CELLAR: A High Level Cellular Programming Language with Regions

Gianluigi Folino and Giandomenico Spezzano

ISI-CNR
c/o DEIS, Università della Calabria
87036 Rende (CS), Italy
{folino,spezzano}@si.deis.unical.it

Abstract

This paper describes CELLAR, a language for cellular programming which extends the cellular automata model through the concept of regions. Regions are spatio-temporal objects that define zones of the automaton (set of cells), containing interesting and meaningful data patterns or trends that can be defined as events. Each cell of the automaton can monitor regions for a given period and observe their evolution by global functions (max, min, sum etc.). Furthermore, each cell can have an associated attribute called its perception rating, that indicates how far that cell can 'see'. On the basis of this value and the cell's position in the cellular space, we can define the regions that are visible to the cell. Using these constructs, a cell can define significant events to extract data of interest in one or more regions and perform actions when an event is detected. In the paper, we show that regions simplify programming and allow the building of more complex models. After describing the main constructs of CELLAR, the paper illustrates the region-based programming model by describing the design of a parallel model of animal migration. Performance results of the model implemented on a Meiko CS-2 are also given.

1. Introduction

Cellular computing [1] is an emergent parallel programming paradigm based on the computational model cellular automata (CA) that is effective both for scientific and engineering computations and as a framework to enable fine-grained parallel computations.

A CA is composed of an array of interacting cells, either one-dimensional or multidimensional. Each cell can have a finite number of states. The states of all the cells are updated synchronously according to a local rule, called a *transition function*, according to which, the state of a cell at a given time depends only on its own state at the previous time step and the states of its "nearby" neighbors (however defined) at that previous step. Thus, the state of the entire automaton advances in discrete time steps. The global behavior of the system is determined by the evolution of the states of all the cells as a result of multiple interactions.

CA model has been applied to a wide range of practical applications such as freeway traffic flow, landslides, lava flow, particle dynamics, forest fire, and soil bioremediation. However, to support a larger number of real applications several extensions and modifications to the basic model have been proposed. A detailed description of these extensions is presented in [2]. The main changes concern: the possibility to have a more complex representation of the state of a cell instead of a few bits, temporal and spatial inhomogeneity both in the transition function and in the neighborhood, asynchrony so that each cell can in each step non-deterministically choose between changing its state according to the transition function or keeping its current state, complex time-dependent neighborhood (i.e. block rule), probabilistic and hierarchical transition functions. Moreover, many phenomena, especially those studied in biology, ecology and sociology, require information from other cells which do not belong to the cell's neighborhood.

Research activities on implementation of cellular automata models was mainly focused on algorithms and applications. For many years, the programming issues were not considered one of the major issues to be faced and solved. However, in recent years a number of cellular programming language such as CELLANG [3], CDL[4], CARP [5], and CEPROL [6] have been defined.

In our previous works [7,8,9] we defined and implemented a parallel cellular programming environment

and a special language, named CARPET, to support both practical development of computational science applications and parallel algorithms in artificial intelligence. CARPET is a high-level language based on C with additional constructs to describe the rule of the state transition function of a single cell of a cellular automaton. The main features of CARPET are: the possibility to describe the state of a cell as a record of typed substates, the simple definition of complex neighborhoods (e.g., hexagonal, Margolus) that can be also time dependent and the specification of non-deterministic, time-dependent and non-uniform transition functions. In CARPET, the approach used in order to have a longer range of interaction among the cells consists in considering the cell's neighborhood within a larger radius. This method is very expensive because it significantly increases the overhead due to the communications.

This paper describes a new *region-based* approach for extending the interactions among the cells. Regions are spatio-temporal objects that define zones of the automaton (set of cells) containing interesting and meaningful data patterns or trends that can be defined as events. Each cell can monitor regions for a given period and observe their evolution by global functions (max, min, sum etc.). Moreover, each cell can define significant events that involve one or more regions and take actions when an event is detected. Occurrences of these events are automatically detected by runtime support system.

To exploit the concept of regions we defined and implemented an extension of the CARPET language, called CELLAR, with new constructs to handle region-based programming. The ZPL language also uses the concept of regions for expressing array computation [10].

