Programming Cellular Automata for Image Processing[1]

W. Hasselbring

Computer Science / Software Engineering
University of Essen
Germany
willi@informatik.uni-essen.de

February 1992

[1]Presented at the BCS-PPSG Meeting on Cellular Automata, Imperial College, London, UK, February 12, 1992.

Abstract

A theoretical model of a parallel computer is the cellular automaton. A cellular automaton consists of a finite number of Moore-type automata being fixed to the grid-points of a finite-dimensional space. The automata are connected by a homogeneous pattern. The states of the automata are changed synchronously according to a given function. This architecture provides a synchronous communication model.

Cellular automata might be implemented as programmable special-purpose processors controlled by universal computer systems. This configuration is appropriate, because in general only special tasks of complex problems are efficiently parallelizable. Many tasks for image processing like window operators for filtering, smoothing, or edge detection may be presented in a way particularly suitable for cellular automata. In this paper we present a cellular coordination language for image processing and compare it with an asynchronous communication model.

Contents

List of Figures

1 Introduction

Parallel programming and in particular cellular programming requires a new approach to problem solving compared with conventional sequential programming. It is possible to obtain relatively complex behavior of the entire system by relatively simple local instructions. A well known example for this is the *game of life*, introduced by Conway [Gar70]. A human being can emulate this global operations only in a sequential manner (cell by cell). That is remarkable, just because the human brain itself works in a parallel (neuronal) way.

The first cellular algorithms were constructive proofs for important theoretical questions like the computational universality of cellular automata. But these algorithms provide no practical support for cellular programming, because they mostly do not take advantage of the parallelism. Synchronization problems like the Firing Squad Synchronization Problem [Vol79] have greater practical benefit. These are partial tasks to operate cellular automata.

It is well known that not all tasks are efficiently parallelizable. A measure for the degree of parallelizability is the amount of information (data), that is simultaneously moved around. Examples for good parallelizable numerical tasks are matrix operations.

Also many tasks for image processing are tailor-made for cellular automata. A digital grey-scale image is represented in a computer as a two-dimensional rectangular array of discrete grey-scale values (pixels). Typical methods are window operators for filtering, smoothing, or edge detection. Parallelization is attractive here since the same operation is applied to every image point, at which the resulting value depends on the grey-scale values of the respective neighboring points (in the window). However, also in image processing only partial tasks are efficiently parallelizable. Thus a cellular automaton for it will not be operated as a stand-alone computer, but as a special-purpose processor in an universal computer system [Leg76]. This mode of operation is quite appropriate, because in this way the parallelized tasks may be operated on the cellular automaton and all the services offered by the host system may be used.

Programming of cellular automata in theoretical computer science is done essentially by constructing transition tables. This method is oriented towards the internal representation of cellular algorithms by tables and not towards the requirements of their applications. An additional difficulty for handling cellular programs derives from this distance to the applications. It is hard to see which function such a table computes. We will now present a cellular coordination language in which the transition functions are specified similar to statements in procedural programming languages.

2 The coordination language CELIP

A *parallel programming language* consists of a coordination language and a sequential computation language [CG92]. This section presents the coordination language CELIP [Has90] — a CEllular Language for Image Processing — as an extension to the computation language C [KR83]. A language for cellular automata should be made available as an extension to a sequential computation language, since such an automaton attached to a computer system is to be programmed and controlled by the host.

In the configuration described above there exists the problem of transferring the input (digital image) between the cellular automaton and the host. To obtain a well-balanced relation

```
cell
{
     byte reg1,
          reg2;
     byte regarr [9];
}
```

Figure 1: A definition of the automata cell.

between the time for loading and for processing, in most existing systems the input is pumped through a pipeline (column for column). Hence for window operators the grey-scale values of the neighbor points have to be held. To avoid this difficulty, in CELIP each operation is applied to all points simultaneously. CELIP makes no operations for pipelining available.

In most existing systems for cellular automata it is necessary to define the cellular net (retina). In CELIP the cellular net is fixed in size and shape (512×512 cells). Each image point corresponds exactly to one cell.

