

# Language Constructs and Run-Time System for Parallel Cellular Programming

Giandomenico Spezzano and Domenico Talia

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

**Abstract.** This paper describes the main features of CARPET, a high-level programming language based on cellular automata theory. The language has been designed for supporting the development of parallel high-performance software abstracting from the parallel architecture on which programs run. A CARPET user can write cellular programs to describe the actions of a very large number of simple active agents interacting locally. The CARPET run-time system allows a user to observe, also in a graphical format, the global results that arises from their parallel execution.

## 1 Introduction

The lack of high-level languages, tools, and application-oriented environments often limits the design and implementation of parallel algorithms that are portable, efficient, and expressive. The *restricted-computation structures* represent one of the most important models of parallel processing [8]. The interest for these models is due to the possibility to restrict the form of computations so as to restrict communication volume achieving high performance. *Restricted-computation* models offer a user a structured paradigm of parallel programming and improve the performance of the parallel algorithms reducing the overheads due to the communication *latency*. Further, tools can be designed to estimate the performance of various constructs of a high-level language on a specific parallel architecture.

Cellular processing languages based on the cellular automata (CA) model [10] represent a significant example of restricted-computation that it is used to model parallel computation for a large number of applications in biology, physics, geophysics, chemistry, economics, artificial life, and engineering. 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*, that 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. The most common neighborhoods in the two-dimensional case are the von Neumann neighborhood consisting of the North, South, East, West neighbors and the Moore neighborhood composed of eight neighbor cells. The global behavior of the an automaton is defined by the evolution of the states of all cells as a result of multiple interactions.

CA are intrinsically parallel so they can be simulated onto parallel computers running the cell transition functions in parallel with high efficiency, as the communication flow between processors can be kept low. In fact, in our approach, a cellular algorithm is composed of all the transition functions of cells that compose the lattice. Each transition function generally contains the same local rule, but it is also possible to define some cells with different transition functions (inhomogeneous cellular automata).

According to this approach, we designed and implemented a high-level programming language, called CARPET (Cellular Programming Environment) [9], that allows a user to design cellular algorithms. In particular, CARPET has been used for programming cellular algorithms in the CAMEL (Cellular Automata environment for systEmS ModeLing environment) [3]. A user can design cellular programs by CARPET describing the actions of many simple active agents (implemented by the cells) interacting locally, then the CAMEL system runs cell transition functions in parallel allowing a user to observe the global complex evolution that arises from all the local interactions. A number of cellular programming languages such as Cellang [4], CDL [6], and CARP [7] have been defined in the last decade. However, none of those contains all the features of CARPET neither a parallel run-time support for them has been implemented till today.

## 2 CARPET

The rational of CARPET is to make parallel computers available to application-oriented users hiding the implementation issues coming from their architectural complexity. A CARPET user can program complex problems that may be represented as discrete across a lattice. Parallelism inherent to its programming model is not apparent to the programmer.

CARPET implements a cellular automaton as an SPMD program. CA are implemented as a number of processes each one mapped on a distinct PE that executes the same code on different data. According to this approach, a user must specify by CARPET only the transition function of a single cell of the system he wants to simulate. The language uses the control structures, the types, the operators of the C language. A CARPET program is composed by a declaration part that appears only once in the program and must precede any statement (except those of C pre-processor) and by a program body. The program body has the usual C statements, without I/O instructions, and a set of statements to deal with the state of a cell and its neighborhood. Further, CARPET users may use C functions or procedures to improve the structure of programs.

The declaration section includes constructs to define the dimensions of the automaton (dimension), the radius of the neighborhood (radius), the type of the neighborhood (neighbor), and to specify the state of a cell (state) as a set of typed substates that can be *char*, *shorts*, *integers*, *floats*, *doubles* and *arrays* of these basic types.

The dimension declaration defines the number of dimensions of the automaton in a discrete Cartesian space. The maximum number of dimensions allowed in the current implementation is 3. Radius defines the set of cells that can compose the neighborhood of a cell.