The paper is organized as follows. Section 2 introduces the cellular programming model and presents the extension of the model with regions. Section 3 presents an overview of the CARPET language whereas section 4 illustrates the CELLAR language constructs for programming cellular algorithms using the concept of regions. Section 5 describes the programming environment and the parallel run-time system of CELLAR. Finally, section 6 illustrates the region-based programming model by describing the design of a parallel model of animal migration, and presents performance results.

2. Cellular programming model with regions

Cellular algorithms are performed by concurrent programs composed of numerous, fine-grained, iterative processes locally interacting according to usually simple rules (transition function). Local interaction between processes results in complex patterns of evolution of the state of the entire system. Termination of a cellular algorithm may be triggered by reaching a maximum number of generations or by finding an acceptable solution.

A problem can be described by a cellular algorithm defining the local transition function and the initial configuration of the state of each cell. Moreover, the global characteristics of the model, such as the border conditions or the size of the cellular array, must be defined. If the CA is homogeneous, the cellular algorithm is constituted by a collection of identical transition functions applied to all the cells of the automaton; otherwise, different transition functions must be defined for the non-uniform cells.

In traditional CA a cell can interact only with the cells defined within its neighborhood. CELLAR extends the range of interaction among the cells introducing the concept of region. Regions are spatio-temporal objects, statically defined, which allow a cell to know, by global functions (max, min, sum, avg, and, or, etc.), the behavior of a set of cells put within a defined area. At each iteration cells can update their own state not only with the state of the neighbor cells but also considering the global values obtained by global functions defined on substates of the cells of a region. A cell can also define events which involve variables, defined as substates of the cells of a region, and take actions when an event is detected. An event expression is composed of logical connectives (and, or, not) that combine the global functions applied to the event variables, where for each function it is checked if it is less-than, greater-than or equal to a constant value.

Each cell can have an associated attribute called *perception rating,* that indicates how far that cell can 'see'. On the basis of this value and the cell's position in the cellular space, we can define the regions that are visible to the cell. Figure 1 shows the region-based programming model used in CELLAR.

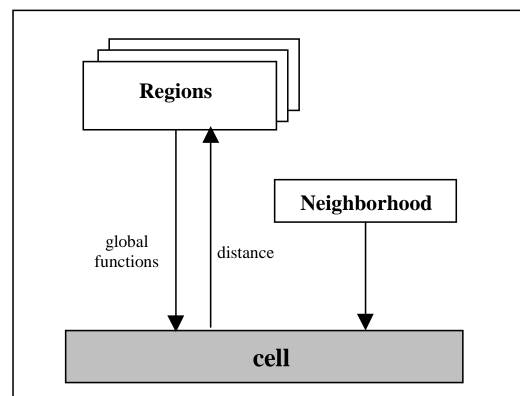

Figure 1. The CELLAR programming model with regions.

3. Overview of CARPET

The CARPET language allows to define the transition function of a cell of the automaton. It handles three kinds of objects: *state*, *neighborhood* and *parameters* that are defined inside the *cadef* declaration section.

State objects are passive entities composed of a set of typed substates. They are structured as a record in which the C basic types: *char*, *shorts*, *integers*, *floats*, *doubles* and mono-dimensional *arrays* of these types can be used to store the physical quantities of a model or the data structures of a parallel algorithm. The predefined variable *cell* refers to the current cell in the n-dimensional space under consideration. A substate can be referred appending to the reserved word *cell* the substate's name by the underscore symbol '_' (i.e, *cell_substate*). Cell substates are updated at each iteration only by the *update* function, in order to guarantee the semantics of cell updating in cellular automata. After an *update* statement, the value of the substate, in the current iteration, is unchanged. The new value takes effect at the beginning of the next iteration. The neighborhood of a cell is defined as the maximum number of cells that a cell can access in reading. For example, in a 2-dimensional automaton defining the radius equal to 1 the number of the neighbors can be up to 8. To define transition functions or neighborhood time dependent, the predefined variable *step* is used. Step is automatically updated by the system. Initially the value of step is 0 and it is incremented by 1 at each iteration. To allow a user to define spatially inhomogeneous CA, CARPET defines the GetX, GetY and GetZ operations that return the value of X,Y and Z coordinates of a cell in the automaton.

