A High-Level Cellular Programming Model for Massively Parallel Processing

Giandomenico Spezzano and Domenico Talia

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

Abstract

Cellular automata are used for designing high-performance applications in many areas. This paper describes CARPET, a high-level programming language based on the cellular automata model. CARPET is a programming language designed to support the development of parallel high performance software. It exploits the computing power of a highly parallel computer releasing a user from using explicit parallel constructs. A CARPET implementation has been used for programming cellular algorithms in the CAMEL parallel environment. By CARPET a user might write programs to describe the actions of thousands of simple active agents interacting locally, then the CAMEL environment allows a user to observe the global complex evolution that arises from their parallel execution and their local interactions.

1. Introduction

A model of parallel computation represents an abstract machine designed to separate the concerns of program development from those of effective parallel execution. In fact, a model acts as a stable target for the software development process. Software developers can assume the existence of a stable abstract machine, so they can design software for it, without being concerned about architecture issues and developments. At the same time, a model represents a clear starting point for the implementation effort (design environment, compiler, run-time system) directed at each parallel architecture [9].

Models exist at different levels of abstraction. However, a parallel computation model, to be useful, should mainly be easy to program, expressive, architecture independent and efficiently implementable. These requirements are quite demanding and some of them are in tension with each others. For this reason very often people prefer to define low-level models that are efficiently implementable but make it hard to design and implement parallel software. To solve this problem it is necessary to find the best trade-off between expressiveness and efficiency defining models that are both at high level and might be implemented efficiently.

Today available parallel computing systems can be exploited to efficiently support applications in several application areas. However, the lack of efficiently implemented high-level languages, tools, and development environments does not allow to program parallel algorithms that are portable, efficient and expressive.

According to the Skillicorn's classification [8] the *restricted-computation structures* represent one of the most important models of parallel processing. The interest for this model is due to the possibility to restrict the form of computations so as to restrict communication volume achieving high performance. This allows to offer a user a structured model 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 model [11] represent a significant example of restricted-computation models that are used to design 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 which 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 in the previous time step 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 the 8 nearest neighbor cells. In the three dimensional case up to 26 neighbors can be taken in consideration. The states of each cell composing an automaton are updated synchronously in parallel. The global behavior of the system is determined by the evolution of the states of all cells as a result of multiple interactions.

In our approach a parallel cellular algorithm is composed of all the transition functions of cells that compose the lattice. Each transition function generally contains a same local rule, but it is also possible to define some cells with different state transition functions (*inhomogeneous* cellular automata). Differently from early cellular approaches where cell state is defined as a single or a set of bits, and for extending the range of applications to be programmed by cellular algorithms, we define the state of a cell as a set of typed substates that can be *shorts*, *integers*, *floats* and *doubles*. Further, we introduce a *logic neighborhood* that inside the same radius may represent a wide range of different neighborhoods also time-dependent.

These features have been implemented in a high-level language, called CARPET (CellulAR Programming EnvironmenT), that allows to design cellular algorithms. In particular, the CARPET language has been used for programming cellular algorithms in the CAMEL (Cellular Automata environMent for systEms ModeLing) [1] parallel environment. CAMEL is a software environment designed to support the parallel execution of cellular algorithms, the visualization of the results, and the monitoring of the program execution. The parallel execution of cellular algorithms is implemented by the parallel execution of the state transition function of each cell in a SPMD fashion. It offers the computing power of a highly parallel computer, hiding the architecture issues from a user. By CARPET a user might write cellular programs to describe the actions of thousands of simple active agents interacting locally, then the CAMEL system executes in parallel those actions 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 [5], CARP [6], CEPROL [7] have been defined. However none of those contains all the features of CARPET neither a parallel run-time support has been implemented for them. Reference [12] surveys and compares CARPET and some of those languages.

In this paper we briefly introduce the CAMEL system because it is both the development environment and the parallel run-rime system of CARPET. Then we discuss the main features of the CARPET language. Finally we give some performance figures to show the speed-up of CARPET programs.

