Complex Systems **9** (1995) 169–176

A Generalization of Cellular Automata

J Dana Eckart*
Radford University,
Radford, VA, 24142

Abstract. Cellular automata (CA) models have historically included only a bounded amount of state information. Giving cells the ability to send and receive a possibly unbounded number of finite size messages creates three possible CA programming models: bounded state only, unbounded messaging only, and both bounded state and unbounded messaging. It is shown that the latter, although computationally equivalent to the other models, provides a more effective way of describing some CA.

1. Adding messages

A bounded (or classical) cellular automata (BCA) consists of a possibly infinite, n–dimensional lattice of cells (e.g., [8]). Each cell has a state chosen from a prespecified finite alphabet. Cells update their values by using a transition function. The transition function takes as its input the current states of the local cell and some finite collection of nearby cells that lie within some bounded distance, collectively known as a neighborhood. Each cell calculates its next value independent of other cells and updates its value, synchronously with respect to the cells in whose neighborhood it lies, in discrete time steps. BCAs only require a finite and bounded amount of information be kept at each cell. The amount of information is dependent only upon the size of the alphabet. In particular, it is independent of the size, arrangement, and values given as the initial state of the lattice cells.

A generalized cellular automata (GCA) is a BCA with the added ability of cells to send messages to cells within their neighborhood, including themselves. The number of messages, each of finite state, which a cell may send is potentially unbounded. Messages sent during a time step are received at the beginning of the next time step. Furthermore, messages received during one time step can be repropagated and sent to other cells (or even the receiving cell) during the same time step in which they were received. Because the number of messages (each of finite state) received by a cell in a single time

*Electronic mail address: dana@runet.edu.

step is unbounded, it is, in general, impossible to determine the amount of state that a cell will be required to hold. *Cellang* [3] and *StarLogo* [5] are two examples of a growing number of cellular automata (CA) programming systems which provide GCA capabilities.

Another programming system, *Creatures* [6], represents a second subset of GCA, called messaging cellular automata (MCA). Unlike BCA, MCA have no state associated with each cell. Instead, all of the information in the lattice is kept and passed only as messages.

Although BCA, MCA, and GCA are computationally equivalent,[1] this paper argues that for some types of computations GCA provide a more natural way of expressing the computation in addition to allowing for more space and time efficient ways of realizing it. Because BCA are well known for being "embarrassingly parallel," the implications that GCA have for parallelism are also examined. These topics are presented in the context of the *Cellang* [3] programming language and associated compiler.

2. An example

Because of the popularity of BCA and programming systems which support this style of CA programming, GCA will be contrasted only with this subset.[2] By examining a single problem and its solution using both BCA and GCA, the limitations of BCA are demonstrated. Furthermore, it is shown how the corresponding GCA solution easily handles these difficulties.

Christopher Langton's virtual ants provide an excellent problem for a basis of comparison. A virtual ant (vant) is a simple creature that moves about on a grid of black and white cells. If the cell is black the vant turns in one direction and if it is white the vant turns in the other. After turning, the vant changes the color of the cell to the opposing color and moves forward one cell. Figures 1 and 2 show the BCA and GCA versions of the *Cellang* [3] code respectively. Each version was designed and written for a single vant. The implications of supporting multiple vants, and required program changes, are discussed in section 2.2. Although *Cellang* [3] allows the BCA and GCA solutions to resemble one another, the similarity is only superficial.

2.1 One vant at a time

Lines 1–8 of the BCA code (Figure 1) specify that the vant moves about on a 2-dimensional lattice and that each cell in the lattice contains three pieces of information: `color`, `vant`, and `dir`. The `color` indicates the current coloring of the cell, which the vant will change as it moves through the cell. The presence (absence) of a vant is indicated by a value of 1

[1] GCA contains BCA as a proper subset and BCA can simulate a Turing machine [4]. Thus by Church's thesis, GCA and BCA are equivalent. Finally, MCA can emulate BCA by having each cell send itself messages representing the state of the corresponding BCA.

