Global Cellular Automata: a Path from Parallel Random Access Machines To Practical Implementations[*]

Jörg Keller[1] and Andre Osterloh[2**]

[1] FernUniv. in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany
joerg.keller@fernuni-hagen.de
[2] Techn. Univ. Darmstadt, Dept. of Computer Science, 65289 Darmstadt, Germany
osterloh@andre-osterloh.de

Abstract. The Parallel Random Access Machine (PRAM) and the Global Cellular Automaton (GCA) are both models to study parallel algorithms. We investigate the commonalities and differences between these models, because we think that they could nicely complement each other: PRAMs have lots of algorithms, GCAs provide efficient implementation paths. We provide the following results: GCAs are CROW PRAMs, GCAs have more advantageous optimality criteria than PRAMs, and GCA implementations are more efficient than typical PRAM emulations.

Key words: PRAM simulation, Global Cellular Automaton, Optimality Criteria

1 Introduction

The Parallel Random Access Machine (PRAM) is an established model to study parallel algorithms without consideration of details such as interconnection network or synchronization cost. A number of processors work synchronously on a shared memory with unit access time. There are several variants that differ on the access patterns that are allowed, denoted by XRYW, where $X, Y \in \{C, E, O\}$ denote the variant for read (R) and write (W) access, respectively. C means concurrent, i.e. several processors may access the same memory cell in one step. E means exclusive, i.e. only one processor may access a cell in each step. O means owner, i.e. only the owner of a cell may access this cell. While many PRAM algorithms have been developed, and several schemes have been derived to emulate a PRAM on a message-passing machine including hardware prototypes, PRAM algorithms have not found their way yet into practical parallel programming. This hurts more and more as parallel programming is becoming commonplace with multicore processors being the standard for PCs and notebooks. For details on PRAMs and their emulation, we refer to [1,2,3,4].

[*] Research supported by German Science Foundation (DFG) grant HO 893/7-1.
[**] Now with BTC AG, Escherweg 5, 26121 Oldenburg, Germany.

The global cellular automaton (GCA) [5] is an extension of the classical cellular automaton (CA) model. In the GCA, a collection of finite state automata (FSA) called nodes or cells update their local states synchronously where each FSA uses as input its own current state and the current state of its neighbor(s). While the neighbors are fixed by some geometrical rule in the CA, neighbors in the GCA are not fixed by physical interconnect but neighbors are identified by addresses which are part of the local state and thus can vary with node ID or over time. The number of addressable neighbors per node is a parameter of a particular GCA, and typically denoted as the number of hands, i.e. in a one-handed GCA, each node can only address one neighbor. A number of techniques to implement GCAs in reconfigurable hardware (FPGA) or non-configurable hardware (ASIC) have been developed: e.g. fully parallel [6] or data-parallel [7]. A programming and synthesis environment is available [8], i.e. there is a high-level programming language to formulate GCA algorithms, a simulator to debug them, and there exists a compiler producing VHDL code that can be synthesized into an FPGA configuration by a silicon compiler. As the reconfigurability together with programmability allows highly adapted and optimized implementations of parallel algorithms, the GCA seems a promising candidate for a highly efficient massively parallel computing platform. Yet, the number of algorithms developed for the GCA model is still rather small.

In the present paper, we investigate the commonalities and differences between both models, because they could nicely complement each other: PRAMs have lots of algorithms, GCAs provide mature and efficient implementation paths. We give our results in three parts. First, we compare GCAs and PRAMs and show how a PRAM can be simulated by a GCA. As a consequence the wealth of PRAM algorithms is accessible to GCAs. One further result of this comparison is that there is a one-to-one correspondence between a one-handed GCA and a CROW-PRAM, i.e. a GCA can be seen as a CROW PRAM. Second, we show that GCAs have more advantageous optimality criteria than PRAMs, which allows to use simpler PRAM algorithms without penalty. Third, we show that GCA implementations are more efficient than typical PRAM emulations. The similarities between CROW PRAM and GCA and aspects of optimality criteria have been mentioned rather informally in [6].

