Hirschberg's Algorithm on a GCA and its Parallel Hardware Implementation

Johannes Jendrsczok[1], Rolf Hoffmann[1], and Jörg 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. We present in detail a GCA (Global Cellular Automaton) algorithm with $3n$ cells for Hirschberg's algorithm which determines the connected components of a n-node undirected graph with time complexity $O(n \log n)$. This algorithm is implemented fully parallel in hardware (FPGA logic). The complexity of the logic and the performance is evaluated and compared to a former implementation using $n(n+1)$ cells with a time complexity of $O(\log^2(n))$. It can be seen from the implementation that the presented algorithm needs significantly fewer resources (logic elements times computation time) compared to the implementation with $n(n+1)$ cells, making it preferable for graphs of reasonable size.

1 Introduction

The GCA (Global Cellular Automata) model [1, 2] is an extension of the classical CA (Cellular Automata) model [3]. 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 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 for the GCA model is much wider than for the CA model.

The CA model suits 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. Typical applications for the GCA model are – besides all CA applications – graph algorithms, hypercube algorithms, logic simulation, numerical algorithms, communication networks, neuronal networks, games, and graphics.

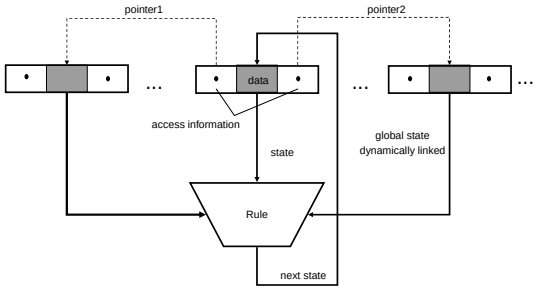


Fig. 1. The operation principle of the GCA.

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. In [5] we have described the hardware implementation of Hirschberg's algorithm to compute the connected components of an n-node undirected graph [6] on the GCA using n^2+n cells. The time complexity was $O(log^2(n))$. We use field programmable gate arrays (FPGAs) as hardware platform.

In this paper we present a different hardware implementation of Hirschberg's algorithm on a GCA using only $3n$ cells with two pointers per cell. This implementation will be evaluated with respect to time complexity and hardware complexity. Also this algorithm will be compared to the GCA algorithm with $n^2 + n$ cells in order to find out the advantages. The FPGA reconfigurability allows to implement different GCAs with very low overhead, thus enabling the use of highly efficient co-processors, as argued in a recent journal issue, see [7].

The remainder of the paper is organized as follows. In Section 2 we sketch relations and differences between PRAMs and GCAs. In Sections 3 and 4, we review Hirschberg's algorithm and present how it can be mapped onto a GCA. Section 5 presents results about our hardware realization. Section 6 concludes.

2 GCAs and PRAMs

The state of a GCA cell consists of a data part and an access information part. In a common implementation 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. Additionally we call the GCA cells *uniform* if all cells have the same transition rule and otherwise *non-uniform*.

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 memory location may only be written by a dedicated processor, the

owner. In principle, the GCA is able to implement any PRAM algorithm, as any algorithm consists of a finite number of instructions 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 particular, Hirschberg's algorithm is well-studied on the PRAM model [8].

A general simulation of CRCW or CREW PRAMs onto a GCA can be achieved by sorting the requests according to owners; efficient sorting is available on owner-write PRAMs [9]. Yet, as the PRAM algorithm for a particular problem can be compiled into the GCA rule set when using reconfigurable hardware, i.e. FPGAs, more efficient methods normally are available.

On a PRAM one seeks an algorithm with a short parallel runtime T_p on a number of processors P with $P \cdot T_p = T_s$, where T_s is the sequential complexity of the problem at hand, i.e. a *work-optimal* algorithm. Either, P is driven to its maximum value in the range of the problem size, to explore the limits of parallelism in a problem, or P is chosen so that a practical implementation is available. While the latter case in general can be derived from the first one via Brent's theorem, often direct methods lead to simpler and faster implementations. In a previous paper [5], we have investigated the first case, here we investigate the latter, for the reasons given below.