In CARPET the state of a cell is composed of a record of typed substates, unlike classical cellular automata where the cell state is represented by a few bits. The typification of the substates extends the range of the applications that can be coded in CARPET simplifying the writing the programs and improving their readability. In the following example the state is composed of three substates:

```
state (short direction, float mass, speed);
```

A substate of the current cell can be referred by the variable `cell_substate` (eg., `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 execution of the update statement, the value of the substate, in the current iteration, is unchanged. The new value does take effect in the next iteration.

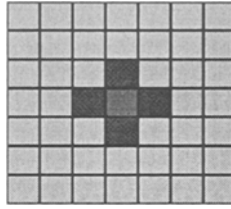
The neighbor declaration assigns a name to a set of specified neighbor cells (of the current cell). This mechanism allows a cell to access by name the values of the substates of its neighbor cells. Neighborhoods can be asymmetrical or have any other special topological properties (e.g., hexagonal). Furthermore, in the neighbor declaration it must be defined the name of a vector that has as length the number of elements composing the logic neighborhood. The name of the vector can be used as an alias in referring to the neighbor cell. For instance, the von Neumann neighborhood shown in figure 1, can be defined as follows:

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

A substate of a neighbor cell is referred, for instance, as `North_speed`. Using the vector name the same substate can be referred also as `Neum[0]_speed`. This referring way makes simpler to write loops in CARPET programs.

CARPET allows the program to know the number of iterations that have been executed by the predefined variable `step`. `Step` is silently updated by the run-time system. Initially its value is 0 and it is incremented by 1 each time all the cells of the automata are updated. This feature allows a user to define also time-dependent neighborhoods. The `step` variable permits also to change dynamically the values of the substates dependent upon the iterations.

In modeling a complex system, it is often necessary to describe some global features of the system. CARPET allows a user to define global parameters and initialize them to specific values. The value of a parameter is the same in every cell of the automaton. For this reason, the value of each parameter cannot be



**Fig. 1.** The von Neumann neighborhood in a two-dimensional CA lattice.

changed in the program but it can only be modified, during the simulation, by the UI. An example of parameter declaration in CARPET is

```
parameter (permeability 0.9) ;
```

Input and output of a CARPET program can be performed by files or by the edit function of the UI. A file can contain the values of one substate of all cells, so these values can be loaded at the step 0 to initialize the automaton. The output of a CARPET program can automatically be saved in files at regular intervals to be post-processed by mathematical or visualization tools. CARPET offers a user the possibility to define non-deterministic rules using of a random function. Finally, CARPET allows a user to define cells with different transition functions by means of the Getx, Gety, Getz operations that return the value of coordinates X, Y, and Z of a cell in the automaton.

Differently from other cellular languages, CARPET does not provide statements to configure the automata, to visualize the cell values or to define the process-to processor mapping. These features can be defined by means of the GUI of its runtime system. The CAMEL GUI allows a user to define the size of cellular automata, the number of the processors onto which an automaton must be executed, and choose the colors to be assigned to the cell substates to support the graphical visualization of their values. By excluding constructs for configuration and visualization from the language, it is possible to execute the same CARPET compiled code using different configurations.

### 3 A Parallel Cellular Run Time System

Parallel computers are the best practical support for the effective implementation of high-performance CA [1]. According to this basic idea has been developed CAMEL. It is a parallel software system based on the cellular automata model that constitutes the parallel run-time system of CARPET. CAMEL has been implemented on a parallel computer composed of a mesh of Transputers connected to a host node [2]. CAMEL provides a GUI to configure a program, to monitor the parameters of a simulation and to dynamically change them at run time. The parallel execution of cellular algorithms is implemented by the parallel execution of the transition function of each cell in the Single Program Multiple

Data (SPMD) way. A portable implementation of CAMEL for MIMD parallel computers based on the MPI communication library is developing.

