SMR/772 - 19

INTERNATIONAL WORKSHOP ON PARALLEL PROCESSING AND ITS APPLICATIONS IN PHYSICS, CHEMISTRY AND MATERIAL SCIENCE

5 - 23 September 1994

ADDITIONAL MATERIAL TO LECTURES PRESENTED BY

Domenico TALIA
CRAI
Localita' di Santo Stefano
Rende
87036 Cosenza
ITALY

TO APPEAR ON PARALLEL COMPUTING, NORTH-HOLLAND, VOL. 21, 1995.

A Parallel Cellular Automata Environment on Multicomputers for Computational Science

M. Cannataro*, S. Di Gregorio°, R. Rongo°,
W. Spataro°, G. Spezzano*, D. Talia*

* *CRAI, Località S. Stefano, 87036 Rende (CS), Italy*
° *Dept. of Mathematics, University of Calabria, 87036 Arcavacata (CS), Italy*
Email: `dot@crai.it` ; Fax: +39 984 446044

Abstract

This paper describes CAMEL *(Cellular Automata environMent for systEms modeLing*), a scalable software environment based on the cellular automata theory implemented on a Transputer-based parallel computer. Cellular automata were originally defined as a theory to model the basic mechanisms of dynamic systems, permitting a new approach which is in many cases simpler and more efficient than the traditional approach based on partial differential equations. Today, cellular automata become more attractive because they are suitable to be effectively and naturally implemented on parallel computers achieving high performance. CAMEL allows a user to program computational science applications exploiting the computing power offered by highly parallel computers in a transparent way. CAMEL implements a cellular automaton as a SPMD program. A load balancing strategy is used to minimize time costs in case of not uniform intervals for transition steps. In the paper the programming environment and the parallel architecture of CAMEL are presented and some experiments are discussed.

Keywords: Cellular automata, multicomputers, parallel processing, simulation, software tools, Transputer.

1. INTRODUCTION

In the past, the behavior of many complex phenomena was investigated only from a qualitative viewpoint since formal models, describing them, were so hard that the main (at that time) computational modality, represented by the integration of differential equations, was impracticable.

Consequently to the development of computer science in these last years, the applicability boundaries have been expanded considerably because of the continuous rise of computing power. At the same time, research in parallel computing have revealed the relevant potentialities of parallel computing models to represent a valid alternative to differential calculus in the description of complex phenomena.

Cellular Automata (CA) were one of the first parallel computing abstract models. They match the parallelism paradigm with the acentrism one and therefore became a powerful tool for modeling phenomena which can be formalized in acentric terms. Conceived by John von Neumann in the 1950's to investigate self-reproduction [21], CA have been mainly used for studying parallel computing methods and the formal properties to model systems [4, 13]. However, with the rapid advances in computational resources during the 1980's, CA have become increasingly utilized for more general computer simulation and modeling [18]. Recently, has been showed as the CA model can be effectively used both as a realistic approach to define abstract parallel machines [11] and as a programming methodology for computational science on parallel computers [2, 3].

CA capture the peculiar features of systems which may be seen to evolve according exclusively to local interactions of their constituent parts [4, 22], and guarantee computational universality [17]. Furthermore, applied aspects of modeling have been widely investigated from a theoretical viewpoint.

Applications of CA are very broad, ranging from the simulation of fluid dynamics, physical [18], chemical, and geological processes, e.g., rock fracturing [15] and lava flows [1], to image processing, biology, and weather modeling.

Parallel computers represent the natural architectures where CA environments might be implemented, however the considerable computing power necessary to the simulations forces a strong optimization of the solutions, since a single architecture cannot satisfy the requirements of different CA models in the same way.

This paper describes CAMEL (*Cellular Automata environMent for systEms modeLing*), a software environment based on the cellular automata model which has been implemented on a multicomputer [6]. CAMEL is a tool designed to support the development of high-performance applications in science and engineering. It offers the computing power of a parallel computer although hiding the architecture issues to a user. It provides an interface which allows a user to dynamically change the parameters of the application during its running, and a Graphical Interface which shows the output of the application. Moreover, the CAMEL architecture is suitable to be implemented on several distributed memory parallel computers such as the Intel iPSC, the Ncube, the CRAY T3D, and the Connection Machine 5.

