# Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton

J. Jendrsczok[1], R. Hoffmann[1], J. Keller[2]

[1]TU Darmstadt
FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, D-64289 Darmstadt
{jendrsczok, hoffmann}@ra.informatik.tu-darmstadt.de

[2] FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Universitätsstr. 1, D-58084 Hagen
Joerg.Keller@FernUni-Hagen.de

## Abstract

*The GCA (Global Cellular Automata) model consists of a collection of cells which change their states synchronously depending on the states of their neighbors like in the classical CA model. In differentiation to the CA model the neighbors are not fixed and local, they are variable and global. The GCA model is applicable to a wide range of parallel algorithms, and it can be implemented on reconfigurable hardware. We discuss the GCA implementation of PRAM algorithms, exemplified by the algorithm of Hirschberg et al., which determines the connected components of a given undirected graph. Insights are that efficient mappings of PRAM algorithms onto GCA exist, and that PRAM and GCA optimality criteria differ because the latter takes memory consumption into account. This makes the GCA a parallel computational model and an implementation platform, thus narrowing the gap between theory and practice.*

## 1. Introduction

The GCA (Global Cellular Automata) model [7, 8] is an extension of the classical CA (Cellular Automata) model [10]. In the CA model the cells are arranged in a fixed grid with fixed connections to their local neighbors. Each cell computes its next state by the application of a local rule depending on its own state and the states of its neighbors. The data accesses to the neighbor's states are read-only and therefore no write conflicts can occur. The rule can be applied to all cells in parallel and therefore the model is inherently massively parallel. The CA model is suited to all kinds of applications with local communication, like physical fields, lattice-gas models, models of growth, moving particles, fluid flow, routing problems, picture processing, genetic algorithms, and cellular neural networks.

The GCA model is a generalisation of the CA model which is also massively parallel. It is not restricted to the local communication because any cell can be a neighbor. Furthermore the links to the neighbors are not fixed; they can be changed by the local rule from generation to generation. Thereby the range of parallel applications is much wider for the GCA model. Typical applications besides the CA applications are graph algorithms, hypercube algorithms, logic simulation [11], numerical algorithms, communication networks, neuronal networks, games, and graphics.

The state of a GCA cell consists of a data part and an access information part. In most implementations the access information part contains one or more pointers (Figure 1). The pointers are used to dynamically establish links to global neighbors. We call the GCA model *one handed* if only one neighbor can be addressed, two handed if two neighbors can be addressed and so on. In our investigations about GCA algorithms we found out that most of them can be described with only one pointer. Also the presented algorithm will use only one pointer. Additionally we call the GCA cells *uniform* if all cells have the same transition rule and otherwise *non-uniform*. The general aim of our research (supported by Deutsche Forschungsgemeinschaft, project *Massively Parallel Systems for GCA*) is the hardware and software support for this model [4]. Recently we have investigated how graph algorithms can be implemented on the GCA [2]. In this paper we describe the implementation of the graph-algorithm of Hirschberg et al. on the GCA as an example of the class of PRAM algorithms.

As the GCA cells work synchronously and can only read from other cells, the GCA resembles the concurrent read owner write (CROW) PRAM model, where each processor may read any cell, whereas each cell may only be written by a dedicated processor, the owner. In principle, the GCA

1-4244-0910-1/07/$20.00 

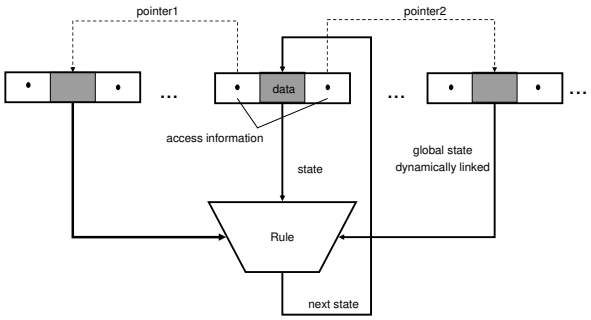


**Figure 1. The operation principle of the GCA.**

is able to implement any PRAM algorithm, as any algorithm consists of a finite number of instruction from a finite instruction set. However, an automaton implementation is particularly advantageous for simple algorithms, which are however available in abundance in the PRAM community.