Parameter objects describe some global features of the system. CARPET allows to define global parameters and to initialize them to specific values. The value of a global parameter is the same in each cell of the automaton. Parameters can be modified by user interface (UI), described in section 4, during the automaton execution.

```
cadef
{
  dimension 2;  /* bidimensional lattice */
  radius 1;
  state (short value);
  neighbor cross[4]([0,-1]North,[-1,0]West,
                    [0,1]South,[1,0]East);
}
 int i; short N = 0;
{
  for (i=0; i<4; i++)
    N = cross_value[i] + N;
  if (N % 2 == 1)
    update(cell_value, 0);
}
```

Figure 2. The parity rule game.

The example in figure 2 shows how the CARPET constructs can be used to implement the parity rule program. In this example the cells can have '0' or '1' values only. Let us call N the number of '1' cells among the four nearest neighbors of a given cell. The transition rule is the following: given a cell, if N is odd, the new value of the cell will be 0; if N is even the cell's value does not change.

4. The CELLAR language

CELLAR is an extension of the CARPET language. It inherits all the CARPET constructs and extends the cellular programming introducing new constructs to handle the region-based model of programming.

In CELLAR, region objects are statically defined, inside the cadef section, by the `region` declaration and identified by the name of a vector with dimension equal to the number of regions defined in the automaton. Inside a region declaration, the areas of interest are enclosed in round brackets and separated by commas.

A *d*-dimensional region is defined by a sequence of indices which represent the geometric coordinates, the time period (starting time and ending time) in which the region is defined, and the interval of monitoring.

The following example defines three separated regions:

region zone[3] (*area1*(10,20,10,30, 0,0,10,300,5),
area2(50,60,10,20,0,0,10,300,5),
area3(10,20,40,50,1,1,10,300,5))

The region *area1* is a rectangular area defined by the x-y-z coordinates, and it is defined only during the time period from 10 to 300 with a monitoring interval equal to 5. The global behavior of the substates of the cells of a region can be observed by global functions that return a numerical value. CELLAR implements the functions *MaxRegion, MinRegion, SumRegion, AndRegion, OrRegion, AvgRegion*, which respectively allow to calculate the maximum, the minimum, the sum, the logical and, the logical or, and the average value of substates of the cells belonging to a region. Other functions can be added in the future. In the following example, the MaxRegion function is applied to the temperature substate

max = **MaxRegion** (area1_temperature, &success)

In the example given above the function MaxRegion assigns to *max* the maximum of the values that cells, belonging to the *area1* region, take for the temperature substate. The *success* variable takes a true value only if the function is performed during the time period specified. In the previous example, if the current iteration is 320, the

value returned by the success variable is false and max equals to 0.

To know if a cell belongs to a spatial region we have defined the *InRegion* function. For example, the function:

val = **InRegion**(area1)

returns a true value if the cell belongs to the *area1* region; otherwise a false value is returned.

To check if a certain region can be monitored in the current iteration we have defined the *InTempRegion* function. In the following example

InTemp = **InTempRegion**(area1)

the *InTemp* variable is true if the current iteration is within the temporal window and the monitoring step is that defined for the *area1* region.

Regions that are visible to a cell are calculated by the *Distance* function. The *Distance* function returns the distance between the cell and the region considered. The Distance function is calculated by taking the integer part of the Euclidean distance between the coordinates of the cell and the coordinates of the cell that represents the center of a region. The center of a region is calculated by taking the average value of the three spatial coordinates. Distance function returns zero if the cell is within the region; otherwise it returns the distance value. The following example:

if (**Distance**(area1) < 20)

allows to check if the *area1* region is visible for the current cell. For example, the cell can define if the value returned from the Distance function is lower than 20 then the area1 region is visible where 20 represents its perception rating.

Significant events can be defined on one or more regions by defining expressions that contain the above defined functions. The basic event-action control structure of CELLAR is

if <*event-expr*> **then** < *action*>

An *event-expr* is an expression that combines the global functions defined on a region by basic numerical expressions as well as relational and logical operators. Actions consist of changing the value of some substates of the current cell or event definitions.