CAMEL has been successfully used for many problems such as the simulation of the lava flows of the last Etnean eruption and other fluid flow models, for image processing, and combustion process modeling. The results achieved in terms of thoroughness of simulation and execution speedup show that CAMEL might be used for simulation and modeling of real systems in many areas in a simple way and achieving high performance.

The main goal of this paper is to present CAMEL as a practical and general tool based on the cellular automata theory which allows to develop applications in the computational science area. The remainder of the paper is organized as follows. Section 2 gives a formal definition of CA. Sections 3 presents a short analysis of the use of parallel computers for the implementation of CA. The CAMEL architecture, its programming model, and the load balancing strategy used in it are described in Section 4. Section 5 gives experimental results. Section 6 compares the features of CAMEL to other CA environments. Section 7 describes a significant application of the system. Finally, Section 8 concludes the paper and presents directions for future work.

2. CELLULAR AUTOMATA

Several definitions of CA can be found in literature. In many cases one looks so dissimilar from the other that they seem to refer distinct objects. That is because it is more convenient to work with definitions adherent to the particular applications to be developed. We introduce a definition concerning the primal and main class of CA, i.e. the homogeneous CA, which are to our knowledge those more involved in systems modeling and simulation. The most cases of definitions can go back to that one by an extension of some property or by opportune transformations from a formal frame to another.

A homogeneous CA can be intuitively considered as a *d*-dimensional euclidean space, the cellular space, partitioned into cells of uniform size (i.e. partitioned with a square, cubic, or hypercubic tessellation), each one embedding an identical Moore elementary automaton (*ea*). Input for each *ea* is given by the states of the *ea* in the neighboring cells, where neighborhood conditions are determined by a pattern invariant in time and constant over all cells. At the time t=0, *ea* are in arbitrary states and the CA evolves changing the state of all *ea* simultaneously at discrete times, according to the transition function of the *ea*.

Formally a CA *A* is a quadruple $A=(E^d, X, S, \sigma)$, where:

- E^d is the set of cells identified by the points with integer coordinates in a *d*-dimensional Euclidean space;
- X, the neighborhood index, is a finite set of *d*-dimensional vectors, which defines the set $N(X,i)$ of neighbors of cell $i=<i_1, i_2,\ldots\ldots,i_d>$ as follows: let $X=\{\xi_0, \xi_1, \ldots., \xi_{m-1}\}$ with $m=\#X$, then $N(X,i)=\{i+\xi_0, i+\xi_1,\ldots\ldots,i+\xi_{m-1}\}$; ξ_0 is always the null vector, and identifies the cell itself.
- S is the finite set of state of the *ea*;
- $\sigma:S^m\rightarrow S$ is the deterministic transition function of the *ea*.

$C=\{c|c:E^d\rightarrow S\}$ is the set of possible state assignments to A and will be called the set of configurations, where $c(i)$ is the state of cell i. Let $c(N(X,i))$ be the ordered set of states of the neighborhood of i. Then the global transition function τ is defined by

$$\tau: C \rightarrow C \mapsto [\tau(c)](i) = \sigma(c(N(X,i)))$$

When τ is not determined through a constant σ and X, then the CA is not homogeneous.

A quiescent state s_q is such that $\sigma(s_q,s_q,\ldots\ldots,s_q)=s_q$; a configuration c is finite when $c(i)=s_q$ except for a finite number of cells.

When appropriate, the previous definition may be easily extended to different types of space, e.g., riemannian space, or different tessellations, e.g., hexagonal, triangular tessellation in a bidimensional space.

We introduce now extensions of the notion of quiescent state, relevant to the implementation of a CA environment.

