CARPET: A Programming Language for Parallel Cellular Processing

Giandomenico Spezzano, Domenico Talia
CRAI
Località S. Stefano, Rende (CS), Italy

Abstract

In this paper we describe CARPET, a parallel programming language based on the cellular automata model. CARPET is the language used for programming cellular algorithms in the CAMEL environment. CAMEL is an environment designed to support the development of high performance applications in science and engineering. It offers the computing power of a highly parallel computer, hiding the architecture issues from a user. The system can be used both as a tool to model dynamic complex phenomena and as a computational model for parallel processing. By CARPET a user might write programs to describe the actions of thousands of simple active agents interacting locally, then the CAMEL system allows a user to observe the global complex evolution that arises from all the local interactions.

1 Introduction

Massively parallel computing can be very useful to engineers and scientists. However, the lack of models and tools that simply and automatically allow the exploitation of parallelism in complex engineering and scientific problems prevents a widespread use of this technology by industry. For a large class of complex simulation and modelling problems, the main issues rising from their complexity can be faced by using an appropriate computational model and implementing it on a commercial parallel computer.

This approach was taken in the development of CAMEL *(Cellular Automata environMent for systEms ModeLing*), a parallel environment for the execution of cellular automata models of complex systems [1]. CAMEL allows by the CARPET language the programming of cellular automata algorithms. Further it provides a simple and user friendly interface that allows on-going monitoring and interactive simulation.

Cellular Automata (CA) represent both a powerful tool for modelling complex dynamic problems and an abstract parallel computing model. CA capture the peculiar features of systems that evolve exclusively according to local interactions of their basic components and guarantee computational universality.

A CA can be represented as a *n*-dimensional lattice, partitioned into cells of uniform size, each one embedding an identical elementary automaton. Input for each elementary automaton or cell at time *t* is given by the states of the neighbouring cells at time *t-1*. The states of the cells are updated in parallel according to a local rule called *transition function*. According to this local rule, 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 neighbours at the previous step. Therefore, the transition function plays in cellular automata an analogue role to that of the evolution equation in classical dynamic models.

CA provide a global framework for the description dynamic phenomena in complex systems which it is based on the use of discrete time, discrete space and a discrete set of state variable values [2]. Moreover, CA are intrinsically parallel, and they can be simulated onto parallel computers with high efficiency, as the communication flow between processors can be kept low. Application areas of CA are very broad, ranging from the simulation of artificial life, physical, biological, and chemical phenomena to the modelling of engineering problems in many fields such as road traffic, image processing, and science of materials.

Parallel computers represent the most natural architecture where CA environments might be implemented. In fact, whether 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 [3].

In the following sections of the paper are shortly described the main features of CARPET, a cellular language defined to program cellular algorithms in CAMEL.

2 The CAMEL architecture

According to the approach mentioned above, CRAI and University of Calabria, in a joint effort, developed CAMEL, a parallel software environment based on the cellular automata model [1]. CAMEL has been implemented on a parallel computer composed of a mesh of 32 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.

The 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 several elementary cells of the cellular automata, and makes use of a communication system which handles the data exchange among cells. All the *macrocells* of the lattice execute in parallel the same local rule which results in a global transformation of the whole lattice state.

CAMEL provides a user a User Interface (UI) to configure a program, to monitor the parameters of a system simulation and to dynamically change them at run time.

3 CARPET features

The approach taken in CARPET (*CellulAR Programming EnvironmenT*) is to make parallel computers available to application-oriented users hiding the implementation issues coming from their architectural complexity. By CARPET a user can program complex problems which may be represented as discrete across a square or hexagonal 2D and 3D lattice. CARPET is a high-level language related to C with some additional constructs to describe the rules of the transition function of a single cell of a cellular automaton.

CARPET implements a cellular automaton as a 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.

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 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 functions to modify the state of a cell and its neighbourhood. CARPET allows to use C functions or procedures to improve the structure of programs.

In the declaration section are included constructs that allow to specify the dimensions of the automaton (`dimension`), the radius of the neighbourhood (`radius`), the type of the neighbourhood (`neighbor`), and to describe the state of a cell (`state`) as a set of typed substates that can be: *shorts*, *integers*, *floats* and *doubles*.

The *dimension* declaration allows to specify by a numeric literal the number of dimensions of the automata in a discrete cartesian space. The maximum number of dimensions allowed in our implementation is three. Each dimension is wrap around, e.g. a two-dimensional lattice forms a torus. Border functions can be used to disable the wrap-around.

Radius defines the set of cells that can compose the neighbourhood of a cell. This value is strictly connected with the *dimension* declaration. In a 2 dimension automaton having *radius* equal to 1, the number of cells that can compose the neighbourhood is 8. In the three dimensional case with *radius* equal to 1 the number of the neighbours is 26.

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 typification of the substates allows to extend the range of the applications that can be coded in CARPET simplifying the writing the programs and improving their readability. Most systems and languages, as for example CELLANG [5] and StarLogo [6], 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 this procedures makes the program longer and difficult to read or to change. The CARPET language frees the user of this tedious task and offers him a high level abstraction to declare the state defined through the *state* declaration. All substates are contained in round brackets and separated by commas. A type specifier must be included for each substate. In the following example the state is constituted of three substates:

```
state(short temp, quote, float volume);
```

A substate of the current cell can be referenced by the predefined variable *cell_substate*. 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* function. After an *update* statement the value of

the substate, in the current iteration, is unchanged. The new value does not take effect until the beginning of the next iteration.