If the number of processors is taken as a measure of machine *cost* or *price*, and parallel time (for a fixed work) is seen as the inverse of performance, then $P \cdot T_p$ represents a price/performance ratio. While the cost measure may be appropriate for PRAMs because even RISC microprocessors are much more complex than memory cells, in GCAs the memory cost has to be taken into account, because a finite automaton with a few registers is cheap in reconfigurable hardware, while memory cost is comparatively high. This means that the price of a GCA requiring n^2 memory cells does not vary much no matter if one takes n^2 or $n^2/\log^2(n)$ processors. This enables further simplification of algorithms.

Yet, our feeling was that employing n instead of n^2 processors, and giving each a local memory of size n, not implemented in registers as before but in much denser RAM storage, might still improve the price-performance ratio, and give processor counts that become practical.

Implementing a CROW PRAM algorithm or a GCA requires similar considerations. The memory is mapped to the owners. For non-local read accesses, the *congestion*, i.e. the number of cells reading from one cell, has to be controlled. In the case that we investigate here, where each GCA cell only has a single data value to be accessed, congestion can only occur because of concurrent reading. Yet, this can be ameliorated by appropriate routing networks, such as Ranade's butterfly network. Hence, we will list the congestion numbers in our result table, but not deal further with the congestion and routing problem.

3 Hirschberg's Algorithm

Our example application is Hirschberg's well known algorithm [6] to compute the connected components of an undirected graph on a CREW PRAM. Yet,

only a CROW PRAM is really needed. Hirschberg's algorithm was seminal and is work-optimal for dense graphs, i.e. graphs with n nodes and $m = \Theta(n^2)$ edges where 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. Each iteration needs time $O(\log n)$ on $n^2/\log^2(n)$ processors, 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 $3n$ cells, for the reasons given in the previous section. Our implementation will need $O(n \log n)$ steps, hence we will see a speedup for graphs with $m = \Omega(n \log n)$ edges, and see maximum speedup for graphs with $m = \Theta(n^2)$ edges. Examples of such graphs appear e.g. when very large graphs are collapsed into smaller ones.

Listing 1.1 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 component. 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 AND 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 1.1. Pseudo code for the algorithm of Hirschberg et al. on the PRAM (reference algorithm)

The original algorithm was defined for SIMD (single instruction multiple data) parallel processors (e.g. vector machines). Later the algorithm was investigated for the PRAM machines [8]. 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. $C(i)$ and $T(i)$ are of type integer and hold the number of a node or a super node: $C = \{C(i)|i = 1 \ldots n\}$, $T = \{T(i)|i = 1 \ldots n\}$. 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.

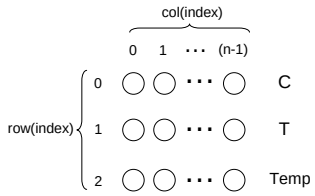


Fig. 2. GCA Field.

4 Hirschberg's Algorithm on the GCA

The GCA algorithm uses a cell array Z in order to store the variables and computational rules. The cell array Z consists of three rows with n cells each (Fig. 2):

Row 0 of Z corresponds to the original vector $C(i)$: $Z0 = C = C_0 \dots C_{n-1}$.

Row 1 of Z corresponds to the original vector $T(i)$: $Z1 = T = T_0 \dots T_{n-1}$.

Row 2 of Z is used to hold temporary results $Temp(i)$: $Z2 = Temp = Temp_0 \dots Temp_{n-1}$, as well as the matrix A, one column per cell. The matrix columns are only accessible by the cell holding them[3].

The cells of Z are ordered by a linear index $K = 0 \dots 3n-1$. A cell $z = Z(K)$ will be accessed from another cell using the linear index K. For convenience a cell $z = Z(J, I) = Z(K)$ may also be accessed by the row index $J = row(K)$ and the column index $I = column(K)$ using the access functions *row* and *column*.