The remainder of this paper is organized as follows. In Sect. 2 we describe how one step of a PRAM can be simulated on a GCA and give upper and lower bounds. In Sect. 3 we discuss optimality criteria for PRAMs and GCAs. In Sect. 4 we describe optimal PRAM emulations via GCAs. Finally, in Sect. 5 we give a conclusion.

2 Correspondence of PRAMs and GCAs

In this section we give lower and upper bounds for the number of steps necessary to simulate t steps of a PRAM on a GCA. A detailed description of the GCA model can be found in [5]. For a description of the PRAM model see [1]. For the rest of this section we assume the instruction sets of the PRAM and the GCA to be equal.

First, we describe a simulation of one step of a priority CRCW PRAM on a GCA. In contrast to a PRAM, a GCA does not have a common memory. Hence, to simulate a PRAM by a GCA we have to simulate the access to the memory. To distinguish between PRAMs and GCAs we call the storage locations of a GCA cells. Note that cells are able to execute instructions. Since all GCAs, even the 1-handed ones, are able to read from all their cells simultaneously, a simulation of reading access to the PRAM memory is no problem. Hence we focus our attention to simulate writing access. Here it is important to remember that a cell is not allowed to write on other cells.

We begin with the description of a simple simulation of one step of a PRAM. Then we improve the simulation to our final result. We simulate a step of a priority CRCW PRAM with p processors and m storage locations. In a priority CRCW PRAM writing conflicts are solved by using priorities. If two or more processors want to write into the same storage location the processor with the highest priority wins. We assume for simplicity and without loss of generality that the priority of a processor is given by its number, i.e. the processor with the highest number has the highest priority. We simulate the PRAM by a 1-handed GCA with $p \cdot m$ cells $c_{i,j}$, $1 \leq i \leq p$, $1 \leq j \leq m$. The basics behind the simulation are:

(1) cell $c_{i,1}$, $1 \leq i \leq p$, simulates processor i
(2) cell $c_{1,i}$, $1 \leq i \leq m$, simulates storage location i
(3) cells $c_{1,i}, \ldots, c_{p,i}$, $1 \leq i \leq m$, are used to solve writing conflicts on storage location i

This situation is depicted in Fig. 1.

Writing on a storage location i, $1 \leq i \leq m$, is simulated by the following three steps:

(1) For all j, $1 \leq j \leq p$: If processor j wants to write on storage location i, the cell $c_{j,i}$ stores value i otherwise it stores value 0.
(2) Cells $c_{1,i}, \ldots, c_{p,i}$ calculate which processor writes on i. Since the processor with the highest number has the highest priority, this can be done by calculating the maximum of the stored values in $c_{1,i}, \ldots, c_{p,i}$. Assume that k, $1 \leq k \leq p$, is the calculated maximum. Then k is stored in $c_{1,i}$.
(3) $c_{1,i}$ reads the value processor k wants to write into storage location i.

In step (1) cell $c_{j,i}$ reads on cell $c_{j,1}$. This can be done in $O(1)$ steps. In step (2) a maximum of p values is calculated. This can be done in $O(\log p)$ steps [9]. So we get a first result

Lemma 1. *One step of a p-processor priority CRCW PRAM with m storage locations can be simulated by a 1-handed GCA with $p \cdot m$ cells in $O(\log p)$ steps.*

One is able to reduce the resources from $p \cdot m$ cells, each containing a polylogarithmic number of bits, to $p + m$ cells containing $p \cdot m$ bits in total. To do this

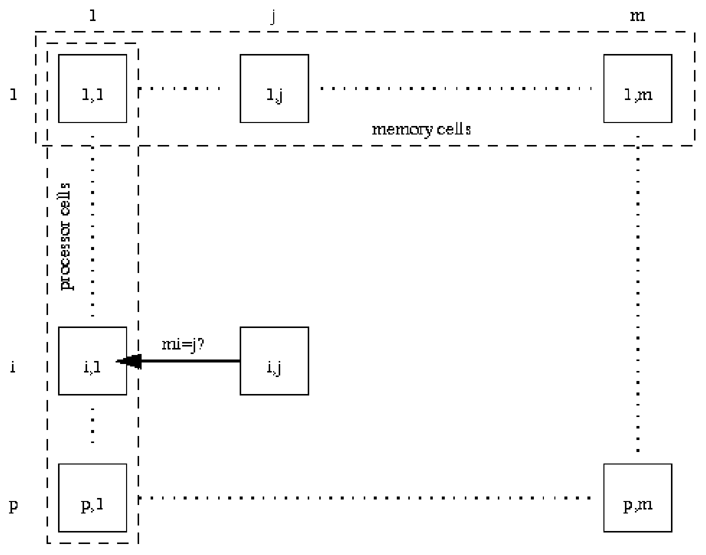

Fig. 1. Simple GCA to simulate concurrent writes.

