Vol. 5 (2012) *Acta Physica Polonica B Proceedings Supplement* No 1

ROTOR-ROUTING ALGORITHMS DESCRIBED BY CA-w [*]

Rolf Hoffmann

Technische Universität Darmstadt, FG Rechnerarchitektur
Hochschulstr. 10, 64289 Darmstadt, Germany
hoffmann@ra.informatik.tu-darmstadt.de

(Received November 20, 2011)

The GCA-w model (Global Cellular Automata with write access) is an extension of the GCA (Global Cellular Automata) model, which is based on the cellular automata model (CA). Whereas the CA model uses static links to local neighbors, the GCA model uses dynamic links to potentially global neighbors. The GCA-w model is a further extension that allows modifying the neighbors' states. Thereby neighbors can dynamically be activated or deactivated. Algorithms can be described more concisely and may execute more efficiently because redundant computations can be avoided. If the neighborhood of the GCA-w model is locally restricted, we will call the model "CA-w" (Cellular Automata with Write-access). Rotor-routing algorithms are good examples showing the usefulness of the CA-w model. The Propp-machine and the Chip-firing problem are first described by CA for comparison, and then by CA-w. It is shown that the CA-w descriptions are more concise, more "natural" compared to the CA descriptions, and more power saving because only the active cells have to be computed.

DOI:10.5506/APhysPolBSupp.5.53
PACS numbers: 87.17.Aa, 92.60.hk, 87.18.Hf

1. Introduction

The GCA-w parallel computing model (GCA with write-access) introduced in [1, 2] is an extension of the GCA (Global Cellular Automata) model [3], which in turn is an extension of the CA model. A cell of a GCA can dynamically establish links to any of its global neighbors, whereas a cell of a CA uses only the fixed links to its local neighbors. The CA and the GCA model do not allow modifying the state of a neighbor. Therefore no write-conflict can occur, simplifying implementations in hardware or in software. However for applications, where the amount of active cells in the

[*] Presented at the Summer Solstice 2011 International Conference on Discrete Models of Complex Systems, Turku, Finland, June 6–10, 2011.

whole field is low or is varying over time, or the locations of the active cells are changing, the GCA-w model is a better choice. The GCA-w model allows writing information to its neighbors. This feature is very important because information can actively be transferred to a destination, and the activity of the destination can be switched on or off. Therefore, the GCA-w model is very useful for the description of problems with moving particles or moving agents, or problems with dynamic activities.

Several GCA-w applications were already described in [1,2,8] (one-to-all communication, synchronization, moving agents, different random walks of particles, pointer inversion, sorting with pointers, Pascal's Triangle, traffic simulation). The purpose of this contribution is to show two other practical applications that can easily be implemented. The two problems are related because they are using the rotor-routing principle, meaning that a rotor (pointer) as part of the cell defines the direction of the particle movement. Furthermore the two problems are using only a local neighborhood, so particles can only be pushed from a cell to its nearest neighbors. If the GCA-w model is restricted to local neighbors it will be called "CA-w" (Cellular Automata with write access).

The two problems are the Propp-machine with one agent [9, 10] and the chip-firing problem [11, 12]. It will be shown that these problems can easier be modeled and more efficiently be executed using the CA-w model compared to the CA model. These problems are perfectly suited to the CA-w model because *(i)* the particles decide on their own upon their next location, *(ii)* possible conflicts are easily resolved because the number of received particles is simply summed up, and *(iii)* cells which do not contain particles are excluded from the computation.

1.1. GCA-w model

Fig. 1 shows the general idea of the GCA-w model: Each cell C is dynamically connected via links (also called *hands*[1]) h_i to other cells, in the example to A, B, D. Cell C updates its own state and the states of its neighbors A, B, D. Thereby the links h_i are also updated. In the following, only one link per cell is assumed, which seems to be sufficient to model most applications. In addition, in the following, the cells will be arranged in a 1-D fashion, although n-D fields can easily be handled by using an n-D indexing scheme.

A potential difficulty is that write-conflicts may appear. The worst case scenario is that all cells want to write onto the same cell. In order to reduce the implementation effort to resolve the conflicts, the GCA-w rules should

[1] If k access pointers are used, then we call the GCA-w "*k-handed*"; this terminology was already used for the GCA model [3].