The CAMEL system is composed by 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 master processor. Each *Macrocell* process implements an automaton partition that includes several elementary cells, and it makes use of a communication system that handles the data exchange among cells. All the *Macrocells* of the lattice execute in parallel the local rule that results in a global transformation of the whole lattice state. CAMEL uses a load balancing strategy for assigning lattice partitions to the processors of a parallel computer [2]. This load balancing is a domain decomposition strategy similar to the scattered decomposition technique.

## 4 A Simple Program

This section shows a simple program written by CARPET. This example should familiarize the reader with the CARPET approach. The program in figure 2 shows how the CARPET constructs can be used to implement the *parity rule* algorithm. The parity rule is a very simple example of "self-reproduction" in cellular automata; an initial pattern is replicated at some specific iteration that is a power of two. The cells can have 0 or 1 values only. A cell takes the sum of its neighbors and goes to 1 if the sum is odd, or to 0 if the sum is even. 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.

```
caef
{ 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); /*updating the state of a cell*/
}
```

Fig. 2. The parity rule algorithm written in CARPET.

## 5 Performance

Together with the minimization of elapsed time, scalability is a major goal in the design of parallel computing applications. Scalability shows the potential ability of parallel computers to speedup their performance by adding more processing elements. The scalability of CARPET programs has been measured increasing the number of PEs used to solve the same problem, and using the same number of processing elements to solve bigger problems [2].

The speedup obtained on 32 Transputers with an automaton composed of 224x70 cells which simulated the Ontake mountain (Japan) landslide was 25.0 (78.1% efficient). Table 1 shows the times (in seconds) and the speed-up obtained running the landslide simulation on CAMEL using 1, 2, 4, 8, 16 and 32 PEs. The second column shows the number of cells, which are mapped, on each PE. Similar performance results have been obtained in other applications developed by CARPET [3].

**Table 1.** Execution times and speed-up.

Number of PEs	Number Cells/PE	Time for 2000 steps	Speed-up
1	15680	10231.59	1
2	7840	6392.48	1.6
4	3920	3181.40	3.2
8	1960	1447.96	7.0
16	980	728.11	14.0
32	490	407.84	25.0

## 6 Final Remarks and Future Work

As stated also by M. J. Flynn [5], the cellular automata model is a new mathematical way to represent problems that allows to effectively use parallel computers achieving scalable performance.

In this paper, we described the CARPET programming language designed for programming cellular algorithms on parallel computers. CARPET has been used successfully to implement several real life applications such as landslide simulation, lavaflow models, freeway traffic simulation, image processing, and genetic algorithms. The experience during the design, implementation, and use of the CARPET language showed us that high-level languages are very useful in the development of cellular algorithms for solving complex problems in science and engineering.

Currently, the CARPET language is used in the COLOMBO (Parallel Computers improve cLean up of sOils by Modelling BiOremediation) project within the ESPRIT framework. The COLOMBO main objective is the use of CA models for the bioremediation of contaminated soils. This project is developing a

portable MPI-based implementation of CARPET and CAMEL on MIMD parallel computers such as the Meiko CS-2, the CRAY T3E, and workstation clusters. The new implementation will make the CARPET programs portable on a large number of MIMD machines.

**Acknowledgements** This research has been partially funded by the CEC ESPRIT project nr 24,907.

## 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. 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.
4. J. D. Eckart, Cellang 2.0: Reference Manual. *ACM Sigplan Notices*, 27:107-112, 1992.
5. M.J. Flynn, Parallel Processors Were the Future and May Yet Be. *IEEE Computer* 29:152, 1996.
6. C. Hochberger and R. Hoffmann, CDL - a Language for Cellular Processing. In: *Proc. 2nd Intern. Conference on Massively Parallel Computing Systems*, IEEE Computer Society Press, 1996.
7. G. Junger, Cellular Automaton Tool User Manual. GMD, Sankt Augustin, Germany, 1994.
8. D.B. Skillicorn and D. Talia, Models and Languages for Parallel Computation. *ACM Computing Survey*, 30, 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, pages 55-63, 1997.
10. J. von Neumann, *Theory of Self Reproducing Automata*. University of Illinois Press, 1966.