Solving Problems on Parallel Computers by Cellular Programming

Domenico Talia

ISI-CNR
c/o DEIS, UNICAL,
87036 Rende (CS), Italy
Email : talia@si.deis.unical.it

Abstract. Cellular automata can be used to design high-performance *natural solvers* on parallel computers. This paper describes the development of applications using CARPET, a high-level programming language based on the biology-inspired cellular automata theory. CARPET is a programming language designed for supporting the development of parallel high-performance software abstracting from the parallel architecture on which programs run. We introduce the main constructs of CARPET and discuss how the language can be effectively utilized to implement *natural solvers* of real-world complex problems such as forest fire and circuitry simulations. Performance figures of the experiments carried out on a MIMD parallel computer show the effectiveness of our approach both in terms of execution time and speedup.

1. Introduction

Cellular processing languages based on the cellular automata (CA) model [10] represent a significant class of restricted-computation models [8] inspired to a biological paradigm. They are used to solve problems on parallel computing systems in a wide range of application areas such as biology, physics, geophysics, chemistry, economics, artificial life, and engineering.

CA provide an abstract setting for the development of *natural solvers* of dynamic complex phenomena and systems. Natural solvers are algorithms, models and applications that are inspired by processes from nature. Besides CA, typical examples of natural solvers methods are neural nets, genetic algorithms, and Lindenmayer systems. CA represent a basic framework for *parallel natural solvers* because their computation is based on a massive number of *cells* with local interactions that use discrete time, discrete space and a discrete set of state variable values.

A cellular automaton consists of one-dimensional or multi-dimensional lattice of *cells*, each of which is connected to a finite neighborhood of cells that are nearby in the lattice. Each cell in the regular spatial lattice can take any of a finite number of discrete state values. Time is discrete, as well, and at each time step all the cells in the lattice are updated by means of a local rule called *transition function*, which determines the cell's next state based upon the states of its neighbors. That is, the state

of a cell at a given time depends only on its own state and the states of its nearby neighbors at the previous time step. Different neighborhoods can be defined for the cells. All cells of the automaton are updated synchronously. The global behavior of the system is determined by the evolution of the states of all cells as a result of multiple interactions. An interesting extension of the CA standard model is represented by *continuous CA* that allow a cell to contain a real, not only an integer value. This class of automata is very useful for simulation of complex phenomena where physical quantities such as temperature or density must be taken into account.

CA are intrinsically parallel and they can be mapped onto parallel computers with high efficiency, because the communication flow between processors can be kept low due to locality and regularity. We implemented the CA features in a high-level parallel programming language, called CARPET [9], that assists cellular algorithms design. Unlike early cellular approaches, in which cell state was defined as a single bit or a set of bits, we define the state of a cell as a set of typed substates. This extends the range of applications to be programmed by cellular algorithms. CARPET has been used for programming cellular algorithms in the CAMEL environment [2, 4].

The goal of this paper is to discuss how the language can be effectively utilized to design and implement scientific applications as parallel natural solvers. The rest of the paper is organized as follows. Sections 2 and 3 introduce the constructs of CARPET and the main architectural issues of the CAMEL system. Section 4 presents a simple CARPET example and describes how the language can be utilized to model the forest fire problem. Finally, performance figures that show the scalability of CARPET programs on a multicomputer are given.

2. Cellular Programming

The rationale for CARPET (*CellulAR Programming EnvironmenT*) is to make parallel computers available to application-oriented users hiding the implementation issues resulting from architectural complexity. CARPET is a high-level language based on C with additional constructs to define the rules of the transition function of a single cell of a cellular automaton. A CARPET user can program complex problems that may be represented as discrete cells across 1D, 2D, and 3D lattices.

CARPET implements a cellular automaton as a SPMD program. CA are implemented as a number of processes each one mapped on a distinct processing element (PE) that executes the same code on different data. However, parallelism inherent to its programming model is not apparent to the programmer. According to this approach, a user defines the main features of a CA and specifies the operations of the transition function of a single cell of the system to be simulated. So using CARPET, a wide variety of cellular algorithms can be described in a simple but very expressive way.