The example in figure 3 describes a simple simulation of the propagation of a forest fire and shows how the main constructs of the CELLAR language can be used. In this example each cell represents a portion of the land. Cells in the lattice can have values included between '0' and '2'. The ground is represented by '0' value, the tree is represented by '1' value and the fire is represented by '2' value. Fire spreads from a cell which is on fire to a von Neumann neighbour that has trees, but not on fire. The *area1* region represents the zone that must be controlled. A user-defined trigger on the *area1* region allows a user to monitor the presence of fire in the zone. The region is isolated when the fire is present. The fire is detected calculating the maximum of the *area1* region for the *land* substate because the fire is represented by '2' value. If the event is verified, then all the cells located at a distance from 30 to 20 from the region with the fire, change their value to *ground* in order to prevent the propagation of the fire. The *max* and *min* parameters can be changed to simulate different areas that isolate the region on fire.

```
#define ground 0
#define tree 1
#define fire 2
cadef
{
   dimension 2;
   radius 1;
   state (short land);
   neighbor cross[4]([0,-1]North,[-1,0]West,
                     [0,1]South,[1,0]East);
   parameter (min 20.0, max 30.0);
   region (area1(40,60,30,40,1,1,1,200,1));
}
int succ, dist;
{
 dist = Distance(area1);
  if((MaxRegion(area1_land,&succ)==fire) &&
       dist < max )&&(dist > min))
      update(cell_land, ground);
     else
      if((cell_land==tree)&&(North_land==fire) ||
         (East_land == fire) ||(West_land == fire)
           || (South_land == fire ))

         update(cell_land, fire);
           else
            if (cell_land== fire)
              update(cell_land, ground);
}
```

Figure 3. The forest fire simulation.

5. The parallel environment

The interactive parallel environment that allows the development and running of CELLAR programs on parallel architectures is called CARAVEL. The main goal of CARAVEL is to integrate computation, visualization and control into one environment to allow interactive steering of scientific applications [11]. CARAVEL consists of

- a graphical user interface (GUI) for editing, compiling, configuring, executing and steering the computation;

- a run-time support for the parallel execution of CELLAR programs;
- a load balancing strategy to evenly distribute the computation among processors of the parallel machine.

CARAVEL's GUI provides a *development window* for editing, compiling and configuring a cellular program and a *simulation window* for the running and the exploratory steering of the simulation.

To configure an application we use he *configure* menu of the development window defines the dimensions of the CA engine, the number of processes in which the CA is divided, the number of processors on which to allocate the processes and the numbers of folds which define the regions for the load balancing strategy described in the section 5.1.

When the simulation window is started the user can execute the simulation for a number of steps defined (by clicking the **go** button) or for an infinite number of steps (by clicking the **loop** button). During the execution of the simulation the results are visualised on multiple displays and the user can steer the computation stopping the execution of the application (by clicking the **pause** button), change parameters and/or the value of some substates and restart the execution. Figure 4 shows how the interface is used to change the value of a parameter. After entering the new parameter values, the user clicks on the **ok** button to send the steering command to the application which uses the parameter value for computations in the next time step.

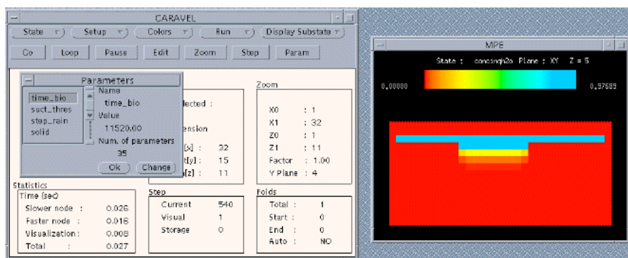

Figure 4. Simulation window of CARAVEL's GUI.

Moreover, the current version of CARAVEL allows

- to choose the colours that can be assigned to the cell substates to support the graphical visualization of their values;
- to change the visualization step to reduce the visualization time;
- to save at regular time the state of the CA in a file.