In the following we will suppose that the *ea* could be endowed with a set of passive states $S_P \subset S$ such that

$$\forall s_0, s_1, \ldots\ldots, s_{m-1} \in S_P \quad \sigma(s_0, s_1, \ldots\ldots, s_{m-1}) = s_0;$$

s is an inert state iff

$$\forall s_1, s_2, \ldots\ldots, s_{m-1} \in S \quad \sigma(s, s_1, s_2, \ldots\ldots, s_{m-1}) = s;$$

S_I is the set of inert states.

The stationary region R_s of a configuration c is defined by

$$R_s = \{i \mid (c(i) \in S_I) \vee (j \in c(N(X,i)) \Rightarrow c(j) \in S_P)\}.$$

Cells with passive states don't change state until the neighborhood remains passive too, while cells with inert states never change state. Thus, it is important to develop methods to manage the stationary region in order to save precious computational power, even if the real computation synchronism of the cells would be lost in such a condition.

When CA are used in simulation of natural phenomena, the cell state often represents different characters of that portion of space corresponding to the cell. It could be useful to introduce the notion of substate, representing a single character, while the set of states is obtained by the cartesian product of the sets of substates.

Some properties which are relevant to our goal also concern the CA complexity trade-offs, which permit to reduce the cardinality of the neighborhood index: a CA $A_1=(E^d, X_1, S_1, \sigma_1)$ can always be simulated by a CA $A_2=(E^d, X_2, S_2, \sigma_2)$ where X_2 is the set of all d-dimensional vectors, whose components may be only 0 or 1 with $\#X_2=2^d$. An application of that property can be found for the unidimensional case in reference [9].

3. PARALLEL COMPUTERS FOR CELLULAR AUTOMATA

As discussed in the previous section, the CA model offers a new methodology for modeling and simulation of complex systems. In real applications which solve problems in the computational science area, a very large number of cells should be defined. 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 one after the other in a sequential way. Thus, sequential computers do not offer a practical support for the implementation of CA.

There are two possible alternatives which allow to achieve a better performance with respect to sequential computers in the implementation of CA. The first one is the design of special hardware devoted to the execution of CA. The second alternative is based on the design of a programming environment to be used on commercially-available parallel computers where the state of cells can be updated simultaneously.

CAM (Cellular Automata Machine) [18] is the most significant example of a specialized hardware which has been designed to run CA simulations. Although the CAM offers an high-level environment for programming CA and can run CA simulations in an efficient way, it is limited in the size of the automata which can be simulated and in the number of states per cell. Furthermore, it is a specialized machine which cannot be utilized as a general purpose computer.

On the other hand, highly parallel computers offer the most natural architecture for a CA machine. These systems are based on a number of interconnected processing elements (PEs) which perform a task concurrently.

CA are composed of a very large number of identical simple cells which interact only with their neighborhood. They can exploit a kind of parallelism which is called *data parallelism*. According to it, the cells are partitioned on the PEs and each PE executes the same operation on its partition on each step. The data parallelism is generally implemented on SIMD machines, but it can also be efficiently implemented on MIMD machines.

SIMD machines are well suited for CA where all the cells are active during all the time of the simulation. On the other hand, in the majority of real applications the SIMD approach does not achieve very high performance because is not possible to exploit the computing power offered by a SIMD machine when there exists a stationary region for which the PEs compute the transition function which does not change the state of cells. In fact, when a large number of cells are passive or inert many portions of an automaton do not contribute to the final global state and many PEs are not effectively utilized.

MIMD computers are more adaptable to implement CA in comparison with SIMD machines. On a multiprocessor, CA can be implemented mapping on each PE a process which updates a portion of cells. The problem of stationary regions is solved because each process can detect the passive or inert cells in its portion and does not compute their states. Using a shared memory all cells are available to each PE, so no communications of portion boundaries between PEs are needed. The main drawback of multiprocessor systems is the poor speedup they achieve when more than ten or so PEs are utilized. This effect is due to memory access contention which occurs in shared memory multiprocessors, where the bus is a bottleneck when a large number of PEs is used. As an example, in reference [19] is discussed an implementation of CA on a multiprocessor where, in many applications, the bus-memory bottleneck limits the number of PEs that can be utilized in a cost-effective manner to 10 or 12.