2. Overview of CAMEL

The CAMEL system is a parallel environment based on the cellular automata model for developing scientific and engineering applications [2]. CAMEL has been implemented on a parallel computer composed of a mesh of Transputers connected to a host node. The current implementation of CAMEL does not limit the number of Transputers which can compose the parallel computer, so no changes should be necessary in the software of CAMEL whether a very large number of Transputer should be used. Moreover, CAMEL has been designed to be ported on other MIMD distributed-memory platforms.

CAMEL is both the parallel run-rime system of the CARPET language and a development environment for editing, compiling, configuring, executing, monitoring and visualizing the output of CARPET programs. In all these operations a user is supported by the CAMEL user interface (UI) that by pop-up menus assists him in all the software development process.

The CAMEL run-time system is composed by a set of *macrocell* processes each running on a single processing element of the parallel machine and by a *controller* process running on a master processor. Each *macrocell* process implements a CA portion composed of several elementary cells. It makes use of a communication system which handles the data exchange among cells and of a load balancing algorithm that balances the mapping of the CA portions on the processing elements. Reference [1] gives a detailed description of the load balancing algorithm.

Using this parallel architecture, CAMEL allows the parallel execution of the transition function of cells. Besides the UI the CAMEL system offers the Graphical Interface (GI) to rapidly and interactively explore and analyze very large amounts of scientific data gathered during the execution of computer simulations.

Furthermore, a tool called *IVT* (*Interactive Visualization Tool*), designed by MATLAB, has been added to CAMEL to improve data visualization. Utilizing data computed by simulation, *IVT* provides a variety of functions and services, including 2 and 3-dimensional graphical displays of data, hard copy of graphical displays and text, interactive color manipulation, animation creation and display, rotation of the images, saving of data in files according to different data formats.

3. The CARPET language

The CARPET language is a programming model that allows the definition of cellular algorithms. CARPET is a high-level language based on C with additional constructs to describe the rules 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 set of typed substates each one by a user-defined type, and the simple definition of complex neighborhoods (e.g., hexagonal, Margolus, etc.), that can be also time dependent, in a *n*-dimensional discrete Cartesian space.

Using CARPET a wide variety of cellular algorithms can be designed in a simple but very expressive way. The language utilizes control structures, types, operators and expressions of the C language. However it is enhanced by a declaration part that allows to specify the dimensions of the automaton, the radius of the neighborhood, the type of the neighborhood, and to describe the state of a cell as a set of typed substates that can be: *shorts*, *integers*, *floats* and *doubles*. Furthermore, a set of global parameters describe the global characteristics of the system (e.g., the permeability of a soil).

Special constructs allow at each iteration of the program execution to modify the values of the substates of a cell and define a set of cells (e.g., those of the border) with a different transition function. This last characteristic is very interesting because it simplifies the modeling phase of a system that can be represented by a network of cellular automata each describing one of the components in which the model has been divided to capture the different aspects of a phenomenon.

The language does not provide statements to configure the automata, to visualize the cell values or to define data channels that can connect the cells according to different topologies. The configuration of cellular automata is defined by the UI of the run-time system (i.e., the CAMEL environment). The UI allows, by menu pops, to define the size of the cellular automata, the number of the processors onto which the automata must be executed, and to choose the colors to be assigned to the cell substates to support the graphical visualization of their values.

The exclusion from the language of constructs for configuration and visualization of the data allows to execute the same CARPET program with different configurations. Further, it is possible to change from time to time the size of the automaton and/or the number of the nodes onto which the automaton should be executed. Finally, this approach allows to select the more suitable range of the colors for the visualization of data.

The execution of a program with different configurations allows to evaluate a model using various resolution obtained changing the size of the cell. This allows also to measure the scalability and the efficiency of the system.

The structure of a CARPET program is similar to that of a C program. A program is composed by a *declaration part* that appear only once in the program and must precede any statement (except those of C pre-processor) and by a *body program*. The *body program* has the usual C statements, without I/O instructions, and a set of special constructs to update the state of a cell. The body program is executed iteratively for a number of steps that a user can select by the UI. CARPET allows to use C functions or procedures to improve the structure of the programs.