In general, mapping a CROW PRAM algorithm onto a GCA requires a number of considerations. First, in many PRAM algorithms, the number $P$ of processing elements is expressed in terms of the problem size $n$, i.e. $P = P(n)$ while a particular GCA architecture has a fixed number $p$ of cells. Here, Brent's theorem can be applied, stating that each cell shall sequentially simulate $P(n)/p$ processing elements round robin. Second, PRAM shared memory has to be mapped onto the GCA cells, in a manner similar to PRAM simulations. If a cell hosts more than $O(1)$ shared memory elements, then the pointer mechanism of the GCA may have to be revised. Currently, there is only a limited number of registers that can be read in a neighbouring cell. The mapping should be deterministic, in order to guarantee that it is clear in the algorithm design stage. Then, all CROW PRAM algorithms can be implemented directly on a GCA. Third, if PRAM shared memory is distributed among the GCA cells, then for a cell $c$, the number of cells to which $c$ is the neighbour in a current step, i.e. that will read c's registers in this step, can be up to $p - 1$. In the theory of PRAM implementation on distributed memory machines, this number is called *congestion*. The duration of one step is bound from below by the maximum congestion of any cell in this step. As the GCA implements a particular algorithm, steps with known low congestion can be executed faster than those with high congestion.

Congestion can either be large because of concurrent reading (e.g. several cells trying to read the same array element), or because of an unfortunate mapping of memory elements onto cells. Yet, the latter cannot occur for a single-handed GCA, which considerably simplifies the mapping in many cases, such as our example. Concurrent reading can be handled in certain networks, in particular butterfly networks, by special routing algorithms, e.g. Ranade's algorithm. Unfortunate mappings can be prevented either by choosing an appropriate mapping in case where the neighbour relations are known beforehand, or by applying universal hashing. Universal hashing presents two difficulties. First, the owner relationship may get lost, second the congestion can only get down to a value of $O(\log p)$ for hash function classes that can be easily implemented. This can be tolerated if the algorithm works in rounds or generations, the communication results are needed only in the next round. The rounds then resemble Valiant's super steps [9]. If the duration of the communication is not longer than the length of the computation in a super step, then the communication will not lead to any slowdown at all. The duration of the communication is not only determined by the congestion, but also by the communication network. A fully connected network may not be realizable. As a GCA can be implemented in an FPGA architecture, the algorithm can be compiled in the cells' rule set, i.e. in hardware, and the communication structure can be adapted to the needs of the application. Thus, for many problems, the configurability of a GCA can provide better performance than a universal PRAM emulation.

## 2. Hirschberg's Algorithm

Our example application is Hirschberg's well known algorithm [5] to compute the connected components of an undirected graph on a CREW PRAM. Yet, only a CROW PRAM is really needed. The example application serves to explore the considerations sketched above in more detail, and gain more insights into the issues to be considered while implementing a PRAM algorithm onto a GCA. Hirschberg's algorithm was seminal and is work-optimal for dense graphs, i.e. graphs with n nodes where the number $m$ of edges is $\Theta(n^2)$. In this case, the sequential complexity of the problem is $\Theta(m + n) = \Theta(n^2)$. Starting with every single node as a component, the algorithm divides the number of components in every iteration by at least two. So $log(n)$ iterations are needed at most to determine how many components are in the graph and to which component each node belongs to. Each iteration needs time $O(log(n))$, therefore the overall time complexity is $O(log^2(n))$. Each component is represented by its node $v_i$ with the smallest index $i$. These representing nodes are called super nodes. The index of a component is the index of its super node. Our goal is to show that the algorithm of Hirschberg et al. works efficiently on the GCA with $n^2$ cells. This seems to be in contradiction to the optimality criterion, but we refer to the discussion in the next section.

Listing 1 shows the original algorithm (reference algorithm) consisting of 6 steps. Each iteration starts with several non connected components. During every iteration, each component searches a connection to another compo-

nent. This is accomplished by listing 1. First every node of the component searches a connection to a node belonging to another component (step 2). If the node can connect to more than one component, the component with the lowest index is selected. Afterwards the super node picks the component with the lowest index (step 3). The components connect to each other and for each new component a super node is chosen (step 4-6).