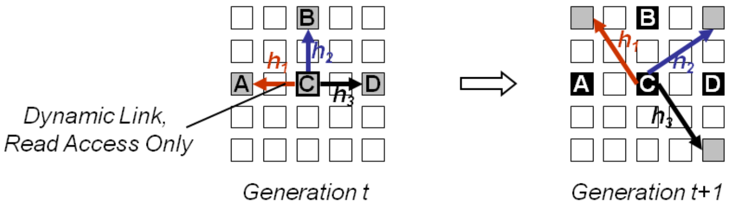

Fig. 1. A 2-D GCA-w example with three links per cell: Each cell is dynamically connected to a global (or locally restricted) set of neighbors (grey), only the activity of the centre cell C is shown. The state of C including the links h_i, and the states of its neighbors can be changed (grey to black) by a local rule. Thereby information can be transferred from C to the current neighbors A, B, D, and the activity of the neighbors may be changed. Write conflicts may occur, *e.g.* if C itself and other cells try to update C at the same time.

be designed in such a way that either no conflict will occur (which is the case for the first application, Sec. 2), or that the amount of conflicts is low or restricted, or that the conflicts can be resolved within a local neighborhood (which is the case for the second application, Sec. 3).

Global Cellular Automata with Write Access:
$$\text{GCA-w} = (I, Q, \delta, h, f, g, e) \tag{1}$$

Index Set, unique labels identifying the cells:
$$I = \{0, 1, \ldots, i, \ldots, N-1\} \tag{2}$$

States: $Q = A \cup P \cup D$ (3)

Active States: $A = \{a_1, a_2, \ldots, a_n\}$ (4)

Passive States: $P = \{p_1, p_2, \ldots, p_m\}$ (5)

Dead States: $D = \{d_1, d_2, \ldots, d_k\}$ (6)

Don't – Write – Symbol: δ

Write – Values: $Q_\delta = Q \cup \{\delta\}$ (7)

Configurations: Q^N (8)

A Configuration: $L = (q_0, q_1, \ldots, q_i, \ldots, q_{N-1}) \in Q^N, i \in I$ (9)

Neighbor's Address (in the case of absolute addressing) $h(i, q)$:
$$h : I \times Q \to I \tag{10}$$

Neighbor's Address (in the case of relative addressing) $h_{\text{rel}}(i,q)$:

$$h_{\text{rel}} : I \times Q \to I_{\text{rel}} = \{0, \pm 1, \pm 2, \ldots, \pm(N-1)\} \tag{11}$$

$$\text{such that } h(i,q) = i + h_{\text{rel}}\,[2] \tag{12}$$

Local – Rule $f(i,q,q^*)$, where q^* = *neighbor's state*:

$$f : I \times Q \times Q \to Q_\delta \tag{13}$$

Write – Rule $g(i,q,q^*)$:

$$g : I \times Q \times Q \to Q_\delta \tag{14}$$

Conflict – Rule $e\left(i, f, g^0, g^1, \ldots, g^j, \ldots, g^{N-1}\right)$:

$$e : I \times Q_\delta^{N+1} \to Q_\delta \tag{15}$$

Rule Application (Synchronous Updating) $\forall i \in I$:

$$q_i \leftarrow \begin{cases} e\Big(i, f\left(i, q_i, q_{h(i,q_i)}\right), g^0_{\to i}, .., g^j_{\to i}, .., g^{N-1}_{\to i}\Big) & \text{IF } q_i \in A \wedge e \neq \delta \quad (16) \\ q_i & \text{IF } q_i \in D \vee e = \delta \quad (17) \\ e\Big(i, \delta, g^0_{\to i}, \ldots, g^{j=i}_{\to i} = \delta, \ldots, g^{N-1}_{\to i}\Big) & \text{IF } q_i \in P \wedge e \neq \delta\,, \quad (18) \end{cases}$$

where $\forall (i,j) \in I \times I$:

$$g^j_{\to i} = \begin{cases} g\Big(j, q_j, q_{h(j,q_j)}\Big) & \text{IF } \; h(j,q_j) = i \wedge q_j \in A \quad (19) \\ \delta & \text{IF } \; h(j,q_j) \neq i \vee q_j \notin A\,. \quad (20) \end{cases}$$

The cells are arranged as a sequence $\langle a_i \rangle_{i \in I}$ of cells, each cell is labeled by its index i (1). The index can also be accessed by the the cell itself in order to implement non-uniform rules. A cell can be in an active state (4), or in a passive/inactive state (5), or in a dead-state (6). These three classes of states are also called *operational states.* Operational states are introduced in order to switch on or off the activity of the cells, and thereby allowing to reduce the computational effort (dead cells are totally excluded from the computation because they remain dead forever, passive cells do not compute themselves but can be activated by other cells). In addition, passive or dead cells can be used to define a termination condition, *e.g.* if all cells are dead, or if all cells are passive.

If a cell is active, it computes all the local functions h, f, g, e. If a cell is passive, it does not compute its local functions h, f, g, but it can be switched into another operational state from outside, *e.g.*, it can be changed into active. If a cell is dead, it will stay dead forever (it can be used as a constant).

