell rule. The cell memory holds the cell values. Each cell memory holds a subset of all cells. Each NIOS II processor applies the cell rule to the cell data in its associated cell memory. To be able to read neighbor data addressed by the links, all processors are connected to a network. Each PU has its own processor ID to be accessible and distinguishable from the other PUs. Write accesses to neighbor cells are not allowed according to the GCA model. The cell memories are implemented as dual port memories. The first port is used for read and write accesses by the associated processor, the second port is used for read accesses by other processors via the network. Read accesses within the PU are called internal, read accesses using the network are called external. The cell memories are capable of holding two generations at a time. The current generation is used for all read accesses while the next generation is used for all write accesses. Thereby data consistency is given at any time. The synchronization process will be explained in detail in section 2.5.

The GCA multiprocessor architecture has been enhanced with a random number generator using a linear feedback shift registers (LFSR) in each processing unit (PU) as shown in figure 3. A LFSR generates pseudo-random numbers and stores them in the highest available block of each cell. The random number is written automatically in parallel whenever the cell's state is updated (or any block of it). In the FPGA implementation, each block of a cell corresponds to a local memory block, therefore parallel access to all blocks is available. The random number generation does not cause any additional processing time, but requires additional memory space for each cell. Random numbers can only be read by the application. Integers instead of floats are used as random variables. They need much less hardware for generation and handling, and the accuracy can be adjusted by modifying the word length.

2.2 The NIOS II Softcore Processor

The NIOS II processor [2] is a general-purpose RISC processor core, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- Floating-point instructions for single-precision floating-point operations
- Custom instructions, as defined by the user

For the implementation of the multiprocessor architecture, the custom instructions [3] turned out to be very useful (Section 2.3).

2.3 Custom Instruction

We extended the NIOS II instruction set with a parametrized custom instruction. The custom instruction can be specified by one of the following functions ($Q \in \{0, ..., n-1\}$ for n blocks):

(1) RD_B{Q} read operation for block Q with automatic decision between internal and external accesses based on processor ID comparison

(2) EXT_RD_B{Q} external read operation for block Q

(3) INT_RD_B{Q} internal read operation for block Q

(4) WR_B{Q} write operation for block Q

(5) NEXTGEN processor synchronization command (Section 2.5)

The automatic read function (1) decides between an internal or external read access by comparing the address with the processor ID. The actual read process is then mapped onto the corresponding read function (2,3) named EXT_RD_B{Q} or INT_RD_B{Q}. This process does not need an extra clock cycle and should therefore always be favored over the seperated access functions (2,3).

2.4 Network

The bus network connects all processing units with a shared bus. As there are no tristate buffers in the FPGA the common bus is implemented using multiplexers. The challenging part is the implementation of an efficient arbiter. Two approaches have been implemented, a round robin arbiter and a dynamic prioritized arbiter.

Dynamic Prioritized Arbiter (BDPA). The dynamic prioritized arbiter checks the request signals by searching the processor with the highest ID that wants to read via the network. The IDs of the processors are compared with a comparator tree. A processor with an active request signal advances its ID otherwise zero through the comparator tree. The comparator tree finds the highest processor ID with an active request and forwards this request with its corresponding address signals onto the bus.

2.5 Generation Synchronization

All processors run independently from each other which leads to unsynchronized executions of the processors. To ensure data consistency a special synchronization point was introduced. The synchronization point, implemented with a custom instruction function as described in section 2.3, ensures that at a certain time all processors have executed the current generation and are ready to start execution of the next generation. A barrier-synchronization technique [34, p. 165] is implemented by *AND* gating.

2.6 FPGA Prototype Implementation

The prototyping platform is a Cyclone FPGA with the Quartus II 8.1 synthesis software from Altera. The Cyclone II FPGA contains 68,416 logic elements (LE) and 1,152,000 RAM bits [1]. The implementation language is Verilog HDL. The NIOS II processor is built with the SOPC-Builder (System-On-a-Programmable-Chip). We used the NIOS II/f processor. This core has the highest execution performance and uses a 6 stage pipeline. The DMIPS [35] rate is 218 at the maximum clock frequency of 185 MHz. The synthesis settings are optimized for speed. Table 1 and table 2 show the numbers of logic elements (LE), logic elements for network, the memory usage and the maximum clock frequency for a different amount of blocks. One implementation uses 2 blocks (CA model), the other implementation uses 4 blocks (GCA model). The two models will be described in section 3.

Table 1: p NIOS II softcores configured on a Cyclone II FPGA. Resources and clock frequency using **2 blocks** (CA model)

processing units (p)	total LEs	LEs for net	register for net	mem bits	register bits	max. clock (MHz)
1	3,390	-	-	195,296	1,932	140.00
2	6,135	50	2	226,720	3,357	127.78
4	11,712	116	3	289,568	6,211	107.15
8	22,898	157	4	415,264	11,921	92.86
16	45,184	354	5	666,656	23,351	75.00

Table 2: p NIOS II softcores configured on a Cyclone II FPGA. Resources and clock frequency using **4 blocks** (GCA model)

processing units (p)	total LEs	LEs for net	register for net	mem bits	register bits	max. clock (MHz)
1	3,421	-	-	326,368	1,932	131.25
2	6,251	83	2	357,792	3,357	120.85
4	12,005	198	3	420,640	6,211	103.34
8	23,473	415	4	546,336	11,921	88.89
16	46,605	892	5	797,728	23,353	73.08

3 CA and GCA Model for Traffic Simulation

In the Nagel-Schreckenberg algorithm each agent has a front view of v_{max} cells where v_{max} denotes the maximum speed of an agent. A straight forward approach is to check the type of all front cells up to the current speed (CA algorithm in section 3.1 to be used for comparison). Instead, our novel approach is to interconnect all cells with each other dependent on their current type (GCA algorithm in section 3.2).