3.1. Declaration part

The declaration part describes the dimensions of an automaton, the radius of the neighborhood, the state of cells, the cells belonging to the neighborhood and the global parameters. These declaration are contained inside of the `cadef` section.

```
cadef
 {
   dimension n;
   radius r;
   state { type_specifier substate_name;
           type_specifier substate_name;
            ....
         }
   neighbor id[n] {[xval, yval, zval] id,
                     ...
                    [xval, yval, zval] id };
   parameter {id value, id value, .....};
 }
```

3.1.1. Dimension

This definition allows to specify by a numeric literal the number of dimensions of an automaton, in a discrete Cartesian space. For example:

```
dimension 2 ;
```

defines a two-dimensional automaton.

The maximum number of dimensions allowed in the current implementation is 3. Each dimension is wrap around, e.g., a two-dimensional lattice forms a torus. Border functions can be used to disable the wrap-around.

3.1.2. Radius

`Radius` defines a numeric value that specifies the radius of the neighborhood of a cell. This value is strictly connected with the `dimension` definition. For example, in a 2-dimension automaton defining the radius equal to 1 the number of the neighbors can be up to 8. In the three dimensional case with radius equal to 1 the number of the neighbors can be up to 26.

The next example defines a radius equal to 2:

```
radius 2 ;
```

Our implementation supports a radius equal to 1 for three dimensional lattices, up to 2 for two dimensional lattices, and up to 50 for one dimensional lattices.

3.1.3. State

In CARPET the state of a cell is composed of a set of typed substates, unlike classical cellular automata where the state is represented by a few bits. The types of substates are: *shorts* (16 bits integers), *integers*, *floats* (reals), and *doubles* (64 bits reals).

By typification of substates, CARPET allows to extend the range of the applications that can be coded by cellular algorithms simplifying the writing of the programs and improving their readability.

Most systems and languages such as CELLANG, define the cell substates only as *integers*. In this case, for example, if a user must store a real value in a substate then 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 him a high level abstraction to define the cell state. The set of substates is defined through the **state** declaration. A type specifier must be declared for each substate. In the following example the state is constituted of three substates (a short and two floats).

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

The predefined variable **cell** refers the current cell in the n-dimensional space under consideration. The different substates can be referred appending to the reserved word **cell** the name of the substate by the underscore symbol '_'. For example, **cell_speed** refers the **speed** substate of the previous example.

3.1.4. Neighbor

As mentioned before, through the radius it is possible to define the maximum number of cells which might compose the neighborhood of a cell and that can be accessed to read their state.

For accessing a substate of a neighbor cell is needed to specify the indexing of the cell relative to the current cell. Relative indexing is explicated by a number of indexes equal to the dimension of the automata enclosed in square brackets and separated by commas. The figure 1 shows, for a two dimensional square automaton with radius equal to 1, the indexing of a cell relative to the current cell indexed with [0,0]. For example, the cell **N** situated at north is indexed with [0,-1], having used in the implementation the reference system indicated in figure 1.

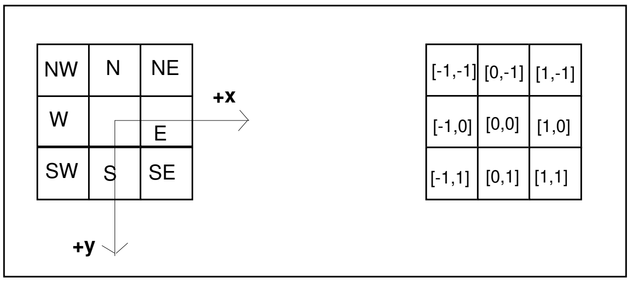

Figure 1. Relative indexing of a cell with respect to the cell [0,0].

CARPET generalizes the concept of neighborhood and allows a user to define by the **neighbor** declaration a *logic neighborhood* that inside the same radius may represent a wide range of different neighborhoods. Neighborhoods can be asymmetrical or have any other special topological properties (e.g., hexagonal).

The neighborhood is identified by the name of a vector with dimension equal to the number of elements composing the logic neighborhood. The elements of the logic neighborhood are put in round brackets and are separated by commas. Each element is described by relative indexing. Furthermore, to each of these elements it can be associated a name that can be used as an alias in referring to the neighbor cell. The von Neumann neighborhood can be defined as follows:

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

The code in figure 2 shows how through the **Neumann** vector it is easy to access to the **green** substate. This access way simplifies to access the substates using the **for** statement. Further, the **green** substate of the cell located at West can be accessed by **West_green** or **Neumann_green[1]**.