we have to assume that all processor cells of the GCA are able to store at least m bits. The idea is the following: we take a balanced binary tree of cells. We call the root of the tree r. In this tree the p cells simulating the processors are the leaves. Such a tree consists of $2p-1$ cells. Each cell of the tree is equipped with a register of m bits. Additionally m other cells exist. These cells simulate the m storage locations. If processor i wants to write on storage location j, the leaf with number i sets the j-th bit of its register. Using a standard technique we calculate in $O(\log p)$ steps in the register of r on which memory locations we want to write, i.e. the i-th bit in the register of r is set if and only if one processor cell wants to write to storage location i. This situation is depicted in Fig. 2. Now each of the m cells is able to calculate in one step by testing the register in cell r if it shall be written. If so, it can calculate in $O(\log p)$ steps which processor cell wants to write to it, by following the path from root r to the appropriate leaf processor cell, which is a standard tree search technique. The memory cell then reads the value to be written from that processor cell, and stores it locally. Hence we get:

Theorem 1. *One step of a p-processor priority CRCW PRAM with m storage locations can be simulated in $O(\log p)$ steps of a 1-handed GCA with $O(p+m)$ cells holding $O(p \cdot m)$ bits in total.*

We are also able to reduce the number of cells from $p \cdot m$ to $O(p+m)$, each containing only a polylogarithmic number of bits, by paying with some extra simulation time. To do this, we employ a GCA with p cells representing the PRAM processors and m cells representing the memory locations. In the PRAM

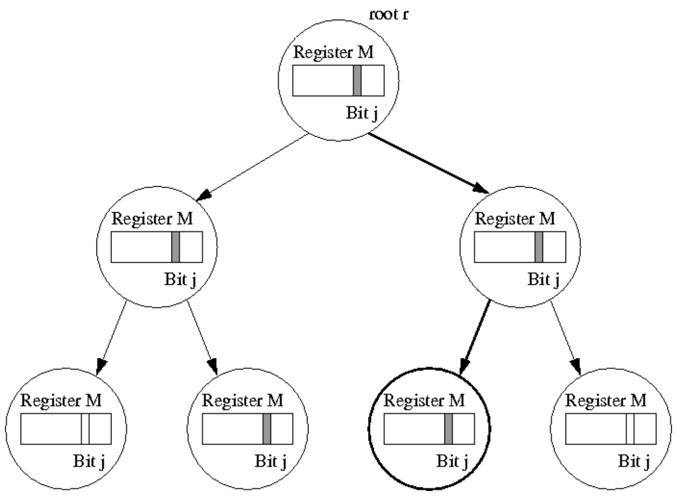

Fig. 2. Tree of GCA cells for concurrent write.

step to be simulated, each processor P_i that wants to write value v_i to shared memory location a_i generates a tuple

$$(a_i, i, v_i)\ ,$$

while other processors choose $a_i = \infty$. The processors sort these tuples, which can be achieved with the help of bitonic sort in $O((\log p)^2)$ GCA steps. To handle concurrent writes, each processor P_i compares its local tuple to the tuple in P_{i+1}. If both tuples contain the same address, then processor P_i changes its local address to ∞, so that only one write to each memory location remains, originating from the processor with highest ID, i.e. highest priority. Now the processors sort these tuples again. Then, each memory cell can apply a binary search among the tuples to check if its address is present. If so, the memory cell performs the write. This takes $O(\log p)$ GCA steps. Thus we arrive at the following theorem.

Theorem 2. *One step of a p-processor priority CRCW PRAM with m storage locations can be simulated by a 1-handed GCA with $O(p+m)$ cells in $O((\log p)^2)$ steps.*

Now we show that we generally cannot reduce the number of steps necessary to simulate one step of a PRAM below a logarithmic bound.