In the following sections C_i denotes the cell C (or its type) at the current index i, Δ the gap size between two agents and d the distance from an agent cell to an empty cell. The abbreviations E = Empty, A = Agent will be used. In our simulations there is one agent type representing a car. Other vehicle types can be added by using multiple agents.

3.1 CA Model with Searching

The Nagel-Schreckenberg algorithm usually is modeled as Cellular Automata which is then sequentially or partially in parallel executed (simulated). In the simulation each cell computes at first its next state s' by using its own current state *s* and the states *sn* of the neighbor cells, *s' := f(sn, s)*. Thereby the sequence of computations is arbitrary. Secondly, after each cell has computed its next state, the next state is copied to the current state *(s := s')*, for all cells in any order. Thereby synchronous updating is simulated. If several processors are simulating subarrays of the whole cell field in parallel, then the processors have to wait for the update step until all processors have computed their next states. In an implementation a barrier synchronization can be used.

In this application the radius is v_{max} to the right (vehicles shall drive from left to right) for an Agent. The radius is v_{max} to the left and two to the right for an Empty cell. In a sequential implementation on a standard computer it is reasonable to check the neighbors sequentially for certain properties with the option to stop the checking when a certain property was found. Thereby the mean number of checks is lower than the maximal neighborhood distance, that is in our case v_{max}. The model with searching requires two blocks $C_i = (r, v)$ per cell, where v is the current speed and r the random number. An Empty cell is defined by $v > v_{max}$, otherwise if $0 \leq v \leq v_{max}$ then the cell type is Agent.

An Agent at first increments its speed ($v' := max(v+1, v_{max})$). Then all cells to the right (relative distance $= 1, 2, ..., v_{max}$) are accessed consecutively to determine the gap size Δ. The new speed may not be higher than the gap size ($v' := min(v', \Delta)$). Then the new speed is reduced by one with probability π: ($v' := v' - 1$), and the cell can be updated: if $v' = 0$ then the new type remains Agent (agent cannot move), otherwise the new type is Empty (agent is moving). In the hardware implementation the probability is computed by checking a 16 bit unsigned integer against a limit. Alternatively the probabilistic reduction of the speed can be written as ($v' := v' - R$), where R is a random binary variable (zero or one).

The cell rule for an Empty cell C_i is more complex. An empty cell has to check all cells to the left until the maximum speed v_{max} or until an Agent is found at position $i - d$. If an agent is found, the same calculation is performed as the one the agent is doing itself, in order to find out whether the agent has to be copied from cell C_{i-d} or not.

Searching from the agent back to the empty cell in order to find the gap size can be avoided, because the cells have already been checked to find the agent. Only if the gap is of size $d+1$ or $d+2$ the agent might move to the empty cell. Therefore it is necessary to check only the two cells to the right of the empty cell in order to react correctly. The agent will only be copied if the incremented and randomly reduced speed is equal to the distance d. The optimized algorithm for Empty is (with the default setting $C_i := E$):

```
1. // search to the left for an agent at rel. distance d
   $d := 0$; repeat $d := d + 1$ until ($C_{i-d} = A$) or ($d = v_{max}$);
   if ($C_{i-d} = E$) then return // empty cell remains empty

2. // execute 2., 3., 4. only if agent was found
   $v' := min(v_{max}, v + 1)$;
   if ($v' < d$) then return // agent cannot reach E

3. if ($v' > d$)&($C_{i+1} = A$) OR ($v' = d$) then $v' := d - R$
   else
       if ($v' = d + 1$)&($C_{i+1} = E$) OR ($v' > d + 1$)&($C_{i+1} = E$)&($C_{i+2} = A$) then
           $v' := d + 1 - R$

4. if ($v' = d$) then $C_i(t + 1) := A$ (with speed $v(t + 1) := v'$) //sync. update
```

The required neighborhood distance with static links is the union of v_{max} to the right, v_{max} to the left, and two to the right for uniform cells (Agent or Empty). When the algorithm is sequentially executed (simulated), then the worst case sequential complexity is $O(N \cdot v_{max})$, because