Each cell $z = (d, p0, p1)$ consists of a data part d and two pointers $p0$ and $p1$. The data part is used for the computation of the connected components storing the node or super node numbers. The pointers dynamically establish links to two other cells (global cells $z^*(p0), z^*(p1)$). The global data is denoted as $d^*(p0)$ and $d^*(p1)$. In general the next cell state $z' = (d', p0', p1')$ depends on its current state z and the states $z^*(p0)$, $z^*(p1)$ of its current neighbors.

A GCA algorithm consists of a sequence of parallel computations. In each computation all cells update their state in parallel in accordance to the local rule. The global state (configuration) of the GCA is given by the cross product of all the local cell states. The configuration changes from time step to time step. In order to emphasize the changing of the configurations with time g the term *generation* is commonly used. The configuration at time g is the g^{th} *generation*.

A GCA algorithm can be clearly represented by a state graph. The state graph consists of states which are reached under certain conditions, e.g. central counter states. In each state two types of operations are performed: *data operations* and *pointer operations*.

The state graph (Fig. 4) shows on the left the computation of the actual pointer p and on the right the data operation of a cell. The pointer p can either be computed in the current generation or one generation in advance. In our algorithm the pointer is computed in the current generation to be used immediately

[3] If the degree of the graph is known to be low, the matrix columns can be replaced by lists.

Step	State	Active Cells (modifying cell state)	# cells with read access	δ = # of concurrent read accesses (congestion)	N algor.		N^2 algor.	
					Gen.	Sub-gen.	Gen.	Sub-gen.
1	0	n	0	0	1		1	
2	1	n	0	0	$2+n$	n	$3+\log n$	$\log n$
	2	n	$n+2n$	$2+0$				
	3	$2n$	$n+2n$	$0+1$				
3	4	n	$n+2n$	$0+1$	$1+n$	n	$3+\log n$	$\log n$
	5	$2n$	$n+2n$	$0+1$				
4	6	n	$n+2n$	$1+0$	1		1	
5	7	n	$n+2n$	$n+0$	$\log n$	$\log n$	$\log n$	$\log n$
6	8	n	$n+2n$	$n+0$	1		1	

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

in the data operation. Therefore the assignment symbol "=" is used for pointer operations. In contrast the synchronous assignment symbol "$\leftarrow$" is used for the data operations.

Although in principle each cell obeys to the same uniform algorithm, the operations to be performed may depend on certain conditions. In this algorithm the data operations depend on the positions of a cell in the field, in particular whether a cell is located in C, T or $Temp$. The conditions to distiguish between the three vectors are: $row(index) = 0$ for C, $row(index) = 1$ for T, and $row(index) = 2$ for $Temp$. The GCA algorithm (Fig. 4) consists of 8 states which correspond to the 6 steps of the original algorithm as shown in Table 1.

State 0. The first step of the reference algorithm requires the data of the cell vector C to be set to the corresponding index ($C(i) \leftarrow i$). So the data of the vector C is initialized with the column number of each cell.

State 1. In order to prepare the field for the calculation of the minimum in the next state the vector $Temp$ is set to ∞. Thereby it is possible to identify whether a minimum will have been found in state 2 or not.

State 2. In this state all the $\min_j$ functions of the Hirschberg algorithm are computed in parallel. If the condition $A(i,j) = 1\ AND\ C(j) \neq C(i)$ is fulfilled and $Temp(i)$ is less than $C(i)$, $Temp(i)$ is set to $C(i)$, otherwise $Temp(i)$ remains unchanged. Thus the data of the vector $Temp(i)$ is the minimum of $C(j)$ after n iterations.