Theorem 3. *Let $a \in \mathbf{N}$. An a-handed GCA needs $\Omega(\log_{a+1} p)$ steps to simulate one step of a CREW PRAM with p processors. This result is independent from the instruction set and the number of cells of the CGA.*

Proof. Given is an a-handed GCA G. The task is to calculate a value that depends on p inputs. The inputs are given in cells $c_1, \ldots, c_p$. The result has to be stored in cell c_{p+1}. In the following we show that G needs $\Omega(\log_{a+1} p)$ steps to calculate the result in cell c_{p+1}.

Before the calculation of the result starts, the content of c_{p+1} depends on no other cell. Since G is a-handed, the state of each cell, in particular c_{p+1}, depends at most on $a+1$ cells (its own state and a other cells) after the first step of a calculation. After $t \geq 2$ steps of a calculation, the state of c_{p+1} depends on the cells it depended on after step $t-1$, and additionally on the cells it accessed in step t. So the maximum number x_t of cells that c_{p+1} depends on after t steps is given by

$$x_1 = a + 1 \tag{1}$$

$$x_t = x_{t-1} + a \cdot x_{t-1},\ t \geq 2\ . \tag{2}$$

So $x_t = (a+1)^t$. Since the input is given in cells $c_1, \ldots, c_p$ and the result depends on all of them, the calculation can only be finished for a t with $(1+a)^t \geq p$, i.e. the desired result.

Finally we shortly discuss why CROW PRAMs are GCAs. We have seen that the simulation of a read access on a storage location is no problem. Since processors of a CROW PRAM are only allowed to write on a storage location where they have owner rights and furthermore each storage location has at most one owner, no write conflicts can occur. So the storage locations for which a processor is owner can be coded in the state of the cell that simulates the processor. In order to arrive at GCA cells with asymptotically equal state sizes, one must further request that each PRAM processor is owner of $O(m/p)$ memory locations.

We also assume that in a p-processor PRAM with m memory locations, normally m is polynomial in p and each memory location contains a number of bits polylogarithmic in m.

3 Optimality Criteria in PRAMs and GCAs

While the computational power of GCAs and PRAMs is equal (see previous section), there are still some differences. In a PRAM, one counts the number of processors as cost, but does not take the memory size and cost into account. This may result from the impression that a processor is a very complex circuit comprising more than 10^9 transistors, while memory cells are incredibly cheap, involving about one transistor per bit for a dynamic RAM cell, or four transistors for a static RAM cell. In contrast, in a GCA each node has a state that comprises both processor state and memory state, and has some circuits for state transitions from cycle to cycle. Thus, counting the nodes of a GCA involves both processing and storage cost, with the ratio between cost per processor and cost per memory bit being very much smaller than in the PRAM model. As a GCA algorithm to be executed is directly encoded into the nodes' finite state machines, a GCA node typically contains only few hundred logic gates for processing. Thus, memory makes out a considerable fraction of the node cost.

This difference becomes apparent when we consider the processor-time product (PTP) of a parallel algorithm. When using a PRAM, the PTP is the product of the parallel algorithm's runtime t_p on a PRAM and the number P of processors of the PRAM. The ratio behind this definition is the ability to compare the parallel algorithm to the sequential complexity t_s of the problem to be solved. The PTP cannot be asymptotically smaller than this sequential complexity. If we assume that we would have a parallel algorithm with a smaller PTP, then we could simulate a PRAM with this algorithm sequentially, instruction by instruction of all PRAM processors, and would arrive at a sequential algorithm with a runtime that asymptotically equals the PTP, which contradicts the above assumption. If the PTP equals the sequential complexity, i.e. if

$$t_p \cdot P = O(t_s) \ , \tag{3}$$

then the parallel algorithm is *PTP-optimal*. Consequently, there has been a strive in PRAM algorithmics to find PTP-optimal parallel algorithms that use as many processors as possible.