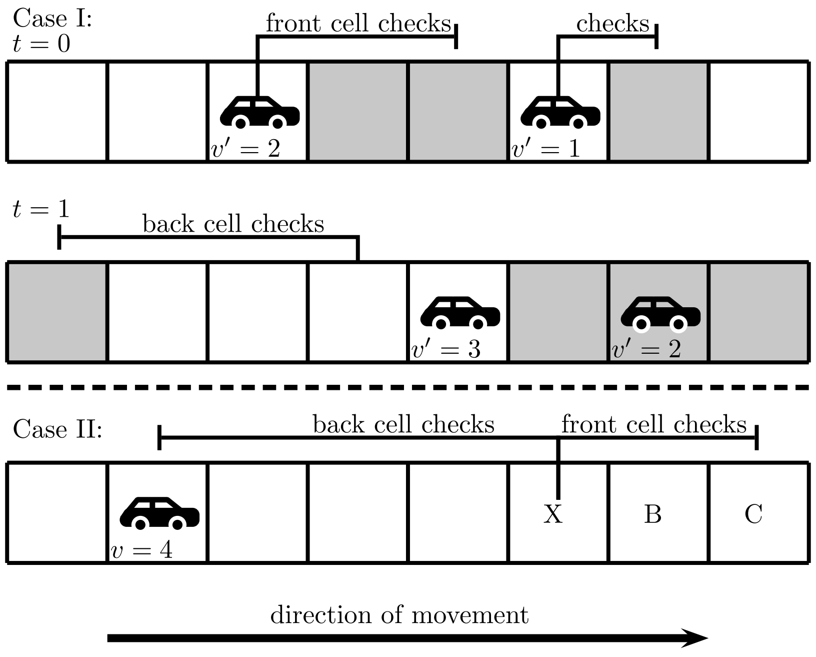

Figure 4: CA algorithm with searching

at most v_{max} (respectively $v_{max} + 2$) steps for searching are necessary. The execution time when simulating on a p processor system will be shown in section 4.

Example: Figure 4 shows the movement of two agents (cases I *(t=0)*, *(t=1)*). The hatched cells denote the cells that have to be checked by the agent. The left agent has a speed of two while the right agent has a speed of one (after acceleration) which leads to a different amount of front cell checks. In case I (t=1) the left agent had increased its speed to three but the amount of front cell checks is limited to two because an agent will be found there. The amount of front cell checks varies between zero and v_{max}.

Case II shows the operations of the empty cell X. The current speed of the agent is four and is increased to five. The empty cell X checks the cells to the left until v_{max} or an agent is reached. In this case an agent is found and X has to calculate the agents destination cell. If cell B is empty then the destination will be X or B (because of random decrease). Only if the final destination is X, the agent will be copied. Another case: if the agent wants to move to C and there is an agent, then the agent's speed is limited to the destination B which can further be reduced randomly to the final location X. So an empty cell has to check two cells to the right if the agents speed is equal or higher than the gap.

3.2 GCA Model with Linked Agents

A novel, more complex approach is a GCA model with interconnected cells. This approach requires four blocks per cell $C_i = (r, L, z, v)$. If $v \leq v_{max}$ then the type is A and v is the speed, otherwise the type is E. The second block holds the link z, representing a relative value (a distance/speed). z is called *precomputed speed*. The third block holds the link L, representing an absolute value (points to a position). And the fourth block holds the random number r. Comparing r against a limit set by using the probability π the binary variable R is determined (0 or 1).

The link L is used by an agent to point to the next agent in front. It is used by an empty cell to point to the next agent behind it that might move to it. Therefore, the link points forward for all agent cells and backward for all empty cells. All cells are interconnected in a circular way because we assume that there are no boundaries (wrap-around). The link L is an absolute address allowing direct access to the position of another cell without further address calculation.

$$L'(i) := \begin{cases} L(L(i)-1) + z(L(L(i)-1)) & (E \to E) \wedge \text{skipped } (C_i) \wedge C_{L(i)-1} = E \quad (1) \\ L(i) - 1, & (E \to E) \wedge \text{skipped } (C_i) \wedge C_{L(i)-1} = A \quad (2) \\ L(i) + z(L(i)), & (E \to E) \wedge \text{not skipped } (C_i) \quad (3) \\ L(L(i)) + z(L(L(i))), & (E \to A) \quad (4) \\ L(i-1) + z(L(i-1)), & (A \to E) \wedge C_{i-1} = E \quad (5) \\ i - 1, & (A \to E) \wedge C_{i-1} = A \quad (6) \\ L(i) + z(L(i)), & (A \to A) \quad (7) \end{cases}$$

$$v'(i) := \begin{cases} E, & (E \to E) \quad (8) \\ z(L(i)), & (E \to A) \quad (9) \\ E, & (A \to E) \quad (10) \\ z(i), & (A \to A) \quad (11) \end{cases}$$

$$z'(i) := \begin{cases} -, & (E \to E) \quad (12) \\ max[min[z(L(i)) + 1, L(L(i)) + z(L(L(i))) - i - 1] - R, 0], & (E \to A) \quad (13) \\ -, & (A \to E) \quad (14) \\ max[min[1, L(i) + z(L(i)) - i - 1] - R, 0], & (A \to A) \quad (15) \end{cases}$$

Figure 5: GCA algorithm for traffic simulation. Rules for new link L', new speed v', new precomputed speed z'.

$$cond(E \to E) : L(i) + z(L(i)) \neq i$$
$$cond(E \to A) : L(i) + z(L(i)) = i$$
$$cond(A \to E) : z(i) \neq 0$$
$$cond(A \to A) : z(i) = 0$$
$$skipped(C_i) : L(i) + z(L(i)) > i$$

Figure 6: Conditions for the GCA algorithm in figure 5

The block z is used by an agent to represent the precomputed speed in the next generation ($z(t) = v'(t) = v(t+1)$). It is not used by empty cells. So the new speed does not need to be computed in the current generation as it was already computed in the previous generation and is given by z. It can also be used as an offset to determine the next empty cell that will become an agent cell. Using the link L and the precomputed speed z it is possible to compute the next generation without searching. At start-up an additional generation is needed to interconnect the agents by the links L and to compute the precomputed speed z. The initial generation is regarded to be given.