3.1.5. The step variable

By the predefined variable **step**, CARPET allows to know the number of iterations that have been executed. **Step** is updated automatically by the system. Initially the value of **step** is 0 and it is incremented by 1 each time that the state of all the cells of the automaton have been updated. This feature allows also to implement neighborhoods which are time dependent.

The **step** variable allows also to change dynamically the values of the substates dependent upon the iterations. A system characterized by several temporal phases can be described using this feature in a transition function. For example, the first iteration can be used to set up some parameters, the second iteration can initialize some substates and the next iterations can calculate, for a defined number of steps, the transition function describing the phenomenon.

```
cadef
{
 dimension 2;
 radius 1;
 state (short red, green, bleu);
 neighbor Neumann[4]([0,-1] North,
       [-1,0] West,[0,1] South,[1,0] East);
 }

  short sum, i;
   ....
   for(i=0; i < 3; i++)
   sum = sum + Neumann_green[i];
   ....
```

Figure 2. Referencing the **green** substate of a neighbor cell.

3.1.6. Global parameters

In modeling a complex system it is often necessary to describe some global features of the system. CARPET allows to define global parameters and to initialize them to specific values. Global parameters can be defined by the **parameter** declaration.

The type of parameters must be **float**, by default its value is zero. The following example shows the use of two global parameters **adherence** and **permeab** (permeability) initialized to 0.5 and 10.0.

```
parameter (adherence 0.5, permeab 10.0);
```

The value of a global parameter is the same in each cell of the automaton. For this reason, the value of each parameter cannot be changed in the code of the cell transition function. But, to assure that it will be updated for all the cells in the lattice, it can only be modified by the UI during the automaton execution.

3.2. Statements

To guarantee the semantics of cell state updating in cellular automata theory the value of one of the substates of a cell can be modified only by the **update** construct. After an **update** execution the value of the substate, in the current iteration, is unchanged. The new value does take effect at the beginning of the next iteration.

For this reason, the updating of the value of a substate cannot be performed by a direct assignment. The output of the program will be wrong if a value is assigned to a substate without the **update** statement. For example, the function:

```
update (cell_temp,45);
```

assigns to the **temp** substate the value 45. This value will be really available only in the next iteration.

Input and output of a CARPET program can be performed through files o by the *edit* function of the UI. A file can hold the values of one substate of all cells, these values can be loaded at the step 0 to initialize the automaton. The substate values can be the result of a previous simulation or they can be generated by a C program. In fact, the output of a CARPET program can be used as input to initialize another automaton because the format of the input and output is identical.

In regular intervals the output of a CARPET program can automatically be saved in a file to calculate global statistical functions (i.e. histogram, etc.) or to be post-processed by a visualization tool.

A user can use additional substates to store values that indicate statistical proprieties of a variable (i.e., mean values) or to hold a history of a substate. For instance, the average of the **temp** substate can be calculated and stored as shown in figure 3. Notice that the predefined **step** variable indicates the number of iterations that have been executed.

```
cadef
{
   dimension 2
   radius 1
   state (float temp, mean);
  ....
}
  ....
  avgtemp = (cell_mean + cell_temp )/step;
  update(cell_mean, avgtemp);

  .....
```

Figure 3. Evaluation of the average of the temp substate.

3.2.1. Special operations

Generally, the rules defined in CARPET are *deterministic,* i.e., the new state of a cell is uniquely determined by the current state of its neighbors: from the same initial conditions one invariably obtains the same evolution. However, CARPET offers the possibility to define non deterministic rules by the use of a *random* function.