[2] All addresses are mapped onto the interval $0..N-1$ by the modulo operation mod N.

The symbol delta ("Don't – Write") is a special output value of the functions f, g, and e. As a possible input of the conflict rule e it signals that no valid value is received from another cell for writing. In the case when the conflict rule does not receive any valid input ($\neq \delta$), it produces δ, too. Then the cell's state will remain unchanged.

The address function h defines the actual neighbor (absolute address, index) in access (read and write) (10). The neighbor's address can also be determined by the use of the relative addressing function h_{rel} (11). Relative addressing is often more adequate and more general to describe spatial relative situations. The local rule f computes the cell's new state if no neighbor is writing additionally. The *write-rule* computes a state value that can be written to the actual neighbor. The *conflict-rule* is dedicated to resolve the conflicts. It receives $N + 1$ messages (write-values), the own rule value f and messages $g^j_{\to i}$ from all cells j. Not all messages need to be activating: A non-activating δ-message is interpreted as a *Don't – Write*-command, meaning that a sender j does not want to write to a receiver i. A message is activating if the sender j is active and the receiver i is selected (18). (As a special case, g may produce δ; also f may produce δ.) If the sender is passive or the receiver is not selected, then δ is received.

Rule Application. All cells are updated synchronously in parallel. Only cells that are not dead need to be updated. If the cell is active (16) then the own rule value f and the messages $g^j_{\to i}$ from all cells j have to be taken into account. In the case that a δ-message (don't – write) is received, it is disregarded by the conflict rule e. If the cell is passive (18) then no computation of h, f, g takes place and the default values $f = \delta$ and $g = \delta$ are assumed, and no neighbor is selected. Although the cell is passive, the conflict-rule has to be awake because there might arrive activating messages.

At a first glance the GCA-w model seems to be too complex because of the conflicts and not very useful.

But in many applications the complexity of the conflict resolution can be reduced significantly, *e.g.* if the dynamic neighborhood access patterns are restricted or are known in advance, or if the rules are *Exclusive-Write* (as the algorithm in Sec. 2), or if the conflict resolution corresponds to a reduction operator (summing up the inputs as in the algorithm to be given in Sec. 3). The main advantage of the GCA-w model is that it allows to describe a certain class of algorithms more natural and concise, less redundant and energy saving (only the active cells are computing). The novel idea is to organize the computational task logically as an array of cells with different operational states (active, passive, dead) with write-access onto dynamically selected neighbors by the use of local rules only.

1.2. CA-w model

We will call the GCA-w model "*CA-w*" (Cellular Automata with write access) if the neighborhood is locally restricted. As in the GCA-w model the CA-w model may use k "hands" to access k neighbors in parallel, then we call the CA-w *k-handed*. Each hand can read from and/or write to a dynamically selected neighbor. Many applications can be described with one hand only. The formal description of an 1-D CA-w with one hand is the following (only the formulas differing from the ones in Sec. 1.1 are given):

Cellular Automata with Write Access:

$$\text{GCA-w} = (I, Q, \delta, h, f, g, e) \tag{1}$$

Neighbor's Address (in the case of absolute addressing) $h(i,q)$:

$$h : I \times Q \to I, \ \ (i-R) \le h' \le (i+R),\ h = h' \bmod N \tag{10}$$

Neighbor's Address (in the case of relative addressing) $h_{\text{rel}}(i,q)$:

$$h_{\text{rel}} : I \times Q \to I_{\text{rel}} = \{0, \pm 1, \pm 2, \ldots, \pm R\} \tag{11}$$

$$\text{such that } h(i,q) = (i + h_{\text{rel}}) \bmod N \tag{12}$$

Conflict – Rule $e\left(i, f, g^{i-R}, \ldots, g^{i-1}, g^{i}, g^{i+1}, \ldots, g^{i+R}\right)$:

$$e : I \times Q_\delta^{2R+1} \to Q_\delta \tag{15}$$

Rule Application (Synchronous Updating) $\forall i \in I$:

$$q_i := \begin{cases} e\Big(i, f\left(i, q_i, q_{h(i,q_i)}\right), g_{\to i}^{i-R}, \ldots, g_{\to i}^{i+R}\Big) & \text{IF } q_i \in A \wedge e \ne \delta \quad (16) \\ q_i & \text{IF } q_i \in D \vee e = \delta \quad (17) \\ e\Big(i, \delta, g_{\to i}^{i-R}, \ldots, g_{\to i}^{j=i} = \delta, \ldots, g_{\to i}^{i+R}\Big) & \text{IF } q_i \in P \wedge e \ne \delta\,. \quad (18) \end{cases}$$

The neighbors address lies within the radius R (10, 11). Inputs of the conflict rule (15) are the own function f and the possible messages $g^{i-R}, \ldots, g^{i+R}$ from the neighbors. The next state q_i at time $t+1$ is given by the output of e (16, 17, 18).