In the corresponding GCA algorithm (Fig. 4) each cell is operating on its own and the operations specified in the graph tell each cell what it has to do. In state 2 only the last row ($Temp$) of the cell array with $row(index) = 2$ is activated. Each cell $T(I)$ computes the minimium of the cell $C(I)$ compared to all cells $C(J)$. The pointer $p0$ is used to access $C(I)$ and the pointer $p1$ is successively incremented (using the subgeneration counter) in order to access all J cells (see access pattern Fig. 3).

State 3. After the calculation of the minimum $\min_j$ the value of $C(i)$ is written back in case none of the conditions of generation 2 was true. For this

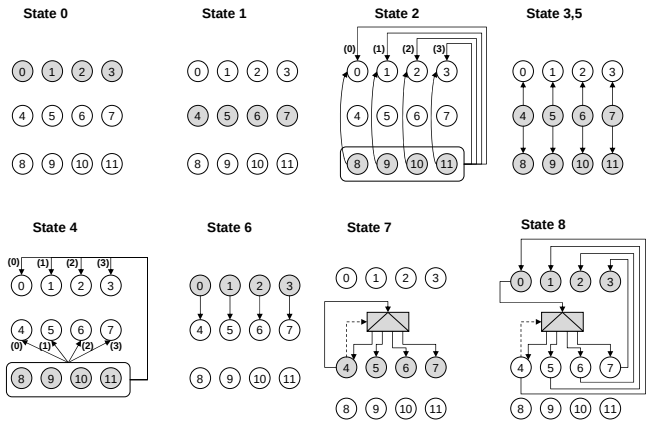


Fig. 3. Access Patterns for $n = 4$. Active cells are shaded.

purpose $Temp(i)$ is substituted for the value of $C(i)$ in case $Temp(i)$ equals ∞. Then the value of $Temp(i)$ is set to ∞ in preparation of the next generation.

The cell $T(I)$ uses its pointer $p0$ to access $Temp(I)$ and its pointer $p1$ to access $Temp(I)$. If $p0 = \infty$ then $p1$ is copied to the data part d of $T(i)$, otherwise $p0$ is copied.

State 4. (Similar to state 2). In contrast to State 2 the condition has changed. If the condition $C(j) = i$ *AND* $T(j) \neq i$ is fulfilled and $Temp(i)$ is less than $T(j)$, $Temp(i)$ is set to $T(j)$, otherwise $Temp(i)$ remains unchanged. Thus the data of the vector $Temp(i)$ is the minimum of $T(j)$ after n iterations.

State 5. State 5 is identical to state 3.

State 6. Vector T is copied to vector C.

State 7. This state iterates $\log n$ times. Only the vector T (corresponding to Hirschberg's $T(i)$ Vector) is modified. The pointers are data-dependent. The cell $T(i)$ points to the cell $T(T(i))$. Thus the neighbor depends on the value of the cell and it is possible to set the value of $T(i)$ to the value of $T(T(i))$ in one parallel computation.

State 8. State 8 is similar to state 7. In both states the pointers are data dependent. In addition to the previous state the value of $C(T(i))$ is compared to the stored value of $T(i)$. The minimum out of both values is saved as the new value for $C(i)$.

Time complexity. (Fig. 4, Table 1) The steps 1, 4 and 6 can be performed in one generation. Each of the steps 2 and 3 need $1 + n + 1$ respectively $1 + n$ generations, because the computation of the minimum needs n sub generations. Step 5 is repeated $\log n$ times.

The steps 2 to 6 are executed in $\log n$ iterations. So the total amount of generations is $1 + \log n \cdot (5 + 2n + \log n)$. This corresponds to a time bound of $O(n \log n)$ using $3n$ cells. In a previous GCA implementation [5], $n(n + 1)$ cells were used in order to execute the algorithm as fast as possible. There the