1. **for** all i **in parallel do** C(i) ← i
   **do** steps 2 through 6 **for** log n iterations
   2. **for** all nodes i **in parallel do**
      T(i) ← $\min_j$ {C(j) |A(i,j)=1 AND
      C(j) != C(i)} **if** none **then** C(i)
   3. **for** all i **in parallel do**
      T(i) ← $\min_j$ {T(j) |C(j)=i,T(j) != i}
      **if** none **then** C(i)
   4. **for** all i **in parallel do**
      C(i) ← T(i)
   5. **repeat for** log n iterations
      **for** all i **in parallel do** T(i) ← T(T(i))
   6. **for** all i **in parallel do**
      C(i) ← min{C(T(i)) ,T(i)}

**Listing 1. Pseudo code for the algorithm of Hirschberg et al. on the PRAM (reference algorithm)**

The original algorithm was defined for the SIMD (single instruction multiple data) parallel processors (e.g. vector machines) [1, 5]. Later the algorithm was investigated for the PRAM machines [3]. All these algorithms use a common memory.

The algorithm uses the following variables and constants: Input is the adjacency matrix $A = \{A(i,j)|i,j = 1 \ldots n\}$. If $A(i,j) = A(j,i) = 1$ then there is a link between node $i$ and node $j$.

$C = \{C(i)|i = 1 \ldots n\}$, $T = \{T(i)|i = 1 \ldots n\}$

$C(i)$ and $T(i)$ are of type integer and hold the number of a node or a super node.

In order to compute the $min$ function in steps 2 and 3 in parallel $n^2$ temporary variables have to be reserved in the common memory. The constant A, the variables C, T and the temporary variables have to be stored in the common memory of the SIMD or PRAM computer.

## 3. Mapping Hirschberg's algorithm on the GCA

Our first insight is the following: If a cell has a constant number of rules, and needs only a constant number of registers of $O(\log n)$ bits to execute the algorithm, then the hardware complexity of a cell itself is asymptotically not more complex than the hardware complexity of a constant number of memory elements. As this prerequisite holds for an implementation of Hirschberg's algorithm, and as the algorithm needs a shared memory of size $O(n^2)$ elements, there is no asymptotic advantage in hardware cost to reduce the number of processing elements below $n^2$.

This presents a conceptional difference to PRAM algorithms where besides the parallel runtime complexity $t_p$, which cannot be less than $\Omega(\log n)$ for most problems, there has been a strive to reduce the number of processing elements $P$ to a level where the work $w = t_p \cdot P$ reaches the sequential time complexity $t_s$ of the problem, i.e. $P = t_s/t_p$. Note that the work cannot be asymptotically less than the sequential time complexity, because Brent's theorem otherwise would suggest a sequential algorithm with runtime less than $t_s$, which is a contradiction. The reason for this definition of work-optimality is that in PRAM algorithmics only processing elements count as cost. Shared memory does not count although shared memory may have much more than $O(P)$ elements. This seems appropriate as processing elements are complex circuits while a memory element of reasonable bit width needs only some hundreds of transistors. In a GCA however, where the algorithm is compiled into reconfigurable FPGA hardware, processing elements, i.e. GCA cells, become cheap, while memory gets more expensive in FPGAs. This motivates our investigation.

**Designing the cell field and the cell data structure.** The first design decision is about the number and the structure of the cells. If plenty of cells are used they can be structured more simply and the execution time can be minimized. For this algorithm we decide between $n$ and $n^2$ cells. We have decided for the $n^2$ case because we want to design and evaluate the GCA algorithm with the highest degree of parallelism.

We use the following data structure: $n^2$ cells $(i,j)$ are arranged in a square matrix. Each cell stores $(a,d,p)$. The data part $d$ is used for the computation of the connected component. It stores node or super node numbers. The pointer $p$ in each cells points dynamically to another cell (so called global cell), and reads from that location the global information $(d^*,p^*)$ which is used for the computation of the next cell state $(d',p')$. Each cell $(i,j)$ also stores in the cell field $a$ the entry $A(i,j)$ of the adjacency matrix.

In addition $n$ cells are necessary to store intermediate results. They form the additional bottom row of the cell matrix. Each of these cells has the data structure $(d,p)$.

If the cells fields are assembled together they form three matrices which are overlaid.

D: $(n+1) \times n$ matrix, P: $(n+1) \times n$ matrix, A: $n \times n$ input matrix

The first column of $D$ corresponds to the vector $C(i)$ and $T(i)$ respectively of the reference algorithm. The last row of $D$ is required to save intermediate results.