The language utilizes the control structures, the types, the operators and the expressions of the C language. A CARPET program is composed of a declaration part that appears only once in the program and must precede any statement (except those of C pre-processor) and of a program body. The program body has the usual C statements and a set of special statements defined to access and modify the state of a

cell and its neighborhood. Furthermore, CARPET permits the use of C functions and procedures to improve the structure of programs.

The declaration section includes constructs that allow a user to specify the dimensions of the automaton (**dimension**), the radius of the neighborhood (**radius**), the pattern of the neighborhood (**neighbor**), and to describe the state of a cell (**state**) as a set of typed substates that can be: *shorts, integers, floats*, *doubles* and arrays of these basic types. The use of *float* and *double* substates allows a user to define *continuous CA* for modeling complex systems or phenomena. At the same time, formal compliance with the standard CA definition can be easily assured by resorting to a discretized set of values.

In CARPET, the state of a cell is composed of a set of typed substates, unlike classical cellular automata where the cell state is represented by a few bits. The typification of the substates allows us to extend the range of the applications that can be coded in CARPET simplifying writing the programs and improving their readability. Most systems and languages (for example CELLANG [6]) define the cell substates only as integers. In this case, for instance, if a user must store a real value in a substate then she/he must write some procedures for the data retyping. The writing of these procedures makes the program longer and difficult to read or change. The CARPET language frees the user of this tedious task and offers her/him a high level in state declaration. A type identifier must be included for each substate. In the following example the state is constituted of three substates:

```
state (int particles, float temperature, density);
```

A substate of the current cell can be referenced by the variable cell_substate (e.g., cell_speed). To guarantee the semantics of cell updating in cellular automata the value of one substate of a cell can be modified only by the **update** operation. 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.

CARPET allows a user to define a logic neighborhood that can represent a wide range of different neighborhoods inside the same radius. Neighborhoods can be asymmetrical or have any other special topological properties (e.g., hexagonal neighborhood). The neighbor declaration assigns a name to specified neighboring cells of the current cell and a vector name that can be used as an alias in referring to a neighbor cell. For instance, the von Neumann and Moore neighborhoods shown in figure 1, can be defined as follows:

```
neighbor Neumann[4]([0,-1]North,[-1,0]West, [0,1]South, [1,0]East);

neighbor Moore[8] ([1,-1]NEast, [0,-1]North, [-1,-1]NWest, [-1,0] West,
                   [1,0]East ,[-1,1]SWest, [0,1]South [1,1]SEast);
```

A substate of a neighbor cell is referred to, for instance, as NEast_speed. By the vector name the same substate can be referred to also as Moore[0]_speed. This way of referencing simplifies writing loops in CARPET programs.

CARPET permits the definition of global parameters that can be initialized to specific values (e.g., **parameter** (viscosity 0.25)). The value of a parameter is the same in each cell of the automaton. For this reason, the value of each parameter cannot be changed in the program but it can only be modified, during the simulation,

by the user interface (UI). CARPET defines also a mechanism for programming non-deterministic rules by a random function. Finally, a user can define cells with different transition functions by means of the `Getx`, `Gety`, `Getz` functions that return the value of the coordinates X, Y, and Z of the cell in the automaton.
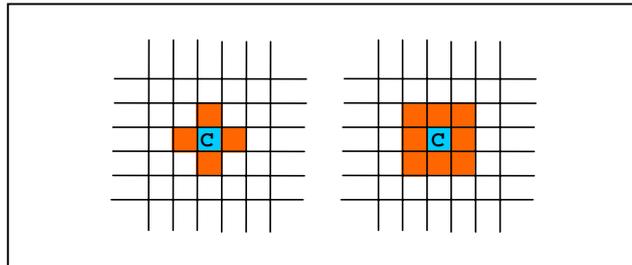
Fig. 1. The von Neumann and Moore neighborhoods in a two-dimensional cellular automaton .