[2] The only known MCA system currently available is the *Creatures* [6] system. Furthermore, there are very few references to this, or any similar system, in the literature.

```
2 dimensions of
    # White = 0 and Black = 1
    color of 0..1
    vant of 0..1

    # North = 0, East = 1, South = 2, West = 3
    dir of 0..3
end

nesw[] for 4 := [0, 1], [1, 0], [0, -1], [-1, 0]
cell.vant := 0

forall i
    if nesw[i].vant & nesw[i].dir = i+%2 then
        dir := nesw[i].dir - 1 when cell.color
            := nesw[i].dir + 1 otherwise

        cell.dir := (dir + 4) % 4
        cell.vant := 1
        cell.color := !cell.color
    end
end
```

Figure 1: BCA code for virtual ants.

```
(1)  2 dimensions of
(2)      # White = 0 and Black = 1
(3)      color of 0..1
(4)
(5)  agent of
(6)      # North = 0, East = 1, South = 2, West = 3
(7)      dir of 0..3
(8)  end
(9)
(10) forall vant : agent
(11)     dir := vant.dir - 1 when cell.color
(12)         := vant.dir + 1 otherwise
(13)
(14)     dir := (dir + 4) % 4
(15)
(16)     agent(dir) -> [0, 1]  when dir = 0
(17)                -> [1, 0]  when dir = 1
(18)                -> [0, -1] when dir = 2
(19)                -> [-1, 0] when dir = 3
(20)
(21)     cell.color := !cell.color
(22) end
```

Figure 2: GCA code for virtual ants.

(0) for `vant`, and `dir` indicates the direction the vant is traveling in, if present.

The BCA solution method is to have each cell look at its neighbors and determine which one, if any, contains a vant headed towards the cell. Line 10 sets the four elements of the `nesw` array, a temporary variable, to the values of the north, east, south, and west neighboring cells respectively. Neighboring cells are indicated by using relative indexing, where $[0, 0]$ would indicate the cell itself. The `forall` loop on line 13 then considers each neighboring cell in turn, and determines whether or not a vant is approaching from that neighbor. By default, the compiler infers that the loop index `i` will take on the values 0..3 since it is used as an index for the `nesw` array. The boolean condition for the `if` statement on line 14 is true only when there is a vant in the neighboring cell and that vant is headed toward the cell. Note here, that `i+%2` is equivalent to `(i+2)%4` where `%` is the modulo operator. Lines 15–18 turn the approaching vant left or right and then assigns the `dir`, `vant`, and `color` fields accordingly. It is important to note the necessity of setting the `vant` field to 0 on line 11. A cell containing a vant must erase its presence since the vant must travel into some other cell.

For a single vant, the BCA solution is not particularly complex, though it may cause confusion for some people to think of having to examine neighboring cells to find vants moving toward a cell. However, even if no vant is present within a cell, that cell still maintains a `dir` field. Whether or not this actually represents wasted space depends upon the implementation, but from a programming standpoint, it is dangerous because it provides an opportunity to (mis)use invalid data.

The GCA solution, shown in Figure 2, on the other hand, uses a message to represent a vant. Each cell still maintains a `color` field, just as the BCA solution did, but the vant itself is represented by an agent (message) which

contains a `dir` field to indicate the vant's direction of travel. The `forall` loop on line 10 examines all of the agents sent to the cell during the previous time step. The loop index `vant` corresponds to each of these agents in turn. Lines 11–14 determine if the vant should turn left or right, in the same way the BCA solution did. Here, however, the loop index `vant` rather than the more awkward `vant[i]` names the vant in question. Lines 16–19 build a new agent (message) with the newly determined direction, and sends the agent (representing the vant) to the appropriate cell, depending upon the direction of travel. Finally, line 21 changes the `color` of the cell once the vant has passed through.

The GCA solution more accurately reflects the components of the problem. Every cell has an associated `color` which is always meaningful. The presence of an agent (message) represents the presence of a vant, and the vant moves (is sent) to the next cell in its newly determined direction of travel. Likewise, the absence of any agents (messages) indicates no vants and the `forall` loop does nothing. Unlike the BCA solution, all of the information at each cell in the GCA solution is always meaningful.