The proposed coordination language allows defining the cell memory, the neighborhood connections, and the transition functions for the cellular automaton. In principle it is possible to define an arbitrary neighborhood.

These assumptions are not realistic with a view towards a possible hardware realization, but they are well suited to prototyping cellular algorithms for image processing.

A comparable approach to parallel image processing may be found in Paragon [Ree91] which provides data-parallel operations on so-called *Parrays* ("parallel arrays") to avoid overspecification of loops. Data-parallel operations in Paragon are applied as function calls, whereas CELIP overloads predefined operators of the host language C.

2.1 The definition of the automata cell

The user defines the local memory which is divided into several 8-bit-registers for the cells (identical in structure for every image point). The declarations take place in a `cell`-block to set them apart from the declarations for memory on the host. An example is given in Fig. 1.

The reserved word `byte` indicates an 8-bit-register (one byte) which can hold the integer values from 0 to 255 ($2^8 - 1$). It is planned to add further register types. Especially 16-bit-register to increase the range for intermediate results. Such supplements would not change the basic concepts of CELIP.

2.2 Intensities

Sometimes it is necessary to operate in several registers in all cells with certain intensities (grey-scale values). To meet this requirement the data type `intensity` is introduced. `intensity`-variables can hold the same values as a cell register, hence the integer values from 0 to 255. The memory for variables of this data type is located in the host's memory, not in the cellular automaton.

A	**B**	**C**
D	**E**	**F**
G	**H**	**I**

Figure 2: The '8-Neighborhood' in image processing.

$$\begin{array}{ccc} (-1, & 1) & (0, & 1) & (1, & 1) \\ (-1, & 0) & (0, & 0) & (1, & 0) \\ (-1, & -1) & (0, & -1) & (1, & -1) \end{array}$$

Figure 3: Neighborhood index for the '8-Neighborhood'.

2.3 The definition of the neighborhood

In theory the neighborhood connections are specified with a neighborhood index [Vol79]. In a two-dimensional space the neighborhood index consists of a finite number of pairs of natural numbers. All pairs are different. The neighborhood index is a set of pairs. Such sets are also called *maps* in set theory. The cellular automaton is an euclidean space, where each cell has an unique pair of x- and y-coordinates. The neighborhood index *maps* x-coordinates to y-coordinates. The x- and y-coordinates of the neighbor of a cell are obtained by adding the corresponding elements of the neighborhood index (offsets) to the coordinates of this cell.

Fig. 2 displays an example for the so-called 'Moore-Neighborhood'. Especially in image processing it is also called '8-Neighborhood'.

In image processing it is usual to represent the neighborhood connections in this way. In this example the name E is implicitly assumed to mark the central cell. The pair $(0, 0)$ corresponds in the neighborhood index to the central cell. The neighborhood index in Fig. 3 corresponds with the above definition. Strictly spoken the name '9-Neighborhood' would be more suitable, because each cell is a neighbor of itself.

In CELIP the definition of the neighborhood takes place with (modest) graphical means. The implemented CELIP-compiler converts this representation into an internal representation, that is in accordance with the neighborhood index. An example specification for the '8-Neighborhood' is given in Fig. 4.

The introducing reserved word `neighborhood` could be replaced by the synonymous word `neighbourhood`. The declaration of the neighbors takes place inside the braces. The neighborhood index is assigned according to the positions of the names with respect to white-space characters. The offsets are related to the position of the central cell, which is marked with the symbol `$`. The first neighbor in the row with the central cell obtains the lowest x-offset in

```
neighborhood
{
     A    B     C
     D   $E     F
     G    H     I
}
```

Figure 4: The '8-Neighborhood' in CELIP.

```
neighborhood
{
   N [
         —    2     —
         3   $0     1
         —    4     —
     ]
}
```

Figure 5: The '4-Neighborhood'.

the neighborhood index. The line feed decreases the y-offset by 1. It is obligatory to declare exactly one such central cell. The optional name behind `$` has to follow directly without preceding white-space characters.

It is possible to define an indexed neighborhood array. This is done by declaring the array indices explicitly to obtain the relation to their respective positions. These array indices should not be confused with the neighborhood indices, which have geometrical means.