A more specific 1-D case (as an example, see Fig. 2): The radius is $R = 1$ and only the left neighbor $(i-1)$ or the right neighbor $(i+1)$ can be addressed. (The access to the own cell i is excluded here because the own state is available through the function f.) Then the definition simplifies to

Conflict – Rule $e\left(i, f, g^{i-1}, g^{i+1}\right)$:

$$e : I \times Q_\delta^{3} \to Q_\delta \tag{15}$$

Rule Application (Synchronous Updating) $\forall i \in I$:

$$q_i := \begin{cases} e\left(i, f\left(i, q_i, q_{h(i,q_i)}\right), g^{i-1}_{\to i}, g^{i+1}_{\to i}\right) & \text{IF } q_i \in A \wedge e \neq \delta \quad (16) \\ q_i & \text{IF } q_i \in D \vee e = \delta \quad (17) \\ e\left(i, \delta, g^{i-1}_{\to i}, g^{i+1}_{\to i}\right) & \text{IF } q_i \in P \wedge e \neq \delta\,. \quad (18) \end{cases}$$

Now there only three inputs of the conflict rule: f (own function), $g^{i-1}_{\to i}$ (message from left neighbor), and $g^{i+1}_{\to i}$ (message from right neighbor).

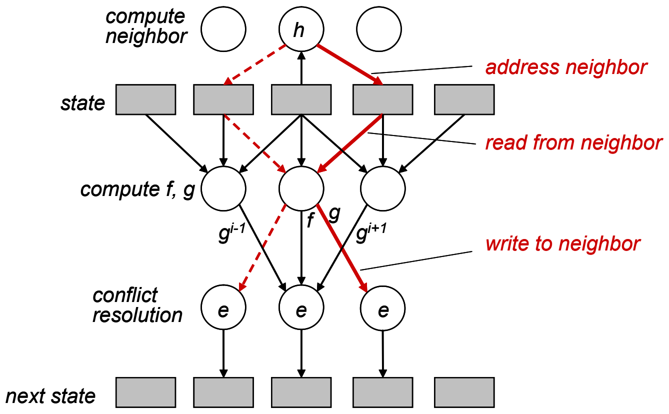

Fig. 2. 1-D CA-w model with one hand. The right neighbor is selected in this example. The functions f, g are computed, the value g is send to the right neighbor. The centre cell receives from the left g^{i-1}, and from the right g^{i+1}. These messages are taken into account by the conflict resolution function e.

1.3. Related work

The PSA (Parallel Substitution Algorithm) model [4] of computation is a very general and powerful model based on *substitution rules*. It allows also modifying the state of arbitrary target cells (*right side* of the substitution) using a "*base*" and a "*context*". In relation to the GCA-w the base corresponds to the cell under consideration, the context corresponds to the read neighbors and the right side corresponds to the cells which are modified. There is also a relation to the CRCW-PRAM [5,6] model. The PRAM model is based on a physical view with p processors that have global memory access to physical data words whereas the GCA-w is based on logical computing cells tailored to the application. Another difference of the GCA-w model compared to PRAM is the direct support of dynamic links and the rule based approach similar to the CA model.

Jim Propp's machine, also known as the "Propp-machine" or "rotor-router model", is a deterministic process that simulates a random walk on a graph [9, 10, 11, 12]. Instead of distributing a chip to a randomly chosen neighbor, the neighbor is determined by a rotor that is rotated after the movement in a fixed cyclic order. Several interesting theoretical results can be obtained by analyzing this deterministic process, and by comparing it to a true random walk. The purpose of this paper is not to analyze rotor-routing algorithms but rather to show that it is another application showing the usefulness of the CA-w model.

2. Rotor-routing with 1 agent

As a first example for a simple rotor-routing algorithm we will model the following problem by CA (for comparison) and by CA-w.

Given is a two-dimensional field of cells with cyclic boundaries. Exactly one agent is situated on one of the cells. Each cells contains a rotor (pointer) pointing to one of the nearest neighbors N, E, S, W. Then the agent reads the rotor's direction and moves to the neighboring cell defined by this direction. When the agents moves it rotates the rotor according to a given cyclic sequence, *e.g.* (toN, toW, toE, toS) $= (\uparrow, \rightarrow, \leftarrow, \downarrow)$.