To avoid the bus-memory bottleneck, a multicomputer system composed of a large number of PEs cooperating by message passing might be used. This makes the CA environment a scalable system where no bottleneck will emerge when the number of processors is increased or when the size of an automaton is scaled up.

In a multicomputer data parallelism is exploited using the *Single Program Multiple Data* (SPMD) model. According to this model an application on N PEs is composed of N similar processes, each one mapped on a different PE that operates on a different set of data. For an effective implementation, data should be partitioned in such a way that communications locality is taken into account, and the computation load is shared among the PEs in a balanced way.

The SPMD model is well suited for the parallel implementation of CA environments exploiting a medium-grain parallelism. In fact, the grain size of SPMD processes is larger than the grain size of processes in SIMD machines. On each PE of a multicomputer a SPMD process executes the transition function for a partition of cells and cooperates with the neighbor PEs for the exchange of the states of cells which are on the border of a partition. Using this approach, multicomputer systems might be used for the implementations of cost-effective highly parallel CA.

Based on the earlier evaluations, we have chosen to implement a CA-based software environment for the simulation of complex systems on a multicomputer. The next section shows how CAMEL has been implemented on a Transputer-based multicomputer in a scalable way using the SPMD model together with an efficient load balancing strategy.

4. CAMEL

The architecture of CAMEL is composed of a set of *Macrocell* processes, each one running on a computing node of the parallel computer, and a *Controller* process running on a master node. One or more contiguous portions of the lattice which represents the system to be simulated is assigned to each *Macrocell* process. The set of the *Macrocell* processes implement a cellular automaton. The *Controller* implements the cooperation among the *Macrocell* processes, the file system, and the User Interface.

The User Interface of CAMEL allows the programmer to define the rules and the initial status of the cellular automata, and to interact with the system. Using the User Interface a programmer may change, during a simulation, some parameters such as, the output visualization interval, the values of the physical parameters of the simulated system, the status of the automata cells, and the size of the graphical output.

CAMEL has been implemented on a parallel computer composed of a mesh of 32 Transputers [12] 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 relevant changes should be necessary in the software of CAMEL whether a very large number of Transputer should be used.

Furthermore, as mentioned before, the CAMEL architecture is suitable to be implemented on others distributed memory parallel computers such as the Intel iPSC, the Ncube, the CRAY T3D, the CM-5, and the TI C40-based multicomputers.

In the following we discuss how CAMEL may be programmed and describe the architecture of the system.

4.1. Programming model

The development of programming methodologies for parallel computers is an essential issue for a large use of these systems. Generally the implementation of parallel applications is difficult because the computation must be divided over the processors in a balanced way taking into account the physical configuration and avoiding a large communication overhead.

The approach taken in CAMEL is to make parallel computers available to application-oriented users hiding the implementation issues coming from their architectural complexity. CAMEL allows the solution of complex problems which may be represented as discrete across a square or hexagonal grid or lattice. In such systems, the values of a specific point at time *t+1* is dependent upon the value at time *t* of its neighbor points on the basis of a state transition function.

Each system which can be specified using this approach can be modeled or simulated using CAMEL. CAMEL implements a cellular automaton as a SPMD program. CA are implemented as a number of processes each one mapped on a distinct PE which executes the same code on different data. According to this approach a user must specify the system he wants to simulate by a procedure written using a sequential language such as C, Pascal, or the sequential statements of Occam. In it are defined the macroscopic quantities of the system to be simulated, but nothing must be specified about the parallel execution. For example, in case of a fluid simulation, the user must encode the state transition function of a single cell in a procedure which will constitute the core of the processes running on each computing node of the parallel computer, and all the macroscopic features of the fluid such as density, velocity, temperature are expressed as parameters of the procedure.