The GCA algorithm is shown in figure 5. Figure 6 shows the conditions for the GCA algorithm. For each cell C_i the new link $L'(i)$ (rules 1-7), the new speed $v'(i)$ (rules 8-11) and the new precomputed speed $z'(i)$ of an agent (rule 12-15) is computed synchronously in parallel.

In case of the movement of an agent, the rules 5, 6, 10 apply for AGENT ($A \to E$) and the rules 4, 9, 13 for EMPTY ($E \to A$). If an agent moves from position i to k (from agent cell C_i to empty cell C_k) then $C_i := E$ and $C_k := A$. The condition ($E \to A$) for cell k is $L(k) + z(L(k)) = k$, meaning that the empty cell E points back with L to an agent A which in turn moves with offset z forward to E. The corresponding condition for the involved agent is $L(i + z(i)) = i$, meaning that the agent moves to an empty cell which in turn points back to the agent. As the specific destination cell is not important for the agent's cell, the condition can be substituted by $z(i) \neq 0$.

An agent does not move ($A \to A$) if its precomputed speed z is zero (rules 7, 11, 15). An empty cell remains empty if the agent behind it cannot reach it (rules 3, 8), or jumps over it (rules 1, 2, 8).

In the GCA model random access to the information of another cell is standard, e.g. $z(L(i))$. Not standard is the double indirect access, e.g. $z(L(L(i)))$. This extension can be allowed and can be supported by hardware using one more memory in a hardware implementation (pipeline with three cascaded memories: read cell contents including pointer $p1$, read global cell including pointer $p2$ at pointer $p1$, read next global cell including pointer $p2$; see hardware pipeline of a data parallel architecture [13, 15]. If the extended model is mainly sequentially simulated on a p processor system (see section 4), then the double indirection is no principal problem but may slow down the execution. If the algorithm is executed fully in parallel using $p = N$ processors (one for each of the N cells), then the parallel complexity is $O(1)$. When the algorithm is sequentially executed (simulated), then the sequential complexity is $O(N)$, as in the CA model with a fixed v_{max}. But the reason to prefer the GCA model is its faster execution on a p processor system, as will be shown in section 4.

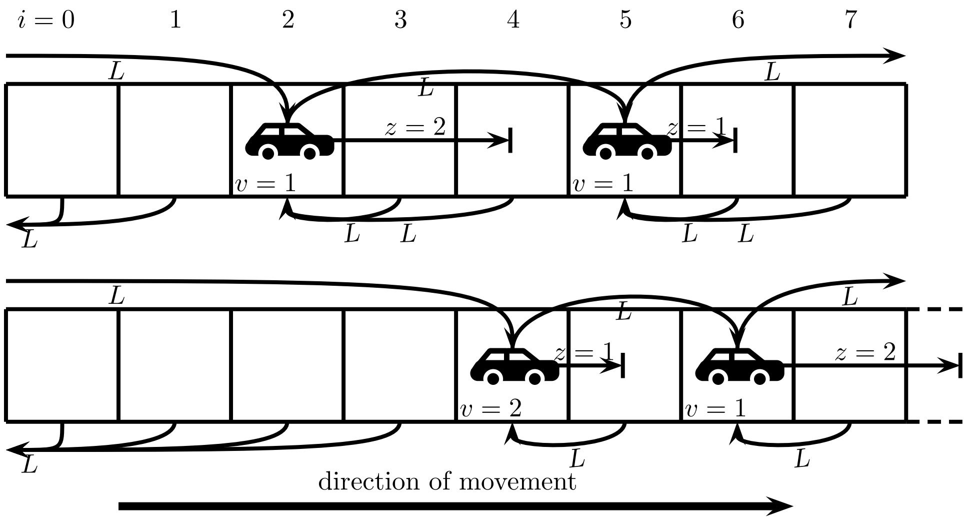

Figure 7: GCA algorithm with links

Example: Figure 7 shows the movement of the agents and the changes of the link L and the precomputed speed z. In generation $t = 0$ the left agent has a speed of one ($v = 1$) and the precomputed speed is two ($z = 2$). The offset z determines the destination cell for that agent and the agent is copied by the empty cell (generation $t = 1$). The precomputed speed z was reduced to the new gap size between the two agents in generation $t = 1$. An agent at position i computes its new precomputed speed z' by the data it can directly or indirectly access through the links (rule 13, 15).

An empty cell uses its link L to point backwards to the next agent A behind it. The empty cell at $i = 3$ is skipped therefore rule (1) is applied and for the empty cell $i = 1$ the rule (3) is applied. The empty cell at $i = 7$ is not skipped, therefore rule (3) of the algorithm applies, L' is incremented by z, $(L'(i) := L(i) + z(L(i)))$.