Further, CARPET allows to define cells with different transition functions by means of the **GetX, GetY, GetZ** operations that return the value of the coordinates X, Y, and Z of the cell in the automaton. Using those functions it is possible to specify a different transition function for a single cell. Varying only a coordinate it will be possible to associate the same transition function to all cells belonging to the same row or column. The example in figure 4 shows how to assign a different transition function for the cell having coordinates (5,8).

```
cadef
{
   dimension 2;
   radius 1;
}
  ......
 Xpos = GetX;
 Ypos = GetY;
 if (Xpos == 5 && Ypos == 8)
    func_trans_1();
  else
    func_trans_2();
  ......
```

Figure 4. Definition of a different transition function for the cell (5,8).

4. A simple example

The example in figure 5 shows how the CARPET language can be used to implement a simple simulation of the propagation of a forest fire.

In this example each cell represents a portion of the land. Cells in the lattice can have the values included between '0' and '2'. The ground is represented by '0' value, the fire is represented by '1' value and the tree is represented by '2' value. Fire spreads from a cell which is on fire to a von Neumann neighbor that is treed, but not on fire.

This simple example shows as using a high-level language designed for programming cellular algorithms may strongly simplify the algorithms design process and reduce the program code.

5. An advanced example

In this section we show how by CARPET can be designed a cellular algorithm that implements gas diffusion simulation using the Margolus neighborhood [10].

The Margolus neighborhood is a complex time-dependent neighborhood defined as follows:

1. the array of cells is partitioned into a finite, disjoint and uniformly arranged collection of blocks having 2×2 size.
2. the same transition function is applied to every block, rather than to a single cell as in an ordinary cellular automaton. The blocks do not overlap and no information is exchanged between adjacent blocks.
3. two partitions are used, as showed in figure 6, at alternative times. At each step, there is not any overlap between the blocks used at one step and those used at the next one.

```
#define ground 0
#define fire 1
#define tree 2
cadef
{
 dimension 2;   /*bidimensional lattice */
 radius 1;
 state (short land);
 neighbor cross[4] ([0,-1]North,[-1,0]West,
                    [0,1]South,[1,0]East);
}
{
if (cell_land == fire ||
   (cell_land == tree &&
   (North_land==fire || South_land==fire ||
    East_land==fire || West_land==fire)))
update(cell_land, cell_land - 1) ;
}
```

Figure 5. The forest fire simulation written in CARPET.

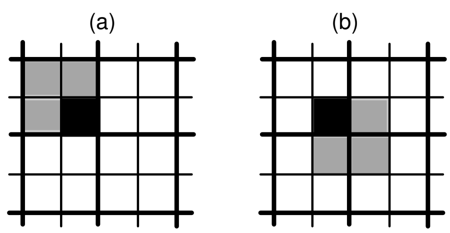

Figure 6. The two partitions for Margolus neighborhood. The black cell will have a different neighborhood considering from time to time the partition (a) (thick lines), or the partition (b) (thin lines).

```
cadef
{
    dimension 2;
    radius 1;
    state (short which, rand, gas);
    neighbor Margolus[9]([1,0]East,[1,1]SE,[0,1]South,[-1,1]SO,
                [-1,0]West,[-1,-1]NW,[0,-1]North,[1,-1] NE,[1,0] East);
}
 int i; short temp, temprand;
{
 if((cell_which == 0 && step %2 == 1)||(cell_which == 3 && step % 2 == 0))
 { temprand = 0;
      for(i=0; i < 3; i++)
      temprand = temprand + Margolus_rand[i];
      temprand = temprand + cell_rand;
      if (temprand % 2 == 1)
        update(cell_gas, South_gas);
      else
        update(cell_gas, East_gas);
 } else
 if((cell_which == 1 && step % 2 == 1)||(cell_which == 2 && step % 2 == 0))
  { temprand = 0;
      for(i=2; i < 5; i++)
      temprand = temprand + Margolus_rand[i];
      temprand = temprand + cell_rand;
      if (temprand % 2 == 1)
        update(cell_gas, West_gas);
      else
        update(cell_gas, South_gas);
  } else
 if((cell_which == 3 && step %2 == 1)||(cell_which == 0 && step % 2 == 0))
    { temprand = 0;
      for(i=4; i < 7; i++)
      temprand = temprand + Margolus_rand[i];
      temprand = temprand + cell_rand;
      if (temprand % 2 == 1)
         update(cell_gas, North_gas);
      else
         update(cell_gas, West_gas);
    } else
      {  temprand = 0;
         for (i=6; i < 9; i++)
         temprand= temprand + Margolus_rand[i];
         temprand = temprand + cell_rand;
         if (temprand % 2 == 1)
           update(cell_gas, East_gas);
         else
           update(cell_gas, North_gas);
      }
 temp = (cell_rand + East_rand + North_rand + West_rand + South_rand ) % 2
 update(cell_rand, temp);
}
```

Figure 7. Gas diffusion using a Margolus neighborhood.