2.2 Multiple vants

Watching a single vant pass through its myriad of gyrations can eventually become tiresome. It is only natural to consider what would happen when each of the previous solutions is extended to allow multiple vants. If each cell contains at most one vant at a time, the previous solutions continue to work correctly. Unfortunately, it is not always possible to determine that vants will never meet. If two or more vants enter the same cell simultaneously it is important to consider what should happen to the cell `color`, how the vants should turn, and whether or not the previous solutions must be modified. When multiple vants arrive at a cell at the same time step, assume that all vants at that cell see the same initial `color`, turn in the same direction (all turning left or all turning right), and change the cell's `color` to the same color after passing through. Finally, assume that the number of vants in the cell lattice is finite, say 10. Thus, no more than 10 vants will ever be within the same cell at the same time.

Figure 3 shows the resulting updated version of the BCA solution. It is structured much as it was before except that instead of a single cell field each for `vant` and `dir`, lines 4 and 7 show arrays of 10 fields for each. Thus a `forall` loop is needed on lines 12–14 to remove all of the vants currently located at the cell, although vants could subsequently move into the cell via the remaining code. An additional `forall` loop, line 18, and loop index `j` are needed to examine each of the array elements in the `vant` and `dir` field arrays of the neighboring cells. The index `k` is used, lines 16, 23, 24, and 26, to place vants in subsequent positions within the cell's field arrays.[3] The

[3]In actuality, the *Cellang* programming language does not allow the use of `k` as an index into an array. It has been used here to simplify what would otherwise become a more complex solution.

```
(1)  2 dimensions of
(2)      # White = 0 and Black = 1
(3)      color of 0..1
(4)      vant[] for 10 of 0..1
(5)
(6)      # North = 0, East = 1, South = 2, West = 3
(7)      dir[] for 10 of 0..3
(8)  end
(9)
(10) nesw[] for 4 := [0, 1], [1, 0], [0, -1], [-1, 0]
(11)
(12) forall j
(13)     cell.vant[j] := 0
(14) end
(15)
(16) k := 0
(17) forall i
(18)     forall j
(19)         if nesw[i].vant[j] & nesw[i].dir[j] = i+%2 then
(20)             dir := nesw[i].dir[j] - 1 when cell.color
(21)                 := nesw[i].dir[j] + 1 otherwise
(22)
(23)             cell.dir[k] := (dir + 4) % 4
(24)             cell.vant[k] := 1
(25)             cell.color := !cell.color
(26)             k := k + 1
(27)         end
(28)     end
(29) end
```

Figure 3: BCA code for multiple virtual ants.

requirement that all of the vants see the same cell `color` and change it to the same color is accomplished since references to cell fields within expressions (i.e., lines 20 and 25) use the original value of the cell field and not the new one which may have been assigned (i.e., line 25) but which does not take effect until the next time step.

While the BCA solution has become more complex, the previous GCA solution works without any modification. Furthermore, the BCA solution shown here fails when the number of vants exceed 10, whereas the GCA solution in no way depends upon a prespecified bound of the number of vants in a cell. Furthermore, if the number of vants in the lattice is small, the BCA solution wastes a significant amount of storage, whereas the GCA solution does not.

If an unbounded number of vants is called for, the GCA solution still remains unchanged. The BCA solution, on the other hand would require extensive redesign. Since a BCA has bounded state, an unbounded number of vants could only pass through it in unbounded time. Thus it would be necessary to trade additional (unbounded) time for lack of (unbounded) space. The solution would become decidedly more complex. GCA provide a more natural, extensible, and space efficient means for expressing solutions to some problems than do BCA.

3. Implications for parallelization

Given that GCA allow more readable algorithms and programs to be written in some cases, if they cannot be efficiently implemented then they are of little practical concern. Techniques for implementing BCA are well known and understood. Thus, the discussion here is limited to the efficient implementation of messages. On a single processor system, one of the most straightforward implementations is to associate a list pointer with each cell. Messages sent to a cell are added to that cell's associated linked list. Although this approach can be somewhat wasteful of memory, it is simple and reasonably quick. This technique is used by the *Cellang* compiler of the *Cellular* system [3]. An alternate technique, used by *Creatures* [6], is to use bucket hashing with cell locations serving as keys. Since all of the states kept by this system reside within agents (the equivalent of messages) the use of the hash table works reasonably well. The *Cellang* language originally only supported BCA programming. It used a dense representation since many CA, such as the hodge-podge machine and lattice-gas models, are dense in nature.[4] Since the compiler already supported a dense representation, wherein the state of every cell in the lattice is explicitly maintained, the simple linked list approach to implementing messaging was preferred.