3.2.1 NIOS II Cell Rule / C-Code

The NIOS II cell rule implements the equations given in figure 5. Each cell needs four blocks to store the relevant informations $C_i = (r, L, z, v)$. The allocation is as follows: Block 0 holds the cell type. For agent cells the cell type is the current speed. Block 1 holds the link L (address). Block 2 holds the relative speed for the next generation (speed vector). Block 3 holds the random number, used for individual agent behavior. Listing 1 shows the NIOS II cell rule including the custom instructions defined in section 2.3.

Listing 1: Cell rule for linked agents

```
int recalcIndexInLine(int p, int pInLine)
    {return (p%MAXX+(pInLine/MAXX)*MAXX);}

inline int abs(int a) {return (a>=0) ? a : -a;}

inline int min(a,b) {return (a<b) ? a : b;}

int main(){                                        //beginning of the cell rule
  int g, i, speedtype, L, Z, tmp;

  CI(NEXTGEN,0,0);                                    //startup synchronization
  CI(NEXTGEN,0,0);

  for(g=0;g<GENS;g++){                               //GENS: Generations to run
    for(i=SC;i<LOCL_CELLS+SC;i++){                              //SC: Start Cell
      speedtype = CI(RD_B0,i,0);

      if(speedtype==CELL_FREE)                               //cell type is empty
      {
        L = CI(RD_B1,i,0);
        if(L!=i)                                       //at least 1 agent per row
        {
          Z = CI(RD_B2,L,0);
          if(recalcIndexInLine(L+Z,i)==i){ //copy agent, update L, Z & V
            CI(WR_B0,i,Z);
            tmp=recalcIndexInLine(
                  CI(RD_B1,L,0)+CI(RD_B2,CI(RD_B1,L,0),0),i);

            CI(WR_B1,i,tmp);
            if(i<tmp)
              tmp=min(min(Z+1, (tmp-i)-1),MAXSPEED);
            else
              tmp=min(min(Z+1,(MAXX-abs(tmp-i))-1),MAXSPEED);

            CI(WR_B2,i,(tmp>0 && CI(RD_B3,L,0)<P) ? tmp-1: tmp);
          }
          else{                                         //stay empty, update L
            CI(WR_B0,i,CELL_FREE);
            tmp=L+Z;
                                             //check if empty cells are left out
            if( (i>L && i<tmp) || (i+MAXX>L && i+MAXX<tmp) ){
              tmp=recalcIndexInLine(L-1+MAXX,i);
              if(CI(RD_B0,tmp,0)==CELL_FREE){        //predecessor empty
                L = CI(RD_B1,tmp,0);
                CI(WR_B1,i,recalcIndexInLine(L+CI(RD_B2,L,0),i));
              }
              else
                CI(WR_B1,i,tmp);
            }
          else
            CI(WR_B1,i,recalcIndexInLine(tmp,i));
        }
      }
      else{                                    //complete row is empty
        CI(WR_B0,i,CELL_FREE);
        CI(WR_B1,i,i);
      }
    }
    else{
      if(speedtype<=CELL_AGENT)                   //cell type is agent
      {
        Z = CI(RD_B2,i,0);
        if(Z>0){                                          //move agent
          CI(WR_B0,i,CELL_FREE);

          tmp=recalcIndexInLine(i-1+MAXX,i);

          if(CI(RD_B0,tmp,0)==CELL_FREE){          //predecessor empty
            L = CI(RD_B1,tmp,0);         //copy new L from predecessor
            CI(WR_B1,i,recalcIndexInLine(L+CI(RD_B2,L,0),i));
          }
          else              //predecessor is agent, set L to that cell
            CI(WR_B1,i,tmp);
        }
        else{                                      //do not move agent
          CI(WR_B0,i,Z);

          L=CI(RD_B1,i,0);
          tmp=recalcIndexInLine(L+CI(RD_B2,L,0),i);
          CI(WR_B1,i,tmp);

          if(i<tmp)
            tmp=min(1, (tmp-i)-1);
          else
            tmp=min(1, (MAXX-abs(tmp-i))-1);

          CI(WR_B2,i,(tmp>0 && CI(RD_B3,i,0)<P) ? tmp-1: tmp);
        }
      }
      else             //undefined cell types are treated as obstacles
        CI(WR_B0,i,CELL_OBSTACLE);
    }
  }
CI(NEXTGEN,0,0);                          //generation synchronization
}
return 0;}
```

4 Speed-up on an FPGA Multiprocessor System

In order to test the performance of the algorithms in a practical environment, an FPGA multiprocessor system was configured on an Altera Cyclone II FPGA. It consists of p NIOS II processors enhanced with special instructions supporting the GCA model and the access to all the memories. The processors are connected by a bus system with dynamic arbitration with each other. In a previous investigation [26] it turned out that the bus network performs better for agent simulations than other networks for this architecture, therefore this network is also used here. Here the previous multiprocessor system was modified in such a way that each cell memory was separated into memory

Table 3: Clock cycles, execution time and speed-up (one lane, *10% agents*)

	CA Algorithm			
processing units (p)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	400,731	-	2.862	1.00
2	201,399	1.99	1.576	1.82
4	102,250	3.92	0.954	2.99
8	51,450	7.79	0.554	5.17
16	26,179	15.31	0.349	8.20