**Notation:**
index = linear index of $D$ and $P : 0, 1, \ldots (n^2 + n - 1)$
$j$ = row(index) = row index of $D$ and $P : 0, 1, \ldots n$
$i$ = col(index) = column index of $D$ and $P : 0, 1, \ldots (n-1)$
$D$(index) = $D$ <$j$> $[i]$
$D[i]$ = column $i$ of D
$D$ <$j$> = row $j$ of D
$\bullet$ = for all elements in the row or column

$d$ = the data field of a cell, $d =$ $D$ <$j$> $[i]$
$d^*$ = the data field of a global cell, $d^* = D(P$ <$j$> $[i])$
$p$ = the pointer field of a cell, $p =$ $P$ <$j$> $[i]$
$p^*$ = the pointer field of a global cell, $p^* = P(P$ <$j$> $[i])$

$D^\square$ = the square matrix $D$, the first $n$ rows of $D$
$P^\square$ = the square matrix $P$, the first $n$ rows of $P$
D<$n$> $= D_N$ = last row of $D$

**The Generations of the GCA algorithm.** The six steps of the Hirschberg algorithm have been expanded into 12 generations (Table 1). The actions in each generation are controlled by a state machine (state graph, Figure 2).

The state graph shows on the left the computation of the actual pointer $p$ and on the right the data operation $d \leftarrow \ldots$ for any cell. Note that each cell executes the same uniform algorithm. Some operations depend on the position of the cell. In particular the first column $D[0]$, the last row $D_N$ and the square field $D^\square$ are distinguished by appropriate conditions. The pointer $p$ can either be computed in the current generation, just before the global data $d^* = D(p)$ is accessed, or one generation in advance. In our algorithm the pointer is computed in the current generation (therefore the assignment symbol "=" is used for p in the state graph).

The GCA algorithm will be explained using the original variables $C(i)$ and $T(i)$ as well as the GCA variables $D$ and $P$. Also data parallel assignments will be used where in the GCA algorithm (state graph) p- and d-operations are used.

**Generation 0.** The first step of the reference algorithm requires the data of the vector $C$ to be set to the corresponding index $(C(i) \leftarrow i)$.

In order to keep the GCA algorithm (and the logic in a hardware implementation) as simple as possible, the whole field (instead of the first column) is initialized with the row number of each cell. This is not harmful because in the next generation the values of the remaining field are overwritten.

Initially the data field $d$ of a cell is set to its row number, $d \leftarrow$ row(index). This operation can also be described in a classical data parallel way:

$D$(index) $\leftarrow$ row (index), or $D$ <$j$> $[i] \leftarrow j$. The result of the initialization is

$D =$ 000...
111...
222...
...

**Generation 1.** In order to prepare the field for the calculation of the minimum the vector $C$ (saved in the first column $D[0]$) is copied into each row of the field.

$D$ <$\bullet$> $\leftarrow D[0]$

Thus the vector $C$ is saved in the last row $D_N$, too.

All cells of a column $[i]$ point to the cell in row <$i$> and column [0]: $P$ <$j$> $[i] =$ (<$i$> [0]). The computation $d \leftarrow d^*$ is equivalent to $D$ <$j$> $[i] \leftarrow D(P$ <$j$> $[i])$, or $D$ <$j$> $[i] \leftarrow D$ <$i$> [0].

The function $C$ is set up in steps 2 and 3 (generations 1-8). In step 2 (generations 1-4), each node $i$ examines the component memberships of its neighbors and sets $C(i)$ to the smallest-numbered neighboring component. In step 3 (generations 5-8), each $i \in V_r$ examines its own component members (specified by $D(j) = i$) and picks the smallest-numbered of all the smallest-numbered components that the members found [6]. The only difference between the second and the third step of the algorithm is the condition for the calculation of $C'$ and $T'$ respectively in generation 2 and in generation 6. Thus the generations 7 and 8 are performed similarly to generations 3 and 4.

**Generation 2.** If the condition $A(i, j) = 1$ *AND* $C(j) \neq C(i)$ is fulfilled $d$ remains unchanged, otherwise $d$ is set to $\infty$. That means for the first iteration that $d$ is set to $\infty$ for all diagonal positions in $D^\square$ and for all positions which have a 0-entry in the adjacency matrix. In other words: only 1-entries in the adjacency matrix (connections from node $i$ to other cells) leave the value $i$ of the data field unchanged. The last row of $D_N$ remains unchanged. All cells in row <$j$> of $D^\square$ point to the same cell in row <$n$> and column $[j]$: P <$j$> $[i] =$ <$n$> $[j]$.