As an example serves the so-called 'von-Neumann Neighborhood' in Fig. 5. Especially in image processing it is also called '4-Neighborhood'.

The '—'-characters inside the neighborhood definition increases the x-offset by 1. It can be seen as a neighbor without a name or an index. Both left '—'-characters are necessary in Fig. 5 to obtain the right x-offsets for `N[2]` $(0, 1)$ and `N[4]` $(0, -1)$. The right ones were inserted for the optical symmetry.

An indexed neighborhood array is in particular useful for processing the neighbors in a loop.

The access to registers of the neighbors is syntactical similar to the selection of structure components in C. For example in the expression '`A.reg1`' register `reg1` of neighbor `A` is selected. The selection '`E.reg1`' is equivalent to '`reg1`', because the own registers (in the central cell) are the default.

2.4 Cellular statements

The transition functions are specified by assignments to a register of the central cell. These cellular statements are executed on the cellular automaton. The syntax corresponds with the syntax of plain C-assignments. First an example (the above definitions are valid):

not, min, max
*
//
**
−
+
<, <=, >, >=
==, !=
and
exor
or
=

Figure 6: The cellular operators sorted corresponding to their priorities.
The logical operators are represented by the reserved words `and`, `or`, `exor` and the unary operator `not` resp. to distinguish them from the corresponding C-operators. In C there exist different operators for logical and bit-wise logical operations, but in CELIP only bit-wise logical cellular operators are available. The aggregate operators `max` and `min` are used as function calls with a variable number of operands.

```
intensity threshold;
threshold = 100;
reg1 = reg2 > threshold;
```

The first statement is no cellular statement, because it is executed by the host. Cellular statements like the second one are executed by the cellular automaton. The result of the operation on the right side of the assignment then becomes the subsequent state for the register specified on the left hand side.

The binary operator > performs a binarization: The registers `reg1` in all those cells, in which the relation '`reg2 > threshold`' is valid, obtain the integer value 255; elsewhere they will hold the value 0. This operation is a so-called *threshold binarization*. It is a *point operation* because the neighborhood connections are not used. The result of such a binarization is a binary image in the destination register.

There are binary operators for other comparisons and arithmetic and logical operations with the common priorities available. Fig. 6 presents the cellular operators sorted corresponding to their priorities. The results of the arithmetic operations are restricted to the values from 0 to 255 (to be able to hold them in 8-bit-registers).

The following operation is an example for a *window binarization* to present a more complex example:

```
reg1 = 100 < reg2 and reg2 < 200;
```

For binary images *erosion* is used for noise reduction, whereby the object areas are reduced by their marginal points. An object area in a binary image is represented by connected object points. Object points are cells with the integer value 255 in the register for the binary image. An object point only remains an object point, if all his neighbors are object points themselves. This may be expressed as follows:

```
where (reg2 > 100)
{
  there   reg1 = 255;
  else    reg2 = 0;
}
```

Figure 7: Threshold binarization with the `where`-statement.

```
reg1 = A.reg1 and B.reg1 and C.reg1 and D.reg1 and
       E.reg1 and F.reg1 and G.reg1 and H.reg1 and I.reg1;
```

Small interferences, but also thin lines, are eliminated. The reverse operation *dilatation* can be realized with the `or`-operator instead of `and`.

The so-called 'Roberts-Gradient' is an example for an operator for edge enhancement [Wah84]. It is realized in CELIP as follows:

```
reg1 = max (|B.reg1 — A.reg1|, |D.reg1 — A.reg1|);
```

The symbol | can be used as parentheses. For addition and subtraction the resulting values are the absolute values of the result given by the enclosed expression.

A more complex statement is the conditional `where`-statement. Fig. 7 shows another realization of the above-mentioned threshold binarization.

In every `there`- and `else`-branch we allow only one assignment or an interlocked `where`-statement. This restriction is necessary to synchronize the statements. In general in some cells the given condition is valid (the `there`-statements are executed) and in others it is not valid (the `else`-statements are executed). Thus the `there`- and `else`-statements can influence each other via the neighborhood connections. Consequently the necessary synchronization is done in a syntactical way. The `else`-branches are optional.