Yet, these are two conflicting goals. A parallel algorithm without concurrent write that takes input of size n into account completely has a parallel runtime of at least $t_p = \Omega(\log n)$. Because of Eq. (3), there exist only PTP-optimal parallel algorithms using at most

$$P = O(t_s/t_p) = O(t_s/\log n) \tag{4}$$

processors. As in many problems, especially the ones with linear time complexity, the memory consumption is $\Theta(t_s)$, the memory size is asymptotically larger than the processor count, but is not taken into account.

Creation of a PTP-optimal algorithm often occurs in two steps: there is a parallel algorithm that achieves optimal parallel runtime, but needs more processors than given in Eq. (4), typically on the order of the sequential complexity, i.e. $P = O(t_s)$. Then one modifies this algorithm, for example by reducing the problem size by a factor of $\log n$ in a preprocessing step, and then one applies the non-optimal algorithm on the reduced problem. As $\log(n/\log n) = \Theta(\log n)$, the algorithm now is PTP-optimal. Yet, the construction of a PTP-optimal algorithm may be much more elaborated, so that it would be advantageous to be able to restrict to the non-optimal algorithms which are much simpler. Also, in this case, memory size and processor count would be proportional.

As an example, consider the problem of computing the sum of n numbers. A very simple PRAM algorithm with $P = n/2$ processors repeatedly adds two of the numbers with each processor, until the sum remains, see the following pseudocode.

```
int numbers[n];
int sum;

simplesum(int n){ // proc. count equals array size
  for(t=log(n)-1;t>=0;t--){
    if(PID<(1<<t)){ // use 2^t proc.s in step t
      numbers[PID] = numbers[2*PID] + numbers[2*PID+1];
    }
  }
  sum = numbers[0];
}
```

This simple algorithm uses $P = n/2$ processors to sum n numbers in time $t_p = O(\log n)$. Thus it is time-optimal as the sequential complexity is $t_s = O(n)$, but not PTP-optimal. If we can reduce our problem size to $n/\log n$ in parallel time $O(\log n)$ with $n/\log n$ processors, then the simple algorithm can be applied with $n/\log n$ processors in time $O(\log(n/\log n)) = O(\log n)$. The resulting algorithm is PTP-optimal. Such a preprocessing is easily achieved by having each of the $n/\log n$ processors simply sum up $\log n$ numbers, see the following pseudocode.

```
int numbers[n];
int sum;

optsum(int n){ // proc. count is only n/log n
  for(i=1;i<log(n);i++) numbers[PID] += numbers[PID+i*n/log(n)];
  simplesum(n/log(n));
}
```

In contrast, while there exists a simple time-optimal PRAM algorithm for ranking a list of size n with n processors [10], reducing the number of processors to $n/\log n$ requires complex programming means like independent set removal [11].

Another argument why the PTP as defined above might not give best results in reality is a re-interpretation of the PTP. If we see runtime as the inverse of the performance (performance is defined as work per time, and for a given algorithm the work to be done is fixed), then the PTP is also a measure of price per performance. However, this price or cost only includes processors but not the memory cost.

In contrast to the PRAM, the node cost in a GCA includes memory cost as the memory cells are part of the node states. Hence there is no necessity to reduce the processor count below the memory size, i.e. at least the problem size, because this would not simplify the GCA. In turn, this relaxes the PTP-criteria.

Thus, our second result is that in the GCA, optimality criteria are relaxed, and thus non-PTP-optimal PRAM algorithms become optimal in the GCA. Non-PTP-optimal but time-optimal PRAM algorithms typically exist and are much simpler than PTP-optimal PRAM algorithms. As the GCA can avoid these elaborate schemes, simpler algorithms can be applied without any penalty which increases productivity. Also, these simpler algorithms further help the GCA by simplifying the state machines and thus reduce node cost and potentially allow to increase clock frequency.

4 Optimal PRAM emulations via GCAs

While GCAs seem to relate favorably to PRAMs, the realization of reads to arbitrary addresses, possibly concurrent reads, takes time also in GCA implementations. Hence, the question arises whether GCAs only provide new and better insights into optimality of algorithms, or whether they also lead to more efficient implementations than direct PRAM emulations on processor networks, which have been studied for decades.