2.1. CA model

The following algorithm describes the rotor-routing with one agent by CA. Each cell contains a *rotor* and an *agent.* If an agent is situated on a cell then $agent = ACTIVE$ else $agent = PASSIVE$. Note that in CA all cells are computing their new state, even if $agent = PASSIVE$. The nearest neighbors are defined in lines (04, 05). (The equivalences (04–07) can be considered to be substituted in the code thereafter.) Initially one agent is placed in the middle, all rotors point to $\uparrow$(08–12). As the field is a vertex-transitive Caylay graph, the view from each cell is the same, any position can be interpreted as the "middle". If a neighboring cell contains an agent and its rotor point to my cell C$[x, y]$, then my agent becomes active (14–19). If my cell contains an agent ($myagent = ACTIVE$) then it is passivated and the rotor is rotated. If the rotor sequence is chosen as $Rotate(i) = i + 1 \bmod m$ (clockwise rotation) then the agent's movements are given by the sequence

$$\uparrow^m,\ [(\rightarrow, \uparrow^m)^{m-1}, \rightarrow, (\downarrow, \rightarrow^m)^{m-1}, \downarrow, (\leftarrow, \downarrow^m)^{m-1}, \leftarrow, (\uparrow, \leftarrow^m)^{m-1}, \uparrow]^*.$$

In the start-up phase the agent moves m steps to $\uparrow$, Fig. 3, generation 0 to 10. Then a cyclic period of $4 \times m^2$ generations begins, Fig. 3, generation 10 to 410. Between generation 110 and 210 the agent moves (one step to $\rightarrow$, then m steps to $\uparrow$) repeatedly for $m - 1$ times until it reaches the start position again. Then the same behavior can be observed, but 90° clockwise rotated, and so on until the whole period ends. During this period all cells are visited 4 times, and each link (4 for each node) is traversed exactly once. Thus the agent performs a walk on a Hamiltonian cycle during each period.

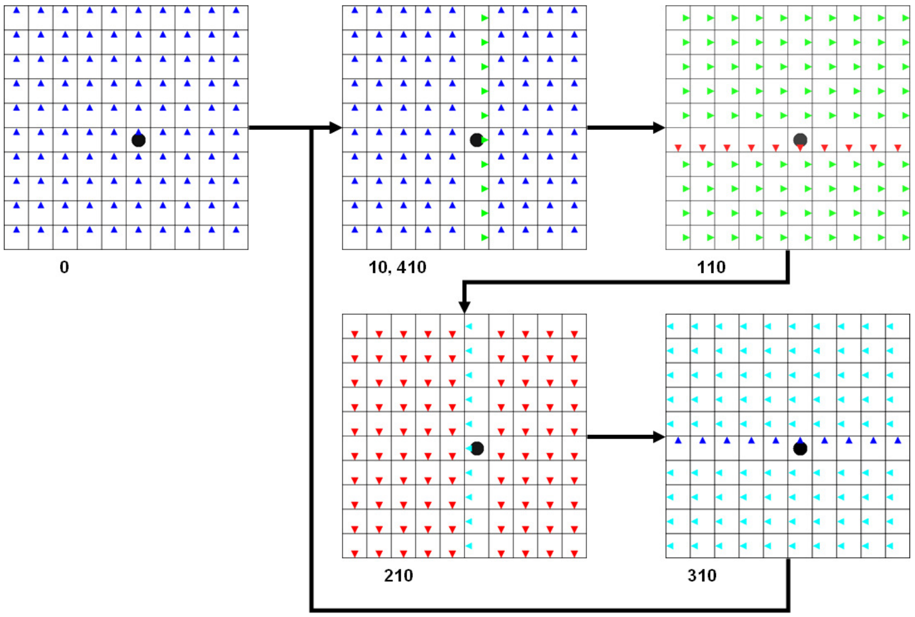

Fig. 3. Rotor-routing with one agent for a 10×10 field. Initially all rotors point toN. After 10 movements toN the agent is again in the middle, the rotors of the middle line are rotated toE. Then the agent moves one step toE. Between generation 10 and generation 410 each of the $4m$ links between the nodes is traversed once (Hamiltonian path), and each node is visited 4 times.