CARPET does not include constructs for configuration and visualization of the data, unlike other cellular languages. As a result, the same CARPET program can be executed with different configurations. The size of lattice, as other details, of a cellular automaton are defined by the UI of the CARPET environment. The UI allows, by menus, to define the size of a cellular automaton, the number of processors on which the automaton will be executed, and to choose colors to be assigned to the cell substates to support the graphical visualization of their values.

3. A Parallel Environment for CARPET

Parallel computers represent the most natural architecture where CA programming environments can be implemented. In fact, when a sequential computer is used to support the simulation, the execution time might become very high since such computer has to perform the transition function for each cell of the automaton in a sequential way. Thus, parallel computers are necessary as a practical support for the effective implementation of high-performance CA [1].

The approach previously mentioned motivated the development of CAMEL (Cellular Automata environMent for systEms ModeLing), a parallel software architecture based on the cellular automata model that constitutes the parallel run-time system of CARPET. The latest version of CAMEL named CAMELot (*CAMEL open technology*) is a portable implementation based on the MPI communication library. It is available on MIMD parallel computers and PC clusters.

The CAMEL run-time system is composed of a set of *macrocell* processes, each one running on a single processing element of the parallel machine, and by a *controller* process running on a processor that is identified as the *Master* processor.

The CAMEL system uses the SPMD approach for executing the CA transition function. Because of the number of cells that compose an automaton is generally greater than the number of available processors, several elementary cells are mapped on each *macrocell* process. The whole set of the *macrocells* implement a cellular automaton and it is called the CAMEL *Parallel Engine*. As mentioned before,

CAMEL provides also a user interface to configure a CARPET program, to monitor the parameters of a simulation and dynamically change them at run time.

CAMEL implements a form of block-cyclic data decomposition for mapping cells on the processors that aims to address the problem of load imbalance experienced when the areas of active cells are restricted to one or few domains and the rest of lattice may be inactive for a certain number of steps [2]. This load balancing strategy divides the computation of the next state of the active cells among all the processors of the parallel machine avoiding to compute the next state of cells that belongs to a stationary region. This is *a domain decomposition* strategy similar to the *scattered decomposition* technique.

4. Programming Examples

To describe practically cellular programming in CARPET, this section shows two cellular programs. They are simple but representative examples of complex systems and phenomena and can explain how the natural solver approach can be exploited by the CARPET language.

4.1. The wireworld program

This section shows the simple wireworld program written by CARPET. This program should familiarize the reader with the language approach. In fact, figure 2 shows how the CARPET constructs can be used to implement the *wireworld* model proposed in the 1990 by A. K. Dewdney [3] to build and simulate a wide variety of circuitry.

In this simple CA model each cell has 4 possible states: space, wire, electron head or electron tail. This simple automaton models electrical pulses with heads and tails, giving them a direction of travel. Cells interact with their 8 neighbours by the following rules: space cells forever remain space cells, electron tails turn into wire cells, electron heads turn into electron tails, wire cells remain wire cells unless bordered by 1 or 2 electron heads.

By taking special care in the arrangement of the wire (initial configuration of the lattice), with these basic rules electrons composed of heads and tails can move along wires and you can build and test diodes, OR gates, NOT gates, memory cells, wire crossings and much more complex circuitry.

4.2. A forest fire model

We show here the basic algorithm of a CARPET implementation of a real life complex application. Preventing and controlling forest fires plays an important role in forest management. Fast and accurate models can aid in managing the forests as well as controlling fires. This programming example concerns a simulation of the propagation of a forest fire that has been modeled as a two-dimensional space partitioned into square cells of uniform size (figure 3).

```
    #define space      0
    #define wire       1
    #define electhead  2
    #define electail   3

     cadef
     {
      dimension 2;           /*bidimensional lattice */
      radius 1;
      state (short content);
       neighbor moore[8] ([0,-1]North,[-1,-1]NorthWest, [-1,0]West,
                          [-1,1]SouthWest,[0,1]South, [1,1] SouthEast,
                          [1,0]East, [1,-1]NorthEast);
     }
     int i; short count;
     {
       count = 0;
       for (i = 0; i<8; i++)
          if (moore[i]_content == electhead)
             count = count + 1;
       switch (cell_content)
        {
          case electail : update(cell_content, wire); break;
          case electhead: update(cell_content, electail); break;
          case wire      : if (count == 1 || count == 2)
                            update(cell_content, electhead);
        }
     }
```

Fig. 2. The wireworld program written in CARPET.