Table 4: Clock cycles, execution time and speed-up (one lane, *10% agents*)

	GCA Algorithm			
processing units (p)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	187,791	-	1.431	1.00
2	95,203	1.97	0.788	1.82
4	49,077	3.83	0.475	3.01
8	24,980	7.52	0.281	5.09
16	12,729	14.75	0.174	8.21

Table 5: Clock cycles, execution time and speed-up (one lane, *50% agents*)

	CA Algorithm			
processing units (p)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	237,911	-	1.699	1.00
2	121,983	1.95	0.955	1.78
4	62,271	3.82	0.581	2.92
8	34,502	6.90	0.372	4.57
16	16,591	14.34	0.221	7.68

Table 6: Clock cycles, execution time and speed-up (one lane, *50% agents*)

	GCA Algorithm			
processing units (p)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	201,793	-	1.537	1.00
2	101,783	1.98	0.842	1.83
4	50,882	3.97	0.492	3.12
8	25,638	7.87	0.288	5.33
16	12,973	15.55	0.178	8.66

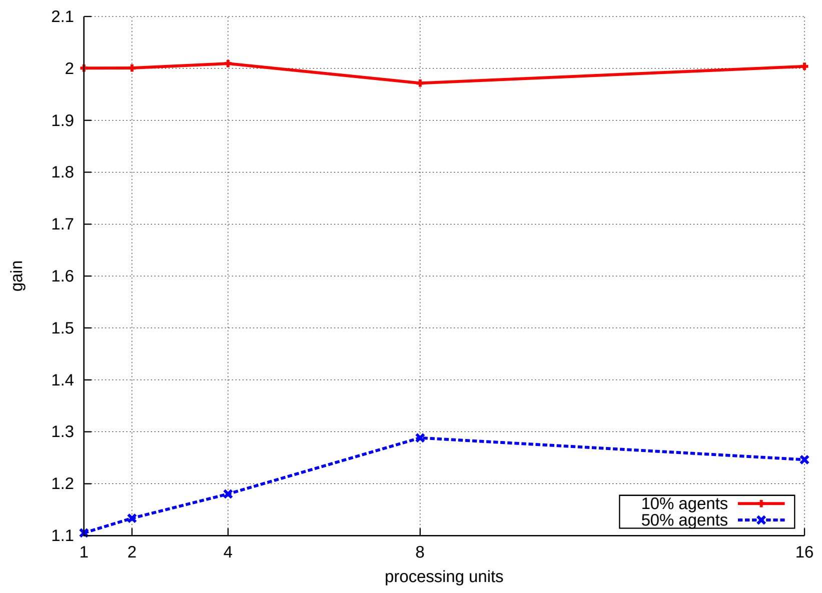

Figure 8: Gain of the GCA algorithm over the CA algorithm. GCA algorithm executes roughly two times faster for 10% agents

blocks (modules), one for each cell field (block). Thereby the cell blocks can be accessed in parallel and can separately be modified. A special instruction was defined for barrier synchronization.

The used size of the whole CA/GCA array was 1×2048 (one traffic lane), and a data segment of $\frac{2048}{p}$ was assigned to each processor. The number of agents was 204 and 1024, corresponding to a density of 10% and 50%. The maximum speed was set to $v_{max} = 5$. The probability to reduce the speed was $\pi = 0.5$. The algorithms were programmed in C, using a set of specific designed custom instructions (e.g., generation synchronization, internal and external memory accesses). A hardware counter was also included, counting the number of ticks (clock cycles) for the execution of an application. The reached clock frequency is 140 MHz ($p = 1$), 75 MHz ($p = 16$) for two blocks per cell (used in the CA model), and 131 MHz ($p = 1$), 73 MHz ($p = 16$) for four blocks per cell (used in the GCA model). The decrease of the clock frequency with a higher number of processors is a normal effect, because the complexity of the logic and the length of the data paths (especially the network) are increasing.

Tables 3, 4, 5 and 6 show the clock cycles, execution time, cycle speed-up and real speed-up for the CA and the GCA algorithm implemented on the multiprocessor system. The *cycle speed-up* for p is defined by the number of cycles for $p = 1$ divided by the number of cycles for p. So the decreasing clock frequency is not taken into account. The *real speed-up* is defined by the execution time for $p = 1$ divided by the execution time for p. Whereas the execution time of the GCA algorithm is relatively independent of the agent's density, the execution time of the CA algorithm is depending significantly on the density, because in the case of high density the searching of the gap is faster (the gap is smaller). The gain

$$gain = \frac{execution\ time\ for\ the\ CA\ algorithm}{execution\ time\ for\ the\ GCA\ algorithm}$$

is shown in figure 8. For a density of 10% the GCA algorithm is around two times faster. Further simulation with a density of 1% (20 agents) with different maximum speeds have been performed. For higher maximum speeds $v_{max} = \{5, 10, 20, 40, 80\}$ a gain of $\{2.3, 4.1, 7.9, 14.9, 29.3\}$ was reached using $p = 16$ processors and 20 agents (figure 9). So the GCA algorithm performs much better compared to the CA algorithm if the agent's density is low and the speed is high. In [21] the maximum speed was defined by the mean length of a vehicle. A higher maximum speed does not seem to be reasonable. However it can be used for a higher resolution of the grid or very fast agents.