5.1. The parallel runtime system

The CELLAR run-time system maps CA programs on a parallel computer hiding the architecture issues to a user. Parallel computers are the best practical support for the effective implementation of high-performance CA [12]. The CELLAR run-time support is implemented as a SPMD (Single Program Multiple Data) program. The current implementation is based on the C language plus the standard MPI library and can be executed on different machines such as the Meiko CS-2, CRAY T3E and LINUX cluster of workstations. The concurrent program which implements the architecture of the system is composed by a set of *macrocell* processes, a *controller* process and a *GUI* process. Each macrocell process, which contains a strip of cells of the CA, runs on a single processing element of the parallel and executes the updating of the state of cells belonging to its partition. The synchronization of the automaton and the execution of the commands, provided by a user through the GUI interface, are carried out by the *controller* process. MPI primitives handle all the communications among the processes using MPI *communicators.* CARAVEL uses the capability of the MPE graphics library of MPI, that allows a set of processes to share an X display, in order to visualise the results of a simulation on line.

At each iteration, the region operations are performed in two steps. In the first step, the operations are performed on local data. Each processor performs that portion of the computation applicable to the index values of the data of a region stored in its memory. In the next step these data are combined using the MPI *AllReduce* function that applies the operations to local data and returns the final result to all processors. For example, in a SumRegion operation before the sum of the values stored on each processor is computed locally. Then these partial sums are combined and distributed to all processors. In this way, at the beginning of each iteration, the result is available for each cell of the automaton.

To improve the performance of applications that have a diffusive behavior such as CFD, the CELLAR run-time system implements the same load balancing strategy for mapping lattice partitions on the processing elements used in CARPET [13].

This load balancing strategy is a trade-off between the static and dynamic approach. In fact, the cells partitioning is static, whereas the amount of cells mapped on each partition is dynamic. In CARAVEL, the grid of cells is first divided into n vertical folds; each fold is then partitioned into N strips, where N is the number of processors of the multicomputer. The *i-th* strip of each fold is assigned to the generic processor P*i*. To avoid useless computation the user may change, at run time, the set of folds on which the state transition function must be applied. Each macrocell process will compute only the strips of the specified folds. The set of active fold will be augmented or restricted just before some cells become active or passive. The choice of the active folds can be automatic including some tests.

6. Example: a parallel model of animal migration

We illustrate the CELLAR programming model by describing a parallel model of animal migration. The example is based on the NOYELP model developed by a group of modelers and biologists at the University of Tennessee [14]. NOYELP is a spatially-explicit individual-based model that simulates the search, movement and foraging activities of groups of animals across a landscape. The landscape is composed of a grid of cells where a cell represents a portion of the landscape. An initial quantity of available forage is assigned to each grid cell based on its habitat type and burn status at the beginning of each simulation. The algorithm used in NOYELP to simulate forage search and movement of the animals assumes that if an animal is located on a cell with available forage then the animal stops and grazes. Otherwise, the animal will search in concentric squares, up to a radius equal to the maximum moving distance, for another cell with available forage. NOYELP model does not consider some real features of the animals such as the possibility to use the sense of smell and the sight to determinate the availability of forage. In our model the animals can see in all directions up to a value equal to the perception rating and all the regions included within this radius can be monitored. We suppose that the forage is distributed only in some areas of the landscape defined as rectangular regions.

```
#define NumRegion 6
#define perception_rate 20
#define NumAnim 50

cadef
{
 dimension 2;
 radius 1;
 state(float forage, short animaldir[NumAnim]);
 neighbor Moore[8]([0,-1]North, [1,-1]NE,
         [1,0]East,[1,1]SE,[0,1]South,
         [-1,1]SW, [-1,0] West, [-1,-1]NW);

   parameter (thres_forage 0.1);

   region  sight[6](
     meadow1 (10,20,10,30 ,0,0,1,2000,5),
     meadow2 (10,20,10,30 ,0,0,1,2000,5),
     meadow3 (10,20,10,30 ,0,0,1,2000,5),
     meadow4 (10,20,10,30 ,0,0,1,2000,5),
     meadow5 (10,20,10,30 ,0,0,1,2000,5),
     meadow6 (10,20,10,30 ,0,0,1,2000,5));
}
```

Figure 5. The CELLAR declarations of animal migration model.

Figure 5 shows the CELLAR declarations for the model of animal migration. The cell state is composed of 2 substates which describe the quantity of forage and the moving direction of 50 animals. The cell's neighborhood contains 8 cells. Regions are six and represent the zones where is distributed the forage. They can be monitored each five iterations.