Each cell can represents a portion of land of 10x10 meters. Cells in the lattice can have the values included between '0' and '3'. The fire is represented by '0' value, the ground is represented by '1' value, and trees are represented by '2' value if they are alive or by '3' value if they are dead. Fire spreads from a cell which is on fire to a Moore neighbor that is treed, but not on fire.

Each tree follows a simple rule: if it catches fire from one of its neighbors cells, it will burn and spread the fire to any neighboring trees. The fire spreads in all directions. The fire's chance of diffusing towards all the forest depends critically on the density (dens) of trees in the forest.

The most interesting aspect of this simulation is that using a very simple rule it is possible to observe a real complex behavior in the fire evolution. The fire spreads from tree to tree and in some places the fire reaches a dead end surrounded by an empty region. Running the simulation with different densities of trees, by changing the percentage of tree-cells in the lattice, has been observed different spreading of the fire across the forest.

This example shows as using a high-level language designed for programming cellular algorithms can strongly simplify the algorithms design process and reduce the program code. Moreover, the programming environment allows a user to observe the dynamic evolution of the fire spreading on a computer display where the application is visualized as in figure 4.

```
#define tree 2
#define fire 0
#define dead 3
#define land 1

cadef
{ dimension 2;
  radius 1;
  state (short ground);
  neighbor moore[8] ( [0,-1]North,[-1,-1]NorthWest, [-1,0]West,
                    [-1,1]SouthWest,[0,1]South, [1,1] SouthEast,
                    [1,0]East, [1,-1]NorthEast);
  parameter (dens 0.6);
}
 float px;
{ if (step == 0)
  {
   px = ((float) rand())/RAND_MAX;
   if (px < dens)
      update(cell_ground, tree);
   else
      update(cell_ground, land);
  }
 else
 if((cell_ground == tree) && (North_ground == fire ||
     South_ground == fire || East_ground == fire ||
     West_ground == fire || NorthWest_ground == fire ||
     SouthWest_ground == fire ||
     SouthEast_ground == fire || NorthEast_ground == fire))
        update(cell_ground, fire);
 else
   if (cell_ground == fire)
      update(cell_ground, dead);
}
```

Fig. 3. A simple forest fire program written in CARPET.

The simple forest fire model discussed here could be extended considering forests with roads, rivers, and houses, by defining a state value for each cell that represents one of these objects. The transition function will take into account the behavior of fire when these cells will be encountered. It can also be extended considering different weather conditions, wind speed, fuel types, and terrain conditions.

4.3 Performance results

Here are presented some performance figures obtained from the implementation of the forest fire model in CARPET on a Meiko CS2 parallel computer. Table 1 shows the elapsed time (in seconds) for the execution of 1000 steps of the simulation using different lattice sizes on different processor sets.
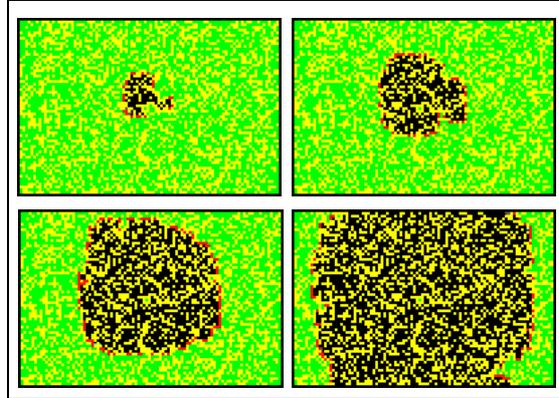
Fig. 4. Four snapshots of visualization of the forest fire simulation by CARPET.