Emulations of a given P-processor PRAM on a parallel message-passing machine (MPP) typically work in the following manner. The PRAM address space is distributed over the MPP memories in a pseudo-random manner. The size p of the MPP is chosen such that $P = l \cdot p$ and that an l-relation[3] can be handled by the MPP's interconnection network in time $O(l)$. For example, this is true for a coated mesh of size $p \times p$, where p processors occupy only the first and the last row of the mesh, and the routing is done in time $l = O(p)$ [12]. This also holds probabilistically for a wrapped butterfly network of p rows, with processors only present in the first column of the butterfly network. This situation is depicted in Fig. 3(a). The routing of an $l = O(\log p)$ relation takes time $O(\log p)$ in a butterfly network with high probability [13].

Each MPP-processor simulates l PRAM processors, instruction by instruction. Accesses to shared addresses are realized via requests (and responses in case of reads) over the interconnection network to the MPP-memories holding those addresses. Concurrent accesses to some address are handled by combining the requests in the interconnection network or at the memory module holding this address, in order to avoid a hot spot [14,15].

Thus one step of P PRAM processors can be simulated in time $\Theta(l)$ on a p-processor MPP. The Θ is necessary because the interconnect latency gives an upper bound, but the simulation of one step of l PRAM processors on one MPP processors still takes time $\Omega(l)$. As $P = l \cdot p$, this is time-optimal. As the latency in constant-degree networks is $\Omega(\log p)$ or even $\Omega(\sqrt{p})$ if we restrict to constant length wiring between nodes, it follows that $p = O(P/\log P)$. Still, the cost of the MPP is not $\Theta(p)$ but $\Theta(P)$ as at least the state for each PRAM processor must be kept, and also many interconnection networks, such as the butterfly network used in the Fluent Machine and the SB-PRAM, have cost $O(l \cdot p) = O(P)$ [3]. While those considerations have been largely ignored for cases where cost of a processor is much larger than the memory or register cost to store PRAM processor states, it becomes relevant when processing cost is small as in the case of the GCA. A major disadvantage of this emulation is that each emulation step

[3] Each processor-memory pair of the MPP is the source and the origin of at most l packets.

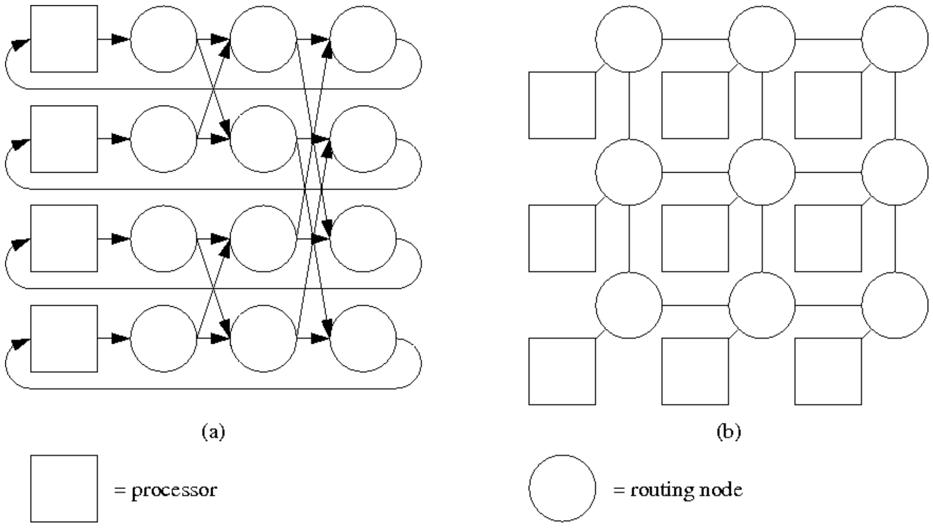

Fig. 3. PRAM emulations: (a) a coated butterfly network, (b) a processor mesh.

takes time $O(l)$, besides the cost being proportional to the number of PRAM processors simulated.