For a dense representation, the standard technique for both shared and distributed memory multiprocessors, is to divide the cell lattice into segments (usually as vertical strips) with each processor being assigned the task of updating the cells within a single segment. Those cells on processor P_i which lay within the neighborhood of cells on processor P_j are called boundary cells. The width of each segment is chosen to be no smaller than the radius of the largest cell neighborhood. This insures that boundary cells are always on a numerically adjacent processor. Note that it is possible for a boundary cell to be a boundary to more than one processor, but to no more than two since the strips are only at least as wide as the radius, and not the diameter, of the largest cell neighborhood.

For the BCA portion of the computation, the state of boundary cells for P_i must be copied/transmitted to it before the transition function is applied. The copied/transmitted cell states are not changed by the receiving processor, so no changes need be sent back. This is what makes CA so easy to parallelize, since cells can only update their own state. Messages sent to cells belonging to another processor, on the other hand, are buffered locally and then transmitted to the processor whose assigned segment contains the intended recipient cell. The transmission and combining of buffered messages takes place after all of the cells have performed their computation. This technique is used by the *Cellang* compiler for shared memory multiprocessors from Sun Microsystems and Silicon Graphics Incorporated. In tests involving up to four processors, over a wide range of problems, the measured speedup was consistently within 3–7 percent of optimal. Beck and Castellanos have

[4]The game of Life, on the other hand, depending upon the initial state of the lattice cells, can use a very sparse representation.

modified this implementation to demonstrate the efficacy of these techniques on distributed memory machines such as the CM-5 [2].

Although BCAs can make effective use of vector processors, and a technique called microvectors [1] brings the same kinds of benefits to general purpose processors, the linked list message implementation described here cannot utilize microvectors. The inability to use the technique stems from the generally accepted processor design practice of having memory pointers be the same size as memory words. The microvector technique would require that several pointers be able to be packed into a single word for any benefit to be realized. Nor can the linked list message implementation be adapted to either the CAM-6 [8] or CAM-8 [7] pipelined architectures. Although both of these architectures use a dense representation of the cell lattice values, neither provides access to a shared heap that could be used to store a linked list. In fact, it seems unlikely that any suitable implementation for unbounded messaging can be developed for either the CAM-6 or CAM-8, though both remain excellent pipelined architectures well optimized for BCA.

4. Conclusions

GCA provide a useful and effective way for describing some classes of CA computation. A straightforward implementation of messaging using linked lists is both simple and efficient for a wide range of general-purpose hardware, including both shared and distributed memory multiprocessors.

5. Acknowledgements

This work was supported in part by NASA, grant number NAG8-1009, as part of the USRA/JOVE program. Thanks also to Dr. Edward Okie for his helpful comments on an earlier version of this paper.

References

[1] Beck, Micah and Antonio Castellanos, *Vector Processing on Scalar Architectures* (University of Tennessee, Knoxville, TN, Computer Science Department Technical Report, September 1994).

[2] Beck, Micah, personal communication (beck@cs.utk.edu).

[3] Eckart, J Dana, "A *Cellular* Automata Simulation System," available via anonymous ftp from `rucs2.sunlab.cs.runet.edu` in the directory `pub/ca` or from `http://www.cs.runet.edu/~dana/ca/ cellular.html`.

[4] Lindgren, K. and Nordhal, M. G., "Universal Computation in Simple One-dimensional Cellular Automata," *Complex Systems*, 4 (1990), 299–318.

[5] Resnick, Mitchel, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* (MIT Press, Cambridge, MA, 1994).

[6] Stephenson, I., *Creature Processing: An Alternative Cellular Architecture* (Technical Report ASEG92.04, Department of Electronics, University of York).

[7] Toffoli, Tommaso and Norman Margolus, "Programmable Matter: Concepts & Realization," *International Journal of High Speed Computing*, **5**, Number 2 (June 1993), 155–170.

[8] Toffoli, Tommaso and Norman Margolus, *Cellular Automata Machines: A New Environment for Modeling* (MIT Press, Cambridge, MA, 1987).