The results are satisfying both in terms of execution time and speedup. In particular, figure 5 shows the speedup measures. On 8 processors, the measured speedup goes from 5.6 when a small lattice is used to 7.9 when we simulated a larger lattice that exploits the computational power of the parallel machine in a more efficient way.

Table 1. Execution times (in sec.) for 1000 steps for three lattice sizes on different processors.

Processors	Lattice sizes		
	56 x 56	112 x 112	224 x 224
1	7.46	27.77	118.93
2	3.66	14.18	59.47
4	1.98	7.03	29.64
8	1.32	3.95	14.91
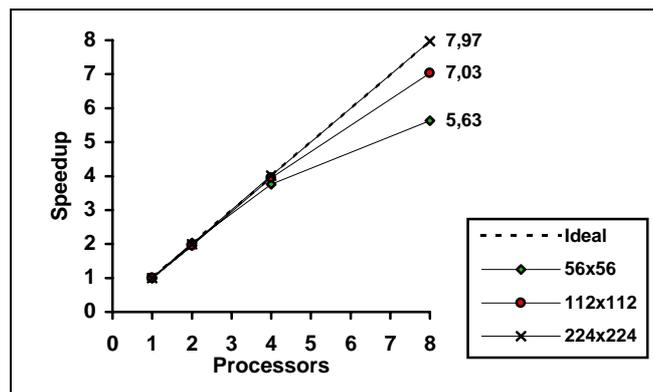
Fig. 5. Speedup of the CARPET forest fire program using different lattice sizes.

These values represent a notable result. In fact, speedup shows how much faster the simulation runs by increasing the number of PEs of a parallel computer. Thus, this

CARPET program appears to be scalable as occurred with other parallel cellular applications developed by the language [4, 5].

6. Conclusion

CARPET has been successfully used to implement several real life applications such as landslide simulation, lava flow models, freeway traffic simulation, image processing, and genetic algorithms [4, 5, 7]. It can also be used to solve parallel computing problems such as routing strategies, task scheduling and load balancing, parallel computer graphics and cryptography.

   A portable MPI-based implementation of CARPET and CAMEL (called *CAMELot*) has been implemented recently on MIMD parallel computers such as the Meiko CS-2, SGI machines and PC-Linux clusters. In our opinion and on the basis of our experience, high-level languages such as CARPET can enlarge the practical use of cellular automata in solving complex problems according to the natural solvers approach while preserve high performance and expressiveness.

References

[1]   B. P. Brinch Hansen, Parallel cellular automata: a model for computational science, *Concurrency: Practice and Experience*, **5**:425-448, 1993.

[2]   M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, A parallel cellular automata environment on multicomputers for computational science, *Parallel Computing*, **21**:803-824, 1995.

[3]   A. K Dewdney, Cellular automata programs that create wireworld, rugworld and other diversions, *Scientific American*, **262**: 146-149, 1990.

[4]   S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, A parallel cellular tool for interactive modeling and simulation, *IEEE Computational Science & Engineering* **3**:33-43, 1996.

[5]   S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, High performance scientific computing by a parallel cellular environment*, Future Generation Computer Systems*, North-Holland, Amsterdam, **12**:357-369, 1997.

[6]   J. D. Eckart, Cellang 2.0: reference manual, *ACM Sigplan Notices*, **27**: 107-112, 1992.

[7]   G. Folino, C. Pizzuti, and G. Spezzano, Solving the satisfiability problem by a parallel cellular genetic algorithm, *Proc. of Euromicro Workshop on Computational Intelligence*, IEEE Computer Society Press, pp. 715-722, 1998.

[8]   D.B. Skillicorn and D. Talia, "Models and languages for parallel computation", *ACM Computing Survey*, **30**:123-169, 1998.

[9]   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.

[10] J. von Neumann, *Theory of Self Reproducing Automata*, University of Illinois Press, 1966.