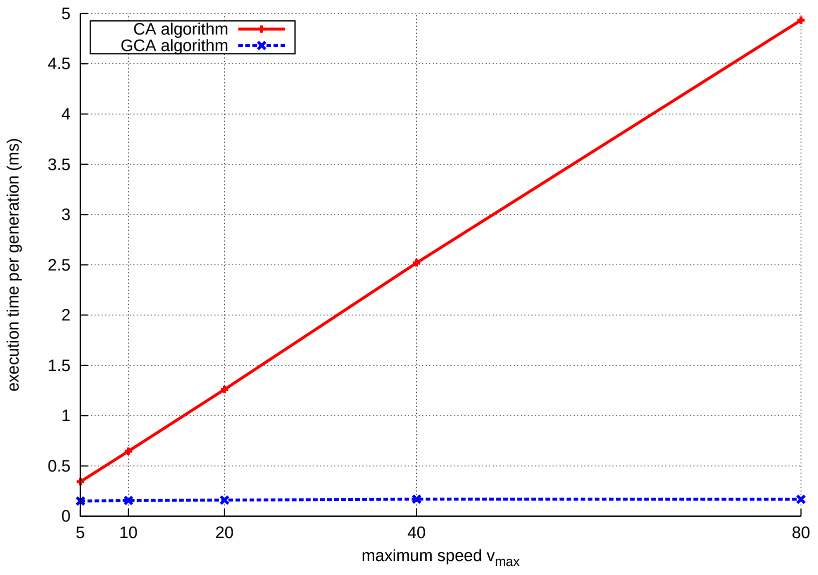

Figure 9: Time per generation for different maximum speeds. Agent density is 1%

5 Conclusion

The Nagel-Schreckenberg algorithm for traffic simulation was mapped onto the GCA computing model. In the GCA algorithm an agent (vehicle) is directly connected to its agent in front and an empty cell is connected to its following agent. An additional block in the cell is used holding the precomputed speed for the next generation. Thereby the new cells' state (movement, new type, new precomputed speed, new links) can directly be computed without any searching. For comparison also an optimized CA algorithm was developed that searches backwards for empty cells and forwards for agent cells. Both algorithms were implemented on an FPGA multiprocessor system. It turned out that the GCA algorithm executes significantly faster, especially for a low traffic density and a high vehicle speed. The cycle speed-up for a 16 processor system was 14.75 for 10% agents (out of 2048) and 15.55 for 50% agents, meaning that the GCA algorithm scales very well. Taking the decreasing clock frequency into account, a real speed-up of 8.21 for 10% agents and a real speed-up of 8.66 for 50% agents was reached on the 16 processor architecture.

References

[1] Altera, Datasheet Cyclone II.
`http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf`, 2006. Last visited: 2010-12-20.

[2] Altera, NIOS II Website.
`http://www.altera.com/products/ip/processors/nios2/ni2-index.html`, 2009. Last visited: 2010-12-20.

[3] Altera, NIOS II Website.
`http://www.altera.com/literature/lit-nio2.jsp`, 2009. Last visited: 2010-12-20.

[4] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3-4):507 – 525, 2001.

[5] A.K. Dewdney. Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American*, December 1984.

[6] R.M. D'Souza, M. Lysenko, and K. Rahmani. SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates. In *Proceedings of Agent 2007 Conference*, Chicago, IL, 2007.

[7] Ioakeim G. Georgoudas, P. Kyriakos, Georgios Ch. Sirakoulis, and Ioannis Th. Andreadis. An FPGA implemented cellular automaton crowd evacuation model inspired by the electrostatic-induced potential fields. *Microprocessors and Microsystems*, 34(7-8):285–300, June 2010.

[8] Wolfgang Heenes, Rolf Hoffmann, and Sebastian Kanthak. FPGA Implementations of the Massively Parallel GCA Model. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*, 2005.

[9] Wolfgang Heenes, Klaus-Peter Völkmann, and Rolf Hoffmann. Architekturen für den globalen Zellularautomat. In *19. PARS Workshop, Gesellschaft für Informatik (GI)*, 2003.

[10] Rolf Hoffmann, Klaus-Peter Völkmann, and Wolfgang Heenes. GCA: A massively parallel Model. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*, 2003.

[11] Rolf Hoffmann, Klaus-Peter Völkmann, and Stefan Waldschmidt. Global cellular automata GCA: an universal extension of the CA model. In *ACRI 2000 "work in progress" session, Karlsruhe, Germany*, 2000.

[12] Rolf Hoffmann, Klaus-Peter Völkmann, Stefan Waldschmidt, and Wolfgang Heenes. GCA: Global Cellular Automata, A Flexible Parallel Model. In *Proceedings of: 6th International Conference on Parallel Computing Technologies PaCT2001, Novosibirsk, Russia, 3. bis 7. Sept., 2001*, Lecture Notes in Computer Science (LNCS 2127), Springer Verlag, 2001.

[13] J. Jendrsczok, P. Ediger, and R. Hoffmann. A scalable configurable architecture for the massively parallel GCA model. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(4):275–291, 2009.

[14] Johannes Jendrsczok, Patrick Ediger, and Rolf Hoffmann. The Global Cellular Automata Experimental Language GCA-L. `http://www.ra.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_RA/papers/JEH08b.pdf`, 2007.