**Generation 3.** In this generation all the $\min_j$ functions of step 2 of the Hirschberg algorithm are computed in parallel. The function $\min_j$ is the minimum of all the elements $d$ in each row <$j$>. The technique is tree reduction. This process needs $\log_2(n)$ iterations (sub generations).

**Generation 4.** The last row $D_N$ contains the initial node numbers and remains unchanged.

The first column of $D^\square$ will contain the results of min functions. If the result was $\infty$ (meaning that there are no connections to other components), then $D^\square$ <$j$>[0] is set to $D_N[j]$, the initial node number.

In order to accomplish this operation the pointers of the cells in the first column have been set to the cells in the last row: $P^\square$ <$j$>[0] = (<$n$>$[j]$). The operation is

$D^\square$ <$j$> [0] $\leftarrow D^\square(P^\square$ <$j$> [0]).

**Generation 5.** Similar to generation 1, the vector $T$ (saved in the first column) is copied into each row of the field $D^\square$.

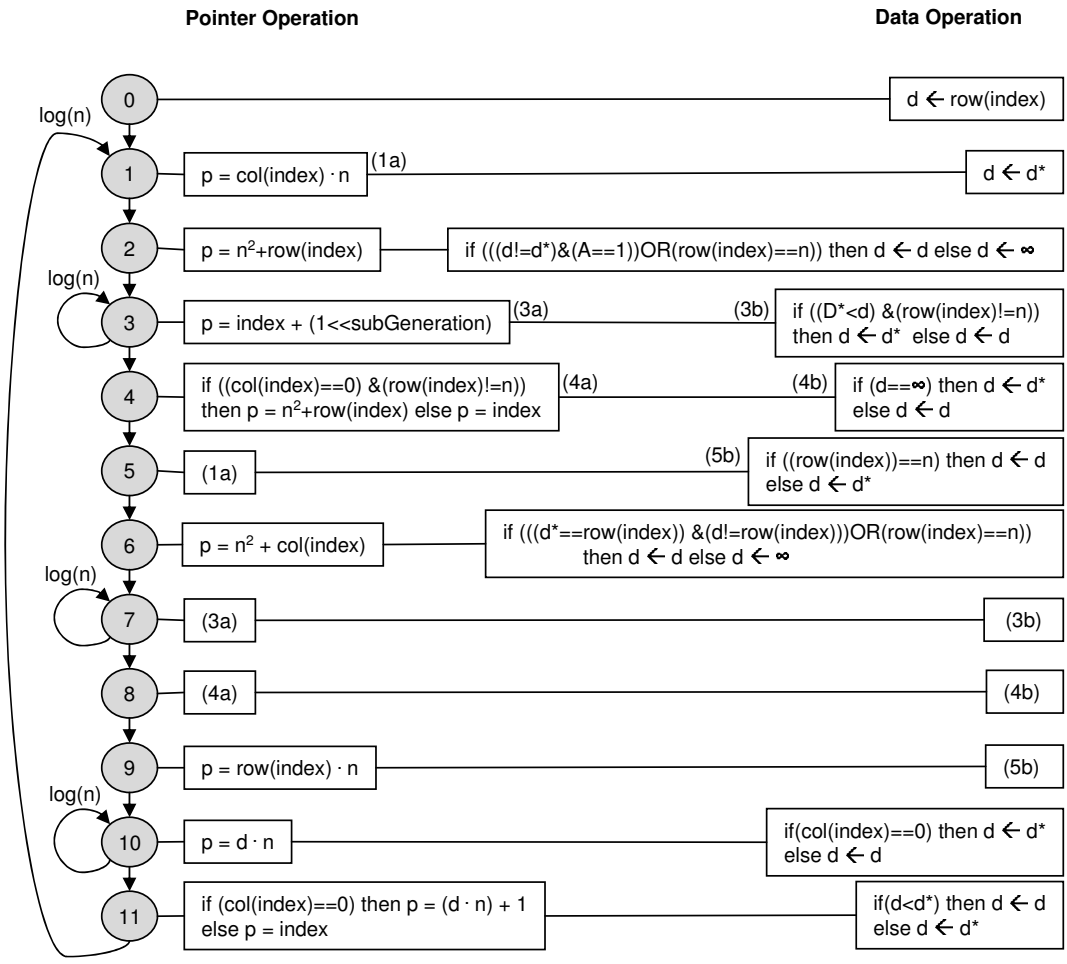


**Figure 2. GCA algorithm with pointer operation (actual access pattern) and data operation.**

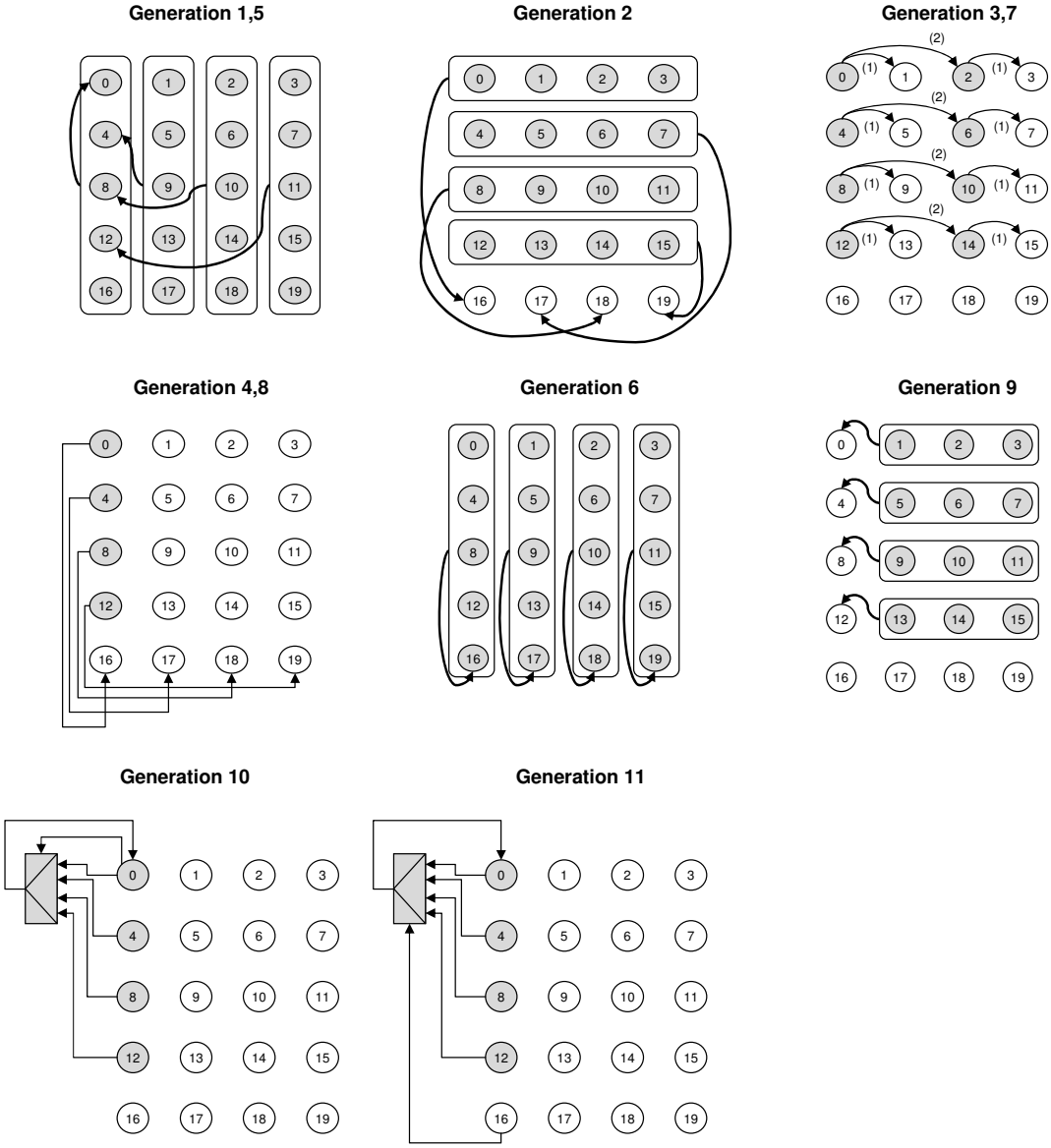


**Figure 3. Access Patterns for $n = 4$. The cell numbers correspond to the linear index. The first four rows form $D^{\square}$, the last row forms $D_N$. Active cells are shaded.**