The process can also be started with a random rotor direction for each cell. Then, after a start-up phase of $k < m^2$ generations (k depending on the actual initial configuration), each cell will be visited 4 times during a period of $4 \times m^2$ generations, and each of the $4 \times m^2$ links will be visited once. For an example simulation the start-up phase took $k = 239$ generations for a 10×10 field. Such a process can be used to simulate a random walk in a deterministic way.

```
00  agent : (PASSIVE, ACTIVE)
01  rotor : (toN, toE, toS, toW) = (0, 1, 2, 3)
02  cell  = (rotor, agent)
03  C = ARRAY [0 .. m-1, 0 .. m-1] OF cell

    // equivalences, neighbors N E S W; addition/subtraction modulo m
04  N <=> C[x,    y-1];  E <=> C[x+1,  y  ];
```

```
05  S <=> C[x,    y+1];  W <=> C[x-1,  y  ];

06  myagent <=> C[x,y].agent   // own agent of centre cell
07  myrotor <=> C[x,y].rotor   // own rotor

    // initially one agent placed in the middle, all rotors point toN
08  PARALLEL C[x, y]
09     myrotor := toN
10     IF (x = n div 2) AND (y = n div 2)
11     THEN myagent := ACTIVE ELSE myagent := PASSIVE ENDIF
12  ENDPARALLEL

    // moving of the agent to the neighbor the rotor is pointing to
13  PARALLEL C[x, y]
       // case empty: if a neighbor is an active agent and points to me then copy
14     IF myagent = PASSIVE
15     THEN    IF ( N.agent = ACTIVE) AND N.rotor = toS ) OR
16                ( E.agent = ACTIVE) AND E.rotor = toW ) OR
17                ( S.agent = ACTIVE) AND S.rotor = toN ) OR
18                ( W.agent = ACTIVE) AND W.rotor = toE )
19             THEN myagent <- ACTIVE ELSE myagent <- PASSIVE    ENDIF    ENDIF

      // case agent: agent deletes itself when it is moving
20    IF myagent = ACTIVE
21    THEN myagent <- PASSIVE
22         myrotor <- Rotate(myrotor) ENDIF // rotate own pointer
23  ENDPARALLEL
```

2.2. CA-w model

The following pseudo-code for the same problem modelled by CA-w is several lines shorter. Whereas the CA description (main part) needs 11 lines (13–23) the CA-w description needs only 5 lines (13–17). The CA-w algorithm is computed only once for the active agent, whereas the CA algorithm is computed for all cells (for this application, the CA computation can also be optimized by computing only the agent and its 4 neighbors). The CA-w algorithm is more appropriate if you think that the agent is the active part of the system only. In contrast, the CA algorithm is more artificial because the movement is described by two rules: the sending cell deletes the agent and the receiving cell copies the agent. So in the CA model the activity is not concentrated in the agent but distributed over the involved cells around the agent.

```
00  agent : (PASSIVE, ACTIVE)
01  rotor : (toN, toE, toS, toW) = (0, 1, 2, 3)
02  rotx  : array [0 .. 3] =  0, 1, 0,-1
03  roty  : array [0 .. 3] = -1, 0, 1, 0
04  cell  = (rotor, agent)
05  C = ARRAY [0..m-1, 0..m-1] OF cell

06  myagent <=> C[x,y].agent   // own agent of centre cell
07  myrotor <=> C[x,y].rotor   // own rotor
```

```
08-12 // initial configuration is the same as in the above CA algorithm

    // moving of the agent to the neighbor the rotor is pointing to
13  PARALLEL C[x, y] WHERE agent = ACTIVE     // only compute where active
14     C[x+rotx[myrotor], y+roty[myrotor]].agent <- ACTIVE // activate new position)
15     myagent <- PASSIVE                     // passivate own cell (old position)
16     myrotor <- Rotate(myrotor)      // rotate own pointer
17  ENDPARALLEL
```

3. Chip-firing

At the beginning k chips are placed in the middle of a $m \times m$ field, here with cyclic boundaries.

The chips shall be distributed by a diffusion process. The diffusion works as follows: If the number of chips is greater than a threshold T then the cell fires and the chips are distributed to the neighbors. A quarter of the chips is moved to each of the four nearest neighbors. The remaining chips are distributed according to the rotor's direction which is rotated after each chip that has moved.

First a pseudo-code description is given for the CA model, and afterwards for the CA-w model.

3.1. CA model

Initially k chips are placed in the middle of the field (10–14), all rotors point to $\uparrow$. Then the chips are distributed (15–38). First the chips that are received from the four neighbors are computed (32). If my cell fires ($mychips > T$) then all my chips are distributed and my new chips are equal to the received chips (34). My own rotor is rotated according to the number of chips that were distributed (35). If my cell did not fire, the received chips are added to my chips (36). The number of received chips is computed in the lines (16–31). Each of the 4 neighbors R[i] are considered. If a neighbor fires, a quarter of the chips are stored in the temporary variable `from[i]`. In addition the remaining chips are added to `from[i]` if the rotated rotor of the neighbor points to my cell ($inverse(i) = i + 2 \bmod 4$).