3 Example: median filtering

Median filtering is a smoothing operation that works with various neighborhood connections (windows) [Wah84].

Our algorithm in Fig. 8 works as follows: the values in the neighborhood of each cell are ordered according to their grey-scale values. The middle value is the resulting median. This program uses a *slanted*, indexed neighborhood connection, thus retaining only slanted objects in the image.

4 Implementation

A CELIP-compiler has been implemented on a μ-VAX running VMS, which serves as a host. The cellular statements are executed on a peripheral image-processing device VTE-PICTURECOM, which emulates the cellular automaton. This special-purpose computer is

```
#define N 13

main ()
{
    int i, j;     /* suitable index counter */
    cell
    {
        byte reg,        /* for the grey-scale image */
             temp;       /* for exchange */
        byte regarr [N]; /* for sorting */
    }
    neighborhood
    {
        SLANTED [
                    -  -  -  0  1
                    -  -  2  3  4
                    -  5 $6  7
                    8  9 10
                    11 12
                ]
    }
    load ("grey-scale.dat", &reg); /* load the image */
    for (i = 0; i < N; i ++)
        regarr[i] = SLANTED[i].reg;
    /* The Bubble-sort-Algorithm:  */
    for (i = 0; i < (N/2)+1; i ++) /* sort up to the middle */
    {
        for (j = N-1; j > i; j --)
        {
            where (regarr[j] < regarr[j-1])
            {
               /* temp is temporary used to exchange the
                * contents (to bubble):
                */
               there  temp = regarr[j];
               there  regarr[j] = regarr[j-1];
               there  regarr[j-1] = temp;
            }
        }
    }
    reg = regarr [N/2]; /* The resulting median */
}
```

Figure 8: Median filtering.
At first the values of the neighbors are stored in the local array `regarr` to sort them there. The obtained values are sorted by a bubble-sort algorithm. It is sufficient to sort up to the middle.

capable of storing 24 grey-scale images (8-bit-register) with 512×512 image points. It can operate with two registers at a time via arithmetic-logical-units (ALUs) and look-up-tables (LUTs). Such a read-write cycle is executed in video-time (40 ms). Appendix A displays the coarse structure of the VTE-PICTURECOM. Shifting the memory contents at the switches is supported by the hardware thus it was straightforward and efficient to implement the access to the neighbors this way.

The restriction to 8-bit-registers for the partition of cell memory is a concession to the capabilities of the VTE-PICTURECOM. 16-bit-registers are desirable for increasing the range of intermediate results. The corresponding new data types for the implementation on different hardware would not change the basic concepts of CELIP.

For the implementation on different hardware the principle of operating a cellular automaton via a host system should be maintained. The main problem is to implement the variable neighborhood connections with justifiable expense. In [Vol79, section 2.1] a method is presented, which allows to transform an algorithm that is designed for an arbitrary neighborhood into an algorithm that uses only the von-Neumann neighborhood. This method could serve as a basis for transforming a prototype written in CELIP into a production program to be executed on a parallel hardware system, in which the processors are connected in the von-Neumann style. See e.g. [Par90] for a full account to program transformation techniques.

5 Comparison with asynchronous communication

Opportunities for automatic detection of parallelism in existing programs are limited and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Hence, it is not the question if we should provide explicit parallelism to the programmer. The question is "How to provide explicit parallelism?".

According to the classification of Flynn [Fly66], cellular automata belong to the *single instruction stream – multiple data stream* (SIMD) computers as opposed to the *multiple instruction stream – multiple data stream* (MIMD) computers.

The coordination language CELIP provides a synchronous communication model for a SIMD architecture. We will now compare this model with an asynchronous communication model for MIMD architectures, viz. the Linda model. The Linda model has been chosen as a representative for asynchronous communication, because the author himself works in this area [Has91].