Figure 6 shows an outline of the transition function of the model. It is composed of two steps because CELLAR does not allow to modify the state of the neighbour cells. In the first step, for each cell where there are animals (cell_animaldir[k] != 0), we calculate the direction of movement of the animals. In the second step, the cell indicated as destination "transports" the animal from the neighbour site containing the animal toward itself.

If the cell contains animals and the value of the current iteration is included in the time period from 1 to 2000 and the monitor step is equals to 5 then the current cell calculates, by the *distance* function, its visible regions. For each visible region, by the *SumRegion* function, the available quantity of forage is calculated. If an animal is within a region and the amount of forage is greater than a threshold (*thres_forage*) then it stops and grazes, otherwise it moves at random within the region. However, if the quantity of forage available in the region is less than a threshold then the animal migrates to the nearest region that has an amount of forage above the threshold. An animal located outside a region moves in the direction in which the rate amount of forage/distance is maximized.

7. Performance

In this section we present the performance results of the CELLAR program that simulates the animal migration model. The model has been implemented on a Meiko CS-2 parallel machine. The CS-2 is a distributed memory MIMD parallel computer. It consists of *Sparc* based processing nodes running the *Solaris* operating system on each node, so it resembles a cluster of workstations connected by a fast network. Each node is composed of one or more Sparc processors, a communication co-processor, the Elan processor, that connects each node to a fat tree network built from Meiko 8x8 crosspoint switches. Our machine is a 12 processors CS-2 based on 200 Mhz *Hypersparc* processors with 256 Mbytes of memory on each processor and running Solaris 2.5.1. The model has been tested with different grid sizes and with six regions. Figure 7 shows a snapshot of migration animal model simulated by the CARAVEL environment. Table 1 shows the elapsed time of the execution of 100 steps, and the speedup measures of the parallel model implementation using different grid sizes on 1, 2, 4 and 8 processors. The performance showed for the 128x64 grid are better than the 64x128 grid. In fact, the automaton is subdivided along the x-axis and this implies that a smaller number of messages is exchanged between two consecutive partitions allocate on different nodes.

```
if (step %2 == 1)
{
 for (k=0; k < NumAnim; k++)
    {
     if (cell_animaldir[k] !=0 )
     {
      if((InTempRegion(meadow1)&&(InTempRegion(meadow2)&&
       (InTempRegion(meadow3)&&(InTempRegion(meadow4)&&
       (InTempRegion(meadow5)&& (InTempRegion(meadow6))
        {
         for (i=0; i < NumRegion; i++)
          {
           dist[i]= distance(sight[i]);
           if (dist[i] < perception_rate)
           {
             sum[i] = SumRegion(sight[i]_forage, &suc);
               see[i] = TRUE;
           }
           else
            see[i]=FALSE;
      }
     if ((InRegion(meadow1)|| InRegion(meadow2) ||InRegion(meadow3)||
          InRegion(meadow4) ||InRegion(meadow5)|| InRegion(meadow6))
      {
       if (cell_forage <= tresh_forage)
          dir = random_migration (dist,sum,see);
          else
            {
              grazes(cell_forage);
              dir = motionless;
            }
        }
       else
         dir = choose_direction (dist, sum, see);
       update(cell_animaldir[k], dir);
      }
     }
    }
       else /* movement rule of the animals */
       {
        for(k=0; k <NumAnimal ; k++)
         if (cell_animaldir[k] != motionless)
           {
           animtemp = 0;
             for (i=1; i<= 8; i++)
               if(moore[i]_animaldir[k] == i)
                   animtemp=i;
             update(cell_animaldir[k], animtemp);
          }
    }
```

Figure 6. The CELLAR transition function of animal migration model.