In order to describe how CAMEL is programmed we show how a simple cellular automaton, the *game of Life*, is implemented in CAMEL. *Life* [8] simulates a population of interacting living organisms or cells in a 2-dimensional grid. Each cell can be in two states: *alive* (1) or *dead* (0). The cells change state, in parallel, using a transition function which depends on the 8 nearest neighbors of the cell (see figure 1) on the grid according to the following rules:

- a living cell can survive for the next generation if and only if it has exactly 2 or 3 living cells in its neighborhood. Otherwise, it dies;

• a dead cell will come to life in the next generation if and only if it has exactly 3 living cells in it neighborhood.

1	*2*	*3*
4	**c**	*5*
6	*7*	*8*

Figure 1. Neighbors of a cell *c*.

Life is implemented in CAMEL coding the two rules of the transition function in a procedure called `compute_state` and defining the global constants of the automaton. These constants describe the main features of the automaton as follows:

```
grid_width    IS  128     -- width of the cell grid
grid_height   IS  128     -- height of the cell grid
neigh_num     IS  8       -- number of neighbor cells except the cell itself
dead          IS  0
alive         IS  1 .
```

The schema of the procedure which represents the transition function of each cell is described in figure 2.

```
PROC compute_state ([NS][grid_width][grid_height] INT curr_state,
                   next_state, INT i, j, [NumOfFeat] REAL macro_feat)

  INT living_neigh

   -- compute the value living_neigh
   CASE
     living_neigh < 2
       next_state[0][i][j] := dead
     living_neigh = 2
       next_state[0][i][j] := curr_state[0][i][j]
     living_neigh = 3
       next_state[0][i][j] := alive
     living_neigh > 3
       next_state[0][i][j] := dead
```

Figure 2. The procedure `compute_state` for *Life*.

In figure 2 `NS` defines the number of substates. The parameters `i` and `j` indicate the position of a cell in the grid, the arrays `curr_state` and `next_state` are respectively the current and the next state of the automaton, and the vector `macro_feat` contains the macroscopic features of the system which is simulated. In this case, due to the simplicity of the example this parameter is not used.

After the procedure is linked to the CAMEL environment, its code is automatically mapped on each computing node and is executed in parallel to update the state of each cell. A user may then observe the system evolution through the Graphical Interface and eventually change the parameters of the simulation by the User Interface.

4.2. Architecture

The architecture of CAMEL is composed of three main components:

- the Master Node,
- the Graphic Node, and
- the Parallel Engine.

In particular, the Parallel Engine consists of a number of identical nodes on which the cellular automaton is executed. Figure 3 shows a scheme of the parallel architecture of CAMEL.