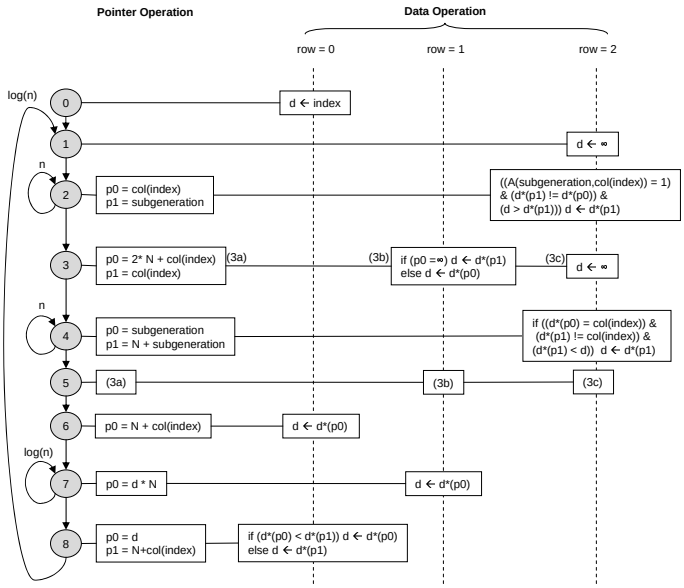


Fig. 4. GCA algorithm with pointer operation (actual access pattern) and data operation

minimum function takes only $\log n$ generations instead of n as presented here. Therefore the total amount of generations was $1 + \log n \cdot (3 \log n + 8)$. This corresponds to a time bound of $O(\log^2(n))$ using $n(n + 1)$ cells. In order to distinguish the two algorithms, the algorithm with $3n$ cells is also denoted as "N algorithm" and the algorithm with $n(n + 1)$ cells as "N^2 algorithm".

5 Fully Parallel Hardware Implementation

We have implemented the two GCA algorithms with $3n$ cells (Fig. 6) and with $n(n + 1)$ cells in hardware (FPGA logic) in order to find out the complexity and efficiency. The platform was the ALTERA Quartus synthesis tool and the Stratix II FPGA (EP2S180). Results from the synthesis are shown in Table 2 and Figure 6.

It turned out that the states 7 and 8 are the same in the N and N^2 algorithm. Therefore the synthesis was splitted into three parts: (1) states 0-6 for the N algorithm, (2) the corresponding states for the N^2 algorithm and (3) the states 7-8 for both algorithms (abbreviated N/N^2). If we assume that the register bits have relatively low implementation cost compared to the logic we can focus our comparison on the used logic elements (ALUTs). For the problem size $n = 64$ the number of ALUTs needed to implement the states 0-6 are 1,853 for the N algorithm, and 56,012 for the N^2 algorithm whereas the calculation time counterwise is 5.2 μs (N) and 1.2 μs (N^2). Multiplying the number of ALUTs with

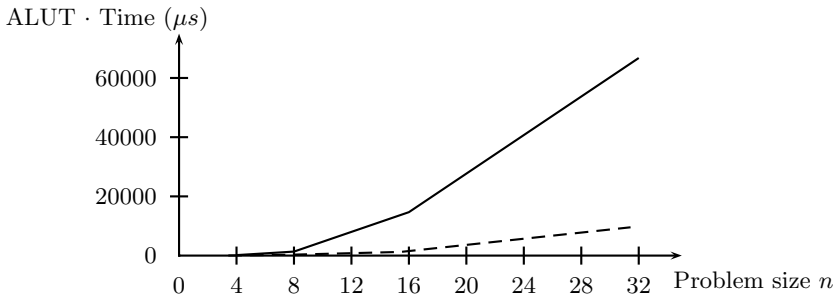


Fig. 5. Resources vs. n (dashed: N algorithm, solid: N^2 algorithm).