The idea of the CA model is to look to the neighboring cells, and then compute the number of chips to be received from them, and then sum them up in the own cell.

```
00  chips : (0 .. k)
01  rotor : (toN, toE, toS, toW) = (0, 1, 2, 3)
02  cell  = (rotor, chips)
03  C = ARRAY [0 .. m-1, 0 .. m-1] OF cell

    // equivalences, neighbors N E S W; addition/subtraction modulo m
04  N <=> C[x,    y-1];  E <=> C[x+1,  y  ];
05  S <=> C[x,    y+1];  W <=> C[x-1,  y  ];
```

```
R[0] <=> N, R[1] <=> E, R[2] <=> S, R[3] <=> W
mychips <=> C[x,y].chips   // chips of centre cell
myrotor <=> C[x,y].rotor   // rotor of centre cell

received, from[0,1,2,3]: temporary variables

// initially k chips placed in the middle, all rotors point toN
PARALLEL C[x, y]
   rotor := toN
   IF (x = m div 2) AND (y = m div 2)
      THEN mychips := k ELSE  mychips := 0  ENDIF
ENDPARALLEL

PARALLEL C[x, y]
  // compute the chips received from neighbors
  FOR i = 0 .. 3     // do for all neighbors, if neighbor fired
    IF R[i].chips > T THEN
            from[i] := R[i].chips div 4  // first distribute equally

          // then distribute the remaining chips in addition
            CASE  R[i].chips mod 4
            =1: IF    R[i].rotor = inverse(i)
                THEN  from[i] := from[i] + 1  ENDIF
            =2: IF    R[i].rotor = inverse(i) OR
                      Rotate(1, R[i].rotor) = inverse(i)
                THEN  from[i] := from[i] + 1  ENDIF
            =3: IF    R[i].rotor = inverse(i) OR
                      Rotate(1, R[i].rotor) = inverse(i) OR
                      Rotate(2, R[i].rotor) = inverse(i)
                THEN  from[i] := from[i] + 1  ENDIF
            ENDCASE
    ENDIF
  ENDFOR
// received chips are now computed
received = from[0]+from[1]+from[2]+from[3]
// if own cell has fired then only receive
IF   mychips > T
THEN mychips <- received
     myrotor <- Rotate(mychips mod 4, myrotor)  ENDIF
// own cell not fired
ELSE mychips <- mychips + received
     myrotor <- myrotor  ENDIF
ENDPARALLEL
```

3.2. CA-w model

The basic idea of the CA-w model is simple: If the own cell fires then the chips are send to the neighbors which have the task to sum them up. In the following pseudo-code the operation $y(+) \leftarrow x$ is used. The meaning is that x is added to an implicit available temporary variable y' (initially set to zero), and later, when the clock event for the synchronous updating appears, y' is copied into y. In terms of hardware it means that the received chips are added up by a parallel adder (or by a sequential adder with y' used as temporary variable). In this example the multiple write-conflict to a neighbor is solved by summing up the received data. Therefore, the conflict resolution can be seen simply as a data reduction operator.

Compared to the CA algorithm, the CA-w algorithm is more concise again. In addition, computing power is only necessary where there are any chips. In contrast, in the CA model the whole field of cells has to be computed, or at least the cells containing chips and its neighbors. The CA computation is also more redundant, *e.g.* all 4 neighbors have to compute the quarter of my chips `R[i].chips div 4` (line 19) whereas this value (`mychips div 4`, line 13) has to be computed only once by my cell in the GCA-w model.