| STEP | GENERATION | | ACTIVE CELLS (modifying cell state) | # cells with read access | $\delta$ = # of concurrent read accesses (congestion) |
|---|---|---|---|---|---|
| 1 | 0 | Initialization | $n(n+1)$ | 0 | |
| 2 | 1 | | $n(n+1)$ | $n^2$<br>$n$ | 0<br>$n+1$ |
| | 2 | | $n^2$ | $n^2$<br>$n$ | 0<br>$n$ |
| | 3 | $\log(n)$ sub generations, minimum calculation | $n^2/2$ | $(n-1)^2$<br>$n+n$ | 1<br>0 |
| | 4 | | $n$ | $n$<br>$n^2$ | 1<br>0 |
| 3 | 5 | | $n(n+1)$ | see gen. 1 | see gen. 1 |
| | 6 | | $n^2$ | see gen. 2 | see gen. 2 |
| | 7 | $\log(n)$ sub generations, minimum calculation | $n^2/2$ | see gen. 3 | see gen. 3 |
| | 8 | | $n$ | see gen. 4 | see gen. 4 |
| 4 | 9 | | $(n-1)^2$ | $n$<br>$n^2$ | $n-1$<br>0 |
| 5 | 10 | $\log(n)$ sub generations | $n$ | $n$<br>$n^2$ | $n$<br>0 |
| 6 | 11 | | $n$ | $n$<br>$n^2$ | $n$<br>0 |

**Table 1. Generations for each step. Active cells are cells that perform a calculation within a generation. $\delta$ is the number of concurrent read accesses to each of the # cells.**

$D^{\square}$ <•>= $D[0]$

The values of the cells of the last row remain unchanged.

All cells $j$ of a column $[i]$ point to the cell in row <*i*> of column [0] like in generation 1: $P$<*j*> $[i]$ =<*i*> [0]. The computation $d \leftarrow d^*$ is executed for the square matrix $D^{\square}$ which is equivalent to $D^{\square}$<*j*> $[i] \leftarrow D^{\square}(P$<*j*> $[i])$, or $D^{\square}$ <•> $[i] \leftarrow D^{\square}$ <*i*> [0].

**Generation 6.** (Similar to generation 2). In contrast to generation 2 the condition has changed.

All cells in row <*j*> of $D^{\square}$ point to the same cell in row <*n*> and column $[j]$: $P$<*j*> [•] =<*n*> $[j]$. If the condition $(d =$<*j*>$)\&(d^* \neq d)$ is fulfilled $d$ is set to $\infty$. The last row of $D$ remains unchanged.

**Generation 7, 8.** They are identical to generations 3, 4.

**Generation 9.** This generation is similar to generation 5, the only difference is that the first row (vector $T$) is copied to the other columns and not the other rows. A second result is that the vector $T$ is saved in the last row of the field ($D_N$).

$D^{\square}[\bullet] \leftarrow D[0], D_N \leftarrow D[0]$

**Generation 10.** This generation iterates $\log_2 n$ times. Only the first column [0] (corresponding to Hirschberg's $C(i)$ Vector) is modified. The pointers are data dependent. The cell <*j*>[0] points to <*row(d)*> [0]. Thus the neighbor depends on the value of the cell and it is possible to set the value of $C(i)$ to the value of $C(C(i))$ in one generation.

**Generation 11.** Generation 11 is similar to generation 10. In both generations the pointers are data dependent. In addition to the previous generation the value of $C(T(i))$ is compared to the stored value of $T(i)$ in the last row ($D_N$) of the field. The minimum out of both values is saved as the new value for $C(i)$.

$P[0] \leftarrow row(d) + 1$

$D[0] \leftarrow min(d, d^*)$

| STEP OF THE ALGORITHM | GENERATION |
|---|---|
| 1 | 1 |
| 2 | $1 + \log(n) + 1 + 1$ |
| 3 | $1 + \log(n) + 1 + 1$ |
| 4 | 1 |
| 5 | $\log(n)$ |
| 6 | 1 |

**Table 2. Generations per step.**

**Time complexity.** (Figure 2, Table 2) The steps 1, 4 and 6 can be performed in one generation. Steps 2 and 3 each need $1 + 3 \cdot \log(n) + 4$ generations, because the minimum needs $\log(n)$ sub generations. Step 5 needs one generation, but this step is repeated $\log(n)$ times. The steps 2 to 6 are executed in $\log(n)$ iterations. So the total amount of generations is $1 + \log(n) \cdot (3 \cdot \log(n) + 8)$. This is equivalent to a time bound of $O(log^2(n))$ using $n \cdot (n+1)$ processors.