Gas diffusion can be simulated using the Margolus neighborhood. The program showed in figure 7, based on the version described in [10], simulates the diffusion of a gas implementing a random walk for each particle/molecule of gas. The random walk can be implemented using the two partitions defined in the Margolus neighborhood at alternate times and having each block of cells randomly rotate their gas particles either clockwise or counter-clockwise. The Margolus neighborhood is implemented in CARPET using the `step` variable and the `which` substate that allows to distinguish between one partition and the other. The `which` substate is initialized alternating rows of 0s and 1s with rows of 2s and 3s indicating the cells relative position in a block. The figure 8 shows a portion of the automaton with `which` values and the Margolus neighborhood.

0	1	0	1	0	1	0
2	3	2	3	2	3	2
0	1	0	1	0	1	0
2	3	2	3	2	3	2

Figure 8. Initialization of the **which** substate with the Margolus neighborhood.

The neighborhood is defined in the program in figure 7 by the `Margolus` vector that acts as an alias for a group of neighbor cells with radius equal to 1. Each four-cell block has the same random number. This is done by maintaining in each cell a two-values (0 or 1) `rand` substate.

By using `step` and the value of the `which` substate, a random number shared by each cell of the block can be calculated. This random number is used to determine whether the particles of gas, denoted by the value of the gas fields, are rotated clockwise or counter-clockwise. To maintain an ever changing `rand` substate for each cell, a new value is calculated using the von Neumann neighborhood implicitly defined in the `Margolus` vector.

This algorithm that by CARPET has been coded in a program composed of about 50 lines would require several hundred lines of code if a language without high-level constructs for defining cellular data structures and programming cellular algorithms was used.

6. Performance

CARPET has been used to implement complex cellular programs to solve real problems in areas such as fluid dynamics, traffic planning, image processing, and genetic algorithms [2]. In particular, CARPET has been used in the CABOTO project, funded by the PCI ESPRIT programme, to implement parallel CA models for the simulation of *bioremediation* of contaminated soils [3].

In this section we present the performance results of the CARPET program that simulates a soil bioremediation model which includes water flow modeling, phenol contamination and the bioremediation event simulation. In this application the fluid flow with diffusion-transport of contamination agents inside the soil and their mutual influences are viewed as a dynamic system based on local interactions with discrete time and space, where the space is represented by cubic cells. Each cell is characterized by specific values (the state) of selected physical parameters, representing physical-chemical specifications, relevant to the evolution of the phenomenon. Here we show the program speed-up and efficiency using different configurations (grid sizes) of the automaton.

In the program execution, 100 steps simulate 50 minutes of the real phenomenon. Table 1 shows the speed-up measures of the parallel model implementation using different grid sizes on 2, 4, 8, 16, and 32 processors. These measures show how much faster the simulation runs on a parallel computer increasing the number of processing elements (PEs).

	Grid sizes of the cellular automaton			
PEs	**32x15x11**	**64x15x11**	**96x15x11**	**128x15x11**
1	1	1	1	1
2	1.967	1.980	1.984	1.977
4	3.830	3.908	3.932	3.930
8	7.441	7.717	7.800	7.812
16	14.119	14.986	15.328	15.421
32	26.204	28.461	29.527	29.960

Table 1. Program speed-up.

Table 2 shows the efficiency measures of the parallel model implementation using different grid sizes on 2, 4, 8, 16, and 32 PEs. These measures show how each single processor is efficiently used during the execution of the simulation.