Linda is a coordination language concept for explicitly parallel programming in an architecture independent way, which has been developed by David Gelernter at Yale University [Gel85]. Communication in Linda is based on the concept of tuple space, i.e. a virtual common data space. Linda defines six operators, which may be added to a sequential computation language. These operators enable sequential processes, specified in the underlying computation language, to access the tuple space. Process communication and synchronization in Linda is called generative communication, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly. Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. Each component of a tuple or template is either an *actual*, i.e. holding a

value of a given type, or a *formal*, i.e. a placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by a matching procedure, where a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields. [CG90] provides a full account to parallel programming in Linda.

In cellular automata it is possible to obtain a relatively complex behavior of the entire system by relatively simple local instructions. A human being can emulate this global operations only in a sequential manner (cell by cell). The programmer often has to focus on more than one process at a time. It is necessary to think in simultaneities.

Programming in Linda provides a spatially and temporally unordered bag of processes. Each task in the computation can be programmed (more-or-less) independently of any other task. This enables the programmer to focus on one process at a time thus making parallel programming conceptually the same order of problem-solving complexity as conventional, sequential programming. It is no more necessary to think in simultaneities.

Process communication and synchronization in Linda is reduced to concurrent access to a large data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a very simple way. This scheme offers all advantages of a shared memory architecture, such as anonymous communication and easy load balancing. It adds a very flexible associative addressing mechanism and a natural synchronization paradigm and at the same time avoids the well-known access bottleneck for shared memory systems as far as possible.

The uncoupled and anonymous inter-process communication in Linda is in general not directly supported by the target machines. This implies a runtime overhead for Linda programs compared to explicit low-level parallel code such as message passing, but supports portability across different machine architectures. Implementations of Linda have been performed on a wide variety of parallel architectures: shared-memory multi-processors as well as on distributed memory architectures. Linda can be compared to explicit low-level parallel code such as message passing, in much the same way as high-level programming languages can be compared to assembly code.

Programs for cellular automata are designed for cellular-like hardware architectures on which they will be executed very efficiently, but they are not portable across different machine architectures.

Another distinguishing property for the applicability of both communication models is the granularity of the application to program. The granularity of a parallel program is the ratio between communication and computation. Cellular automata are well suited for fine-grained applications (communication is inexpensive), whereas Linda seems to be well suited for coarse-grained applications (communication may be expensive).

6 Conclusions

A cellular coordination language has been presented, which allows prototyping cellular algorithms for image processing by operating a cellular automaton attached to an universal computer system as a peripheral device. The cellular extensions are concerned with the definition of the cell memory, the neighborhood connections, and the cellular statements (transition functions). This concept is custom tailored for image processing. The language in

the present form is intended as a proposal for the basic paradigm of a programming language for cellular image processing.

We sketched implementation issues and compared the proposed synchronous model with asynchronous communication in Linda. Our conclusions are:

- Cellular programs may be executed efficiently on cellular-like hardware architectures.
- Cellular automata are well suited for fine-grained applications.
- Linda programs are portable and in general not directly supported bymax hardware.
- Linda programs are well suited for coarse-grained applications.
- Programming in Linda is easier than programming cellular automata.

References

[CG90] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[CG92] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 12:1901–1909, 1966.

[Gar70] M. Gardner. The fantastic combination of John Conway's new solitaire game "life". *Scientific American*, (223):120–123, 1970.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Has90] W. Hasselbring. CELIP: A cellular language for image processing. *Parallel Computing*, 14(5):99–109, 1990.

[Has91] W. Hasselbring. On Integrating Generative Communication into the Prototyping Language ProSet. Informatik-Bericht 05-91, University of Essen, December 1991.

[KR83] B.W. Kernighan and D.M. Ritchie. *Programmieren in C.* Carl Hanser Verlag, 1983.

[Leg76] T. Legendi. Cellprocessors in computer architecture. *Computational Linguististics and Computer Languages*, 11:147–167, 1976.

[Par90] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Ree91] A.P. Reeves. Parallel programming for computer vision. *IEEE Software*, 8(7):51–59, 1991.

[Vol79] R. Vollmar. *Algorithmen in Zellularautomaten.* Teubner Verlag, 1979.

[Wah84] F.M. Wahl. *Digitale Bildsignalverarbeitung.* Springer-Verlag, 1984.

A The coarse structure of the VTE-PICTURECOM