A different approach by Bilardi and Preparata [16] proposes to implement a PRAM on a P-processor network with latency l, e.g. a square mesh, so that each node has routing, processing, and storage functionality. An example is depicted in Fig. 3(b). The cost of the MPP is $O(P)$ as in the previous emulation strategy, and it can host P PRAM processors. While an emulation of shared memory access will take time $O(l)$ as before, simulation of PRAM steps without shared memory access can be done in time $O(1)$, as each MPP processor now only has to host one PRAM processor. The scheme above is formally shown to be optimal in [16]. Intuitively, one can argue that in the general case, all schemes asymptotically have the same cost and the same time span to implement one PRAM step, given that identical networks are used. Yet, in steps that only work locally or in a regular manner (e.g. each PRAM processor may access a memory location hosted by its right neighbor), the PRAM emulations above cannot use this fact to their advantage, even if they are aware of it. Yet, the Bilardi/Preparata scheme only needs a short time for this step.

The Bilardi/Preparata scheme also in a natural manner translates to GCAs, where each cell typically hosts computational and storage functionality, and where implementations complete each GCA step as fast as possible. Furthermore, many algorithms are regular enough so that a compiler can identify steps where concurrent and arbitrary access to the shared memory is not needed; all of these steps can be simulated fast. As thus, GCAs seem an optimal implementation platform for PRAMs.

5 Conclusions

We have presented how Global Cellular Automata provide a bridge between the theory of parallel algorithms, formulated in the PRAM model, and practical realizations in reconfigurable, massively parallel hardware. First, we argued that PRAM algorithms can be efficiently transformed into GCA algorithms. Second, we argued that optimality criteria for GCAs are both closer to reality, and more advantageous than in the PRAM model, thus allowing to use simpler algorithms. Third, we argued that typical implementations of GCAs in reconfigurable hardware are advantageous over classical PRAM emulations with their massive latency hiding, if the full power of concurrent access is not needed too often, which holds for many PRAM algorithms. Thus, we conclude that GCA implementations in reconfigurable hardware provide PRAM emulations tailored to the specifics of the algorithm to be executed.

References

1. JáJá, J.: An Introduction to Parallel Algorithms. Addison Wesley, Reading, MA (1992)
2. Karp, R.M., Ramachandran, V.L.: A survey of parallel algorithms for shared–memory machines. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 869–941
3. Keller, J., Keßler, C.W., Träff, J.L.: Practical PRAM Programming. Wiley & Sons (2001)
4. Valiant, L.G.: General purpose parallel architectures. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 943–971
5. 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
6. Jendrsczok, J., Hoffmann, R., Keller, J.: Implementing hirschberg's PRAM-algorithm for connected components on a global cellular automaton. International Journal of Foundations of Computer Science (IJFCS) **19**(6) (2008) 1299–1316
7. Jendrsczok, J., Hoffmann, R., Lenck, T.: Generated horizontal and vertical data parallel GCA machines for the N-body force calculation. In: Proceedings of the 22nd Conference on the Architecture of Computing Systems (ARCS). (2009) 96–107
8. Jendrsczok, J., Lenck, T., Hoffmann, R., Osterloh, A., Keller, J.: A data parallel GCA machine adapted to the N-body force calculation. submitted (2009)
9. Heenes, W.: Entwurf und Realisierung von massivparallelen Architekturen für globale Architekturen. PhD thesis, Techn. Univ. Darmstadt, Dept. of Comuter Science (2006)
10. Wyllie, J.C.: The Complexity of Parallel Computation. PhD thesis, Department of Computer Science, Cornell University (1979)
11. Cole, R., Vishkin, U.: Faster optimal parallel prefix sums and list ranking. Information and Computation **81**(3) (1989) 334–352
12. Leppänen, V., Penttonen, M.: Work-optimal simulation of PRAM models on meshes. Nordic Journal of Computing **2**(1) (1995) 51–69

13. Leighton, T., Maggs, B., Rao, S.: Universal packet routing algorithms. In: Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science. (1988) 256–269
14. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M.: The NYU ultracomputer — designing an MIMD shared memory parallel computer. IEEE Transactions on Computers **C–32**(2) (1983) 175–189
15. Ranade, A.G.: How to emulate shared memory. Journal of Computer and System Sciences **42**(3) (1991) 307–326
16. Bilardi, G., Preparata, F.P.: Horizons of parallel computation. Journal of Parallel and Distributed Computing **27**(2) (1995) 172–182