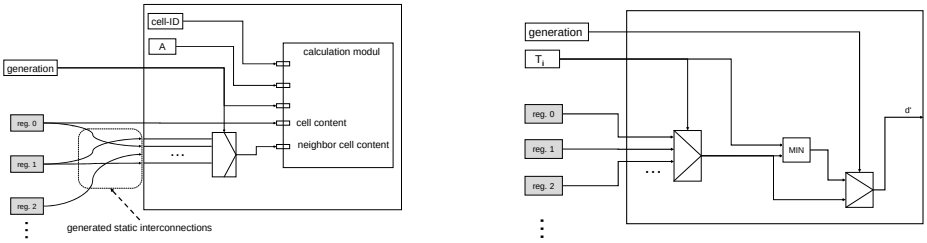
the calculation time gives us a good measure which corresponds to the resource allocation needed to perform the algorithm. We call that measure *resources* for short. It can be seen from Fig. 5 that the *resources* of the N algorithm (states 0-6) are significantly lower compared to the N^2 algorithm. Therefore the N algorithm is more economic with respect to the consumption of resources whereas the N^2 algorithm can produce the result faster.

6 Conclusion

We have presented a GCA algorithm with $3n$ cells for Hirschberg's algorithm to compute the connected components of a directed graph. The algorithm consists of 8 states in which the appropriate operations on the pointer and the data parts of the cells are performed in parallel. The time complexity is $O(n \log n)$. A former GCA algorithm with $n(n+1)$ cells can compute the required minimum function, which is the most time consuming part of the whole algorithm in $\log n$ time. Thereby the time complexity can be reduced to $O(\log^2(n))$. Both algorithms were implemented in hardware (FPGA logic) and evaluated. If the allocated resources which have to be allocated over time (in terms of logic elements $\times$ computation time) are used as a metric then the algorithm with $3n$ cells has showed a 5 to 11 times better performance for $n = 4 \ldots 32$ than the algorithm with $n(n + 1)$ cells.

References

1. Hoffmann, R., Völkmann, K.P., Waldschmidt, S.: Global Cellular Automata GCA: An Universal Extension of the CA Model. In: Worsch, Thomas (Editor): ACRI 2000 Conference. (2000)
2. Hoffmann, R., Völkmann, K.P., Waldschmidt, S., Heenes, W.: GCA: Global Cellular Automata. A Flexible Parallel Model. In: PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies, London, UK, Springer-Verlag (2001) 66–73



(a) Module for state 0-6 (b) Module for state 7,8

Fig. 6. Hardware modules.

CASE	PROBLEM SIZE	CELLS	LOGIC ELEMENTS (ALUT)	f_{max} (MHz)	REGISTERS	# CLOCK CYCLES	CALCULATION TIME (ns)
N	4	12	152	126	48	25	197
	8	24	372	88	120	61	692
	16	48	722	84	288	145	1719
	32	96	1853	65	672	341	5176
N^2	4	20	575	82	80	23	279
	8	72	2570	75	360	40	528
	16	272	12328	51	1632	61	1194
	32	1056	56012	71	7392	86	1195
N/N^2	4	4	18	400	8	6	15
	8	8	172	138	32	12	86
	16	16	653	112	80	20	177
	32	32	4888	57	192	30	523

Table 2. Synthesis results, case N and N^2 stand for state 0-4, N/N^2 stands for state 5 and 6 and is needed for both algorithms

3. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Urbana and London (1966)
4. Heenes, W., Hoffmann, R., Jendrsczok, J.: 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. Jendrsczok, J., Hoffmann, R., Keller, J.: Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton. In: International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Advances in Parallel and Distributed Computational Models (APDCM). (2007)
6. Hirschberg, D.S.: 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, New York, NY, USA, ACM Press (1976) 55–57
7. Buell, D., El-Ghazawi, T., Gaj, K., Kindratenko, V.: Guest editors' introduction: High-performance reconfigurable computing. Computer **40**(3) (2007) 23–27
8. Gibbons, A., Ritter, W.: Efficient Parallel Algorithms. Cambridge University Press, New York, Port Chester, Melbourne, Sidney (1998)
9. Lin, D., Dymond, P.W., Deng, X.: Parallel merge-sort algorithms on owner-write parallel random access machines. In: Europar '97. (1997) 379–383