[15] Johannes Jendrsczok, Rolf Hoffmann, and Thomas Lenck. Generated horizontal and vertical data parallel gca machines for the n-body force calculation. In *ARCS '09: Proceedings of the 22nd International Conference on Architecture of Computing Systems*, pages 96–107, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Charles Kim. Cellular Automata Modeling of En Route and Arrival Self-Spacing for Autonomous Aircrafts. In *50th Annual Meeting of Air Traffic Controllers Association*, pages 127–134, August 2005.

[17] W. Knospe, L. Santen, A. Schadschneider, and M. Schreckenberg. A realistic two-lane traffic model for highway traffic. volume 35, pages 3369–3388, April 2002.

[18] Ari Kulmala, Erno Salminen, and Timo D. Hämäläinen. Evaluating Large System-on-Chip on Multi-FPGA Platform. In S. Vassiliadis et al., editor, *International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 179–189. Springer, 2007.

[19] Anna T. Lawniczak and Bruno N. Di Stefano. Development of road traffic ca model of 4-way intersection to study travel time. In Jie Zhou, editor, *Complex (2)*, volume 5 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 2040–2049. Springer, 2009.

[20] A.T. Lawniczak and B.N. Di Stefano. Digital laboratory of agent-based highway traffic model. In *Acta Physica Polonica B Proceedings Supplement*, volume 3, pages 479–453, 2010.

[21] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2(115):2221–2229, 1992.

[22] M. Rickert, K. Nagel, M. Schreckenberg, and A. Latour. Two lane traffic simulations using cellular automata. *Physica A: Statistical and Theoretical Physics*, 231(4):534–550, October 1996.

[23] Kyeong Keol Ryu, Eung Shin, and Vincent J. Mooney. A Comparison of Five Different Multiprocessor SoC Bus Architectures. In *DSD '01: Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 202–209, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. Evolving individual behavior in a multi-agent traffic simulator. In *Applications of Evolutionary Computation*, pages 11–20. Springer Berlin / Heidelberg, 2010.

[25] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 9th International Workshop, SAMOS 2009, Samos, Greece, July 20-23*, volume 5657 of *Lecture Notes in Computer Science*, pages 98–107. Springer Berlin / Heidelberg, 2009. ISSN 0302-9743 (Print) 1611-3349 (Online).

[26] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. GCA Multi-Softcore Architecture for Agent Systems Simulation. In Erik Maehle Stefan Fischer, editor, *Informatik 2009 Im Focus das Leben*, volume P-154 of *Lecture Notes in Informatics*, pages 278; 2268–82, 2009. ISSN 1617-5468.

[27] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA. In *22. PARS Workshop, Gesellschaft für Informatik (GI)*, volume 26, pages 48–57, 2009. ISSN 0177-0454.

[28] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. Multiprocessor architectures specialized for multi-agent simulation. November 2010. To be published in 2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA).

[29] Thomas Schmickl, Ronald Thenius, and Karl Crailsheim. Kollektive Sammel-Entscheidungen: Eine Multi-Agenten-Simulation einer Honigbienenkolonie. *Entomologica Austriaca*, 13:15–24, März 2006. ISSN 1681-0406.

[30] David Strippgen and Kai Nagel. Multi-Agent Traffic Simulation with CUDA. In *High Performance Computing & Simulation (HPCS)*, pages 106–114, Leipzig, Germany, 2009.

[31] David Strippgen and Kai Nagel. Using common graphics hardware for multi-agent traffic simulation with CUDA. In Olivier Dalle, Gabriel A. Wainer, L. Felipe Perrone, and Giovanni Stea, editors, *SimuTools*, page 62. ICST, 2009.

[32] Ronal Thenius, Thomas Schmickl, and Karl Crailsheim. Einfluss der Individualität bei Sammelbienen (Apis mellifera L) auf den Sammelerfolg. *Entomologica Austriaca*, 13:25–29, März 2006.

[33] Pablo Cristian Tissera, Marcela Printista, and Marcelo Luis Errecalde. Evacuation simulations using cellular automata. *Journal of Computer Science & Technology*, 7:14, April 2007.

[34] Theo Ungerer. *Parallelrechner und parallele Programmierung.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 1997.

[35] Alan R. Weiss. Dhrystone benchmark - history, analysis, "scores" and recommendations. `http://www.ebenchmarks.com/download/ECLDhrystoneWhitePaper.pdf`, 2002.

[36] Kazuhiro Yamamoto, Satoshi Kokubo, and Katsuhiro Nishinari. New Approach for Pedestrian Dynamics by Real-Coded Cellular Automata (RCA). In Samira El Yacoubi, Bastien Chopard, and Stefania Bandini, editors, *Cellular Automata*, volume 4173 of *Lecture Notes in Computer Science*, pages 728–731. Springer Berlin / Heidelberg, 2006.

[37] Shiwu Zhang and Jiming Liu. A Massively Multi-agent System for Discovering HIV-Immune Interaction Dynamics. In *Massively Multi-Agent Systems I*, volume 3446, pages 161–173. Springer Verlag, 2005. ISSN 0302-9743 (Print) 1611-3349 (Online).