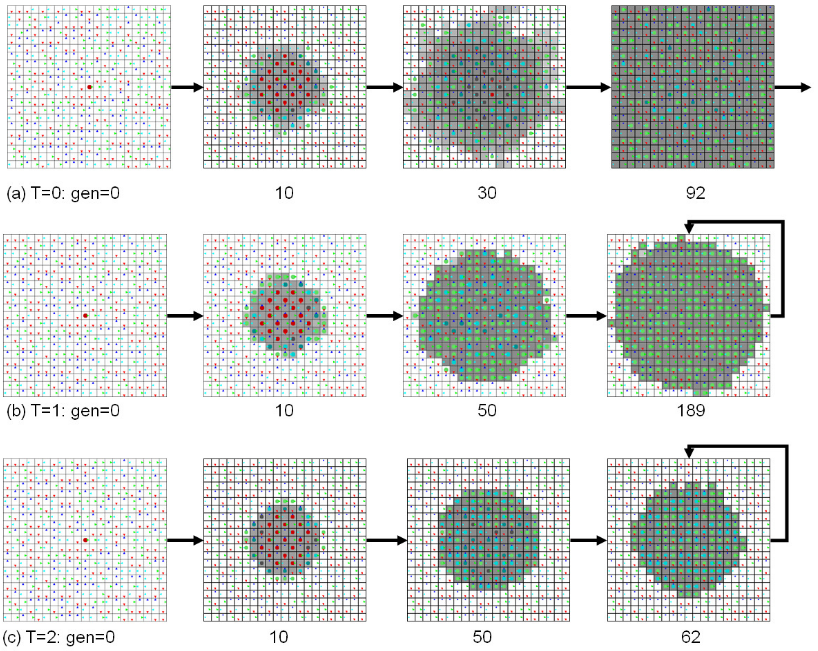

Fig. 4. Chip-firing simulation for a 21×21 field with cyclic boundaries. Visited cells are shown in grey. The number of chips on a cell are shown in black/red (> 4), dark grey ($= 4$), light grey/green or turquoise (1, 2, 3). At the beginning 210 chips are placed in the middle, and all rotors have a random direction. (a) Threshold $= 0$: After 92 generations all cells have been visited and the chips continue to move around (0–4 chips were observed per cell). (b) Threshold $= 1$: After 189 generations a fixed point has been reached. (c) Threshold $= 2$: After 62 generations a fixed point has been reached.

Simulations with different thresholds are shown in Fig. 4. Threshold = 0: After 92 generations all cells have been visited and the chips proceed to move around in a pseudo-random fashion. Threshold = 1: A cell fires if its number of chips is at least two. After 189 generations a stable configuration is reached, the visited area forms almost a circle. Threshold = 2: A cell fires if their number of chips is at least three. A fixed point is reached after 62 generations.

```
00 - 08  // the same as in the CA program

09  PARALLEL C[x, y] WHERE mychips > 0
10    IF   mychips > T
11    THEN                   // fire
        // first distribute equally to N,E,S,W
12      FOR i = 0 .. 3
13        R[i].chips (+)<- mychips div 4 // summed up by R[i]
14      ENDFOR

        // then distribute the remaining chips
15      CASE mychips mod 4
16      =1,2,3:  R[myrotor].chips (+)<- 1
17      =2,3:    R[Rotate(1, myrotor)].chips  (+)<- 1
18      =3:      R[Rotate(2, myrotor)].chips  (+)<- 1
19      ENDCASE

20      myrotor <- Rotate(mychips mod 4, myrotor)
21    ELSE
22      myrotor <- myrotor
23    ENDIF
24  ENDPARALLEL
```

4. Conclusion

The CA-w (Cellular Automata with write access) model was introduced, derived from the GCA-w model by a locally restricted neighborhood. This model allows to modify not only the cell's own state but also the neighbor's state. The neighbors are accessed via dynamic links that can be modified by the cell's rule. It was shown that the CA-w model is useful for describing rotor-routing algorithms. Such algorithms contain a rotor (a pointer) in each cell that defines the moving direction of particles (agents, chips). The Propp-machine with one agent and the chip-firing problem were described by CA-w, and for comparison by CA. The CA-w descriptions are more concise and "natural" because the change of the system state is only controlled by the agents. Furthermore, the computational effort to simulate CA-w in hardware or in software is minimized because only active cells have to be computed.

REFERENCES

[1] R. Hoffmann, *Acta Phys. Pol. B Proc. Suppl.* **3**, 347 (2009).

[2] R. Hoffmann, *Lect. Notes Comput. Sci.* **5698**, 194 (2009).

[3] R. Hoffmann, K.-P. Völkmann, S. Waldschmidt, Global Cellular Automata GCA: An Universal Extension of the CA Model, in: T. Worsch, (ed.), ACRI Conference, 2000.

[4] S. Achasova, O. Bandman, V. Markova, S. Piskunov, *Parallel Substitution Algorithms, Theory and Applications*, World Scientific, 1994.

[5] J. Keller, Chr. Kessler, J. Träff, *Practical PRAM Programming*, Wiley, 2001.

[6] J. JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[7] D. Keil, D. Goldin, Modeling Indirect Interaction in Open Computational Systems, Workshop TAPOCS, 12th IEEE Int. Workshops on Enabling Techn. (WETICE 2003), IEEE Computer Society 2003, ISBN 0-7695-1963-6.

[8] R. Hoffmann, *Acta Phys. Pol. B Proc. Suppl.* **4**, 183 (2011).

[9] J. Cooper, B. Doerr, J. Spencer, G. Tardos, *Eur. J. Combin.* **28**, 2072 (2007).

[10] B. Doerr, T. Friedrich, *Combin. Probab. Comput.* **18**, 123 (2009).

[11] J.N. Cooper, J. Spencer, *Combin. Probab. Comput.* **15**, 815 (2006).

[12] A.E. Holroyd *et al.*, *Prog. Probab.* **60**, 331 (2008).