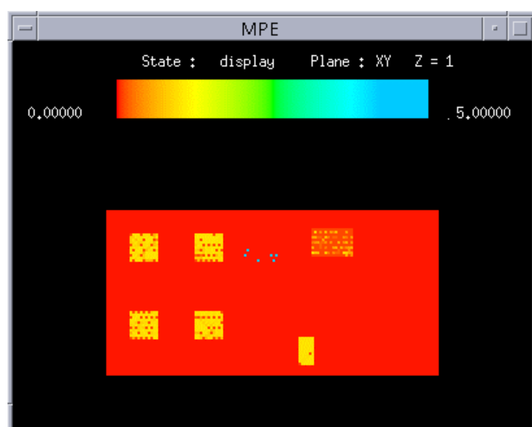

Figure 7. A snapshot of migration animal simulation.

PEs	64x64		64x128		128x64	
	Time (sec.)	Speedup	Time (sec.)	Speedup	Time (sec.)	Speedup
1	220.50	1	438.25	1	439.81	1
2	113.07	1.95	223.60	1.96	222.13	1.98
4	60.41	3.65	117.03	3.74	114.02	3.85
8	36.81	5.99	67.10	6.53	63.11	6.97

Table 1. Elapsed time and speedup for different grid sizes of the cellular automaton.

8. Conclusions

In this paper we have presented the constructs of the CELLAR language to extend the cellular programming with the concept of regions and demonstrated how this concept simplifies programming and allows the building of more complex models. The CELLAR programming environment offers to a user an interface that abstracts from underlying hardware and ensures portability and intellectual abstraction. The region-based programming model of CELLAR is used in the COLOMBO project within the ESPRIT framework. The main objective of this project is the application of parallel computing to the simulation of the bioremediation of contaminated soils using CA models.

Acknowledgements

This research has been partially funded by the CEC ESPRIT project n° 24907.

References

[1] M. Sipper, " The Emergence of Cellular Computing", IEEE Computer, vol.32, n. 7, July 1999.

[2] T. Worsch, Programming Environments for Cellular Automata, *Proc. Cellular Automata for Research and Industry (ACRI 96), Springer-Verlag*, London, pp. 3-12, 1997.

[3] J.D. Eckart, Cellang 2.0:Reference manual. ACM Sigplan Notices, vol. 27, n. 8, pp.107-112, 1992.

[4] C. Hochberger, R. Hoffmann, CDL – a Language for Cellular Processing. *In Proc. 2nd Intern. Conference on Massively Parallel Computing Systems*, 1996.

[5] G. Junger, *Cellular Automaton Tool user manual*, GMD,1994.

[6] F. Seutter, CEPROL – a Cellular Programming Language, *Parallel Computing*, vol. 2 , pp.327-333, 1985.

[7] G. Spezzano and D. Talia, A High-Level Cellular Programming Model for Massively Parallel Processing, in: *Proc. 2nd Int. Workshop on High-Level Programming Models and Supportive Environments (HIPS97)*, IEEE Computer Society, pp. 55-63, 1997.

[8] G. Spezzano, D. Talia and al. A Parallel Cellular Tool for Interactive Modeling and Simulation, *IEEE Computational Science & Engineering,* 3:3, pp. 33-43, 1996.

[9] Folino G., Pizzuti C., Spezzano G., Combining Cellular Genetic Algorithms and Local Search for Solving Satisfiability Problems, *Proc. of 10th IEEE International Conference Tools with Artificial Intelligence (ICTAI'98),* IEEE Computer Society, pp. 192-198 , 1998.

[10] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder, Regions: An Abstraction for Expressing Array Computation, UW CSE Technical Report, UW-CSE-98-10-02, October, 1998.

[11] J.Vetter, **K. Schwan,** High Performance Computational Steering of Physical Simulations, *Proc. 11th International Parallel Processing Conference(IPPS'96),* IEEE Computer Society Press, pp.128-132, 1997.

[12] B.P Hansen, Parallel Cellular Automata: A Model for Computational Science, *Concurrency: Practice and Experience,* 5, pp. 425-448, 1993

[13] M. Cannataro, S. Di Gregorio, R. rongo, W. Spataro, G. Spezzano, and D. Talia, A Parallel Cellular Environment on Multicomputers for Computational Science, *Parallel Computing,* 21, pp. 803-824, 1995.

[14] E. Uziel and M.W. Berry, Parallel Model of Animal Migration in Northern Yellowstone Park, *Int. J. Supercomputer Applications and High Performance Computing,* vol. 9, n.4, pp.237-255,1996.