	Different grid sizes of the cellular automaton			
PEs	**32x15x11**	**64x15x11**	**96x15x11**	**128x15x11**
1	1	1	1	1
2	0.983	0.990	0.992	0.989
4	0.957	0.977	0.983	0.982
8	0.930	0.964	0.975	0.976
16	0.882	0.936	0.958	0.964
32	0.818	0.889	0.923	0.936

Table 2. Efficiency measures.

7. Conclusions

This paper presented the main features of the CARPET programming language and discussed its use for designing cellular algorithms. Further, the paper showed performance figures of a parallel implementation of CAMEL on a distributed memory MIMD computer. This implementation is based on the CAMEL system that represents the parallel run-time support for the CARPET language. However, should be mentioned that the CAMEL system can be replaced with a different run-time support without affecting the CARPET constructs and semantics.

Currently we are working to redesign the CARPET run-time support for implementing it using the message passing interface (MPI) standard. This work will permit to port CARPET on several parallel hardware platforms such as the IBM SP2, the Meiko CS2, the CRAY T3D, and workstation clusters where the MPI package is available.

The CARPET approach is quite different from that followed in the implementation of early cellular processing systems where low-level languages are used to implement cellular algorithms. These languages make difficult to implement, read and port cellular algorithms. Further, really useful simulations are very complex to be programmed by such systems because of they poor expressiveness.

Our experience during the design, implementation and use of the CARPET language showed us that high-level languages are very useful for the development of parallel algorithms for solving real complex problems in several application areas, and in particular in science and engineering. According to this approach, very complex simulations such as fluid flow modeling, soil bioremediation, and freeway traffic flow simulation might be implemented by a few hundreds lines of code.

Finally, from the parallel programming point-of-view the CARPET language represents a good trade-off between expressiveness and efficiency defining a programming model that it is both at high level and at the same time can be implemented efficiently on parallel computers. In our opinion high-level languages like CARPET will allow to enlarge the use of the cellular automata model in solving complex problems preserving high performance and expressiveness, and may contribute to enlarge the community of users of parallel computers.

References

[1] Cannataro M., Di Gregorio S., Rongo R., Spataro W., Spezzano G., and Talia D., “A Parallel Cellular Automata Environment on Multicomputers for Computational Science” , *Parallel Computing,* North-Holland, Amsterdam, 1995, vol. 21, pp. 803-824.

[2] Di Gregorio S., Rongo R., Spataro W., Spezzano G., and Talia D., “A Parallel Cellular Tool for Interactive Modeling and Simulation”, *IEEE Computational Science & Engineering,* IEEE CS Press, 1996, vol. 3, pp. 33-43.

[3] Di Gregorio S., Rongo R., Spataro W., Spezzano G., and Talia D., “A Parallel Cellular Simulator for Bioremediation of Contamined Soils”, in *Development and Applications of Computer techniques to Environmental Studies VI*, Computational Mechanics Publ., Southampton, 1996, pp. 685-695.

[4] Eckart J. D., “Cellang 2.0: Reference Manual”, *ACM Sigplan Notices* 1992, vol. 27, no. 8, pp. 107-112

[5] Hochberger C. and Hoffmann R., “CDL - a Language for Cellular Processing, *Proc. 2nd Intern. Conference on Massively Parallel Computing Systems*, IEEE CS Press, 1996.

[6] Junger G., “Cellular Automaton Tool User Manual”, GMD, Sankt Augustin, Germany, 1994.

[7] Seutter F., “CEPROL a Cellular Programming Language”, *Parallel Computing,* North-Holland, Amsterdam, 1985, vol. 2, pp. 327-333.

[8] Skillicorn D. B., “Models for Practical Parallel Computation”, *Int. Journal of Parallel Programming*, 1991, vol. 20, no. 2, pp. 133-158.

[9] Skillicorn D. B. and Talia D., “Models and Languages for Parallel Computation”, submitted to *ACM Computing Survey*, 1996.

[10] Toffoli T. and Margolus N., *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, 1986.

[11] von Neumann J., “Theory of Self-Reproducing Automata”, *Comm. in Mathematical Physics,* 1985, vol., 96, pp. 15-57.

[12] Worsch T., “Programming Environments for Cellular Automata”, *Proceedings of the 2nd Conference ACRI '96*, Springer-Verlag, Workshop in Computing series, London, 1996.