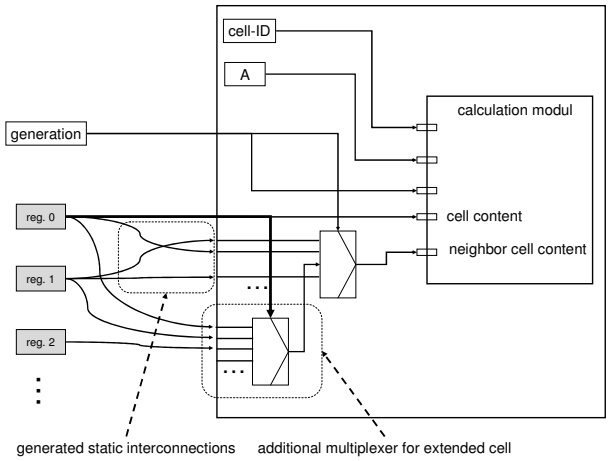


**Figure 4. Cell implementation.**

## 4. Fully parallel hardware implementation

To implement the GCA algorithm of the previous section in hardware, the field is separated into $n^2$ standard cells and $n$ extended cells with the ability to choose the neighbor cell on the basis of the cell data (Figure 4). Except for generations 10 and 11 the neighbor cell is connected statically to each cell and is chosen through a multiplexer addressed by the generation. The extended cells need in addition a second multiplexer with the cell data as address. Every cell saves its own state with a register inside the cell. With a fully parallel hardware where the number of cells is equal to the number of calculation modules, each generation can be calculated in one step. The design was described in Verilog and synthesized for an ALTERA CYCLONE II FPGA. While the congestion suggests that some of the steps are very slow, the static nature of the communication can be used to either implement the concurrent reads in a tree-like manner, or to use replication for arrays $C$ and $T$ to get congestion down to 1. For example, in the second step, each cell $(i, j)$ accesses $C(i)$ and $C(j)$. If the array $C$ is replicated in each row, rotated by $i$ positions in row $i$, then all cells in row $i$ could access all the $C(j)$ values in this row, and each cell of this row could access the $C(i)$ value in its column. This however would require extended cells in all places. One result from the synthesis for the ALTERA CYCLONE II FPGA (EP2C70) with the QUARTUS II software is: $N \times (N+1) = 272$ cells; logic elements = 23,051; register bits = 2,192; clock frequency = 71 MHz.

## 5. Conclusion

We have presented a study how Hirschberg's PRAM algorithm for connected components can be mapped onto a Global Cellular Automaton. By compiling the algorithm into a configurable hardware, the cost per processing element, i.e. cell, is decreasing to a level where it approaches the cost of a small number of memory cells. Hence, the cost of the resulting GCA is dominated by the cost of the $n^2$ memory cells, and thus we can afford to employ appropriate cells, which considerably simplify the implementation. A prototype implementation on a field programmable gate array indicates the feasability of the implementation. The GCA may thus serve as a tool to implement PRAM algorithms on a cost-efficient, yet highly scalable and configurable platform, thus narrowing the gap between theory (PRAMs) and practice (parallel machines). Our future work will comprise the implementation of more elaborate PRAM algorithms.

## References

[1] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982.

[2] C. Ehrt. Globaler Zellularautomat: Parallele Algorithmen. Master's thesis, Technische Universität Darmstadt, 2005.

[3] A. Gibbons and W. Ritter. *Efficient Parallel Algorithms*. Cambridge University Press, New York, Port Chester, Melbourne, Sidney, 1998.

[4] W. Heenes, R. Hoffmann, and J. Jendrsczok. A multiprocessor architecture for the massively parallel model GCA. In *International Parallel and Distributed Processing Symposium (IPDPS), Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS)*, 2006.

[5] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 55–57, New York, NY, USA, 1976. ACM Press.

[6] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.

[7] R. Hoffmann, K.-P. Völkmann, and S. Waldschmidt. Global Cellular Automata GCA: An Universal Extension of the CA Model. In *Worsch, Thomas (Editor): ACRI 2000 Conference*, 2000.

[8] R. Hoffmann, K.-P. Völkmann, S. Waldschmidt, and W. Heenes. GCA: Global Cellular Automata. A Flexible Parallel Model. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 66–73, London, UK, 2001. Springer-Verlag.

[9] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[10] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana and London, 1966.

[11] C. Wiegand, C. Siemers, and H. Richter. Definition of a Configurable Architecture for Implementation of Global Cellular Automaton. 23-26 March 2004.