CARPET generalizes the concept of neighbourhood 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 neighbourhoods. Neighbourhoods can be asymmetrical or have any other special topological properties (e.g., hexagonal). The *neighbor* declaration assigns a name to specified neighbouring cells of the current cell and allows such to refer to the value of the substates of these identified cells by their name. Further, to *neighbor* it must be associated the name of a vector that has as dimension the number of elements composing the logic neighbourhood. The name of the vector can be used as an alias in referring to the neighbour cell. The von Neumann neighbourhood can be defined as follows:

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

Through the vector a substate can be referred as *identifier[i]_substate*. Using this referring way and the *for* statement it is simpler to write the programs.

CARPET allows to know the number of iterations that have been executed by the predefined variable *step*. *Step* is updated automatically by the system. Initially the value of *step* is 0 and it is incremented by 1 each time all the cells of the automata have been updated. This feature allows also to have neighbourhoods which are time dependent. *Step* 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 examples, the first iteration can be used to set-up of some parameters, the second iteration can initialise some substates and the next iterations can calculate, for a defined number of steps, the transition function describing the phenomenon.

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. 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 UI. The global parameters can be declared by the *parameter* section. To each global parameter is associated a name used to refer the parameter during the computation. The type of parameters must be *float*, by default its value is zero.

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, these values can be loaded at the *step* '0' to initialize the automaton. These 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.

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

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 a cellular automaton is defined by the UI of 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 colours 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.

4 Programming examples

The example in figure 1 shows how the CARPET constructs can be used to implement the *parity rule* program.

```
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); /* updating the
                        new state of cell*/
}
```

Figure 1. The parity rule game written in CARPET.

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.
Figure 2 shows the version of 3D life, that is based on work by Carter Bays from University of South Carolina. He has analysed a large variety of 3D versions of the game of life and has individuated *life 4555* and *life 5766* as the most important. Both the versions reproduce many characteristics of the original 2D life game.

```
#define dead 0
#define alive 1
cadef
{
  dimension 3;
  radius 1;
  state (short life);
  neighbor 3D_Life[26]
    ([1,0,0]E,[1,1,0]SE, [0,1,0]S,
     [-1,1,0]SW,[-1,0,0]W, [-1,-1,0]NW,
     [0,-1,0]N,[1,-1,0]NE,[1,0,-1]ED,
     [1,1,-1]SED,[0,1,-1]SD,[-1,1,-1]SWD,
     [-1,0,-1]WD,[-1,-1,-1]NWD,[0,-1,-1]ND,
     [1,-1.-1]NED,[1,0,1]EU,[1,1,1]SEU,
     [0,1,1]SU [-1,1,1]SWU,[-1,0,1]WU,
     [-1,-1,1]NWU, [0,-1,1]NU,[1,-1,1]NEU,
     [0,0,1]UP,[0,0,-1]DOWN);
}
int i;
short sum;
{
/* computing the number of alive neighbors
                  of cell (x,y,z) */
  for(i=0; i<26; i++)
     sum = 3D_Life[i] + sum;

/*computing the new state of cell (x,y,z)*/
  if (cell_life == dead)
   if (sum == 5)
      update(cell_life,alive);
    else
      update(cell_life,dead);
   if (cell_life == alive)
   if ((sum < 4) || (sum > 5))
      update(cell_life,dead);
     else
      update(cell_life,alive);
}
```

Figure 2. The *Life 4555* version of 3D life.

The first two numbers define the destiny of alive cells. The former indicates the minimum of living cells that a cell must have in its neighborhood to not die. The latter indicates the maximum of living cells to not perish from overpopulation. The third and fourth number regulate the destiny of dead cells. They respectively define the minimum and the maximum of adjacent alive cells that a dead cell must have to become alive.

5 Conclusion

In this paper we have described the CARPET language used for programming cellular algorithms in the CAMEL environment. The main features of the language are the possibility to describe the state of a cell as set of typed substates each one by a user-defined type, and the simple definition of complex neighborhood that can be also time dependent. CARPET has been used with success to implement different applications concern freeway traffic simulation, image processing and genetic algorithms.
Currently, CARPET is used in the CABOTO project within the PCI ESPRIT framework. The CABOTO main objective is the use of CA models for the *bioremediation* of contamined soils.

Acknowledgements

This research is partially supported by CEC ESPRIT P.C.I. contract n° 94529419370.

References

[1] 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,* North-Holland, Vol. 21, pp. 803-823, May 1995.

[2] Y. Feldman and E. Shapiro, "Spatial Machines: a More Realistic Approach to Parallel Computation", *Communications of the ACM*, Vol. 35, 1992, pp. 61-73.

[3] B. P. Brinch Hansen, "Parallel Cellular Automata: a Model for Computational Science", *Concurrency: Practice and Experience*, Wiley, Vol. 5, 1993, pp. 425-448.

[4] S. Di Gregorio, G. Spezzano, D. Talia, "Design and Specification of CAMEL Extensions", Deliverable D2 CABOTO Project, 1995.

[5] J. Dana Eckart, "Cellang: Language Reference Manual".

[6] M. Resnick, *Turtles, Termites, and Traffic Jam: Explorations in Massively Parallel Microworlds*, The MIT Press, Cambridge, Mass., 1994.