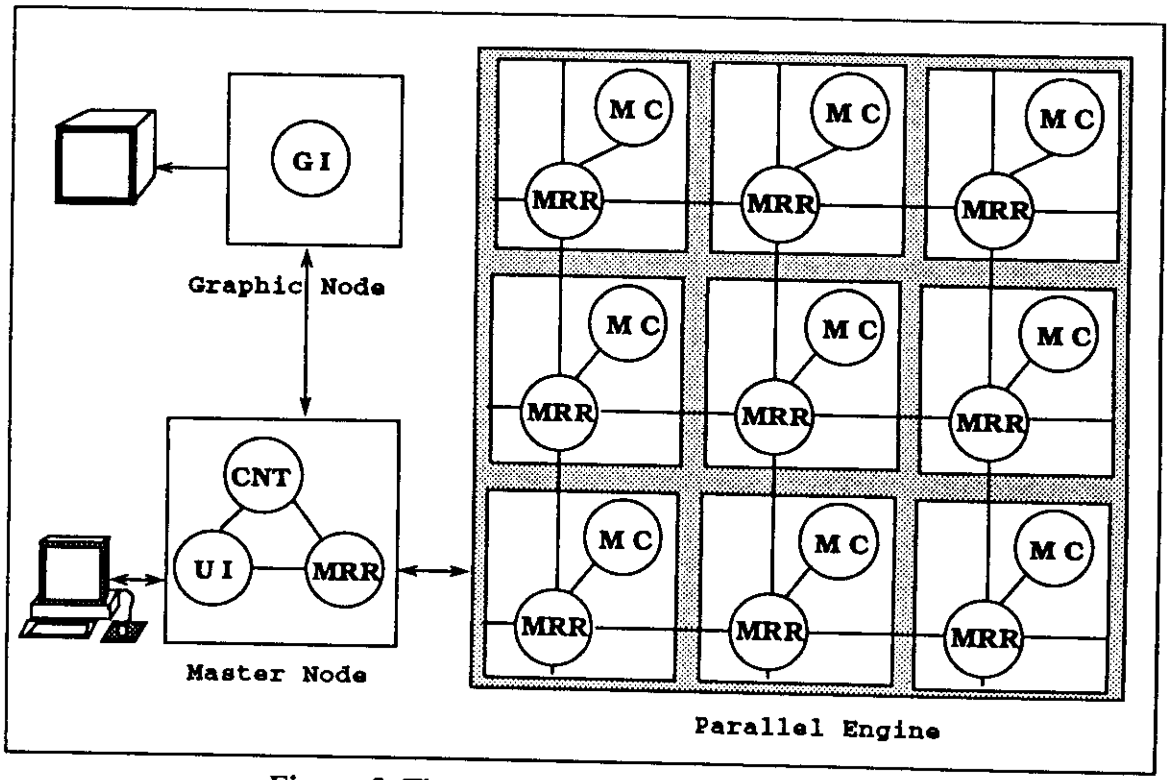

Figure 3. The parallel architecture of CAMEL.

The Master Node is implemented on a Transputer. On it are mapped the *Controller* (CNT), the *User Interface* (UI), and a *Multiple Ring Router* (MRR). The CNT provides a coordination among the *Macrocell* processes, the file system, and the User Interface. The main task of the CNT process is the synchronization of the *Macrocell* processes which are running on the nodes of the Parallel Engine. At the beginning, the CNT sends an *evolve* message to all the *Macrocells*. After that, the Macrocells perform a number of steps, then send the results to be displayed and wait for a signal from the CNT to initiate, in a synchronous way, the execution of the next step sequence.

The UI allows a user to monitor the parameters of a system simulation and to dynamically change them at run time. A detailed description of the UI is given in a next section.

The MRR implements a routing algorithm which provides deadlock-free interprocessor communication among the Transputers of the Parallel Engine and the Master Node. The routing algorithm has many positive characteristics including provable deadlock-freedom, guaranteed message arrival, and automatic local congestion reduction [5]. The collection of the MRR allow message communication between two *Macrocell* processes regardless of where they are physically located on the Parallel Engine and between a *Macrocell* process and the Master Node.

The Graphic Node is implemented on a graphic board. On it is mapped the *Graphical Interface* (GI). The GI displays on a video the state of a system (the set of the states of cells) which is solved by CAMEL. In order to perform this task the GI receives the cells states from the *Macrocells* and commands from the User Interface. Section 4.2.2 describes the main features of the GI.

On each Transputer which composes a node of the Parallel Engine runs a *Macrocell* (MC) process and a MRR. The MC is the heart of CAMEL. Each MC implements a cluster of elementary cells which constitute one partition of a cellular automaton. Each MC, for all the cells which compose the partition, executes the state transition function defined by a user. The whole set of the MCs implement a cellular automaton.

For each step or set of steps each MC receives from the CNT an *evolve* message. Then they compute the next state of the cells which belongs to their own partition. For this, each MC sends to its neighbors the states of cells which are on the borders of its partition and receives from them the borders it needs. Then it computes the state transition function for each cell and sends the result to the Master Node for the visualization. After sending the cell states a MC is ready to receive a new message from the CNT process.

4.2.1. User Interface

High level and user friendly tools allowing interaction with a user are a basic part of the system. The UI of CAMEL is composed by an editor and a command interpreter. It communicates with the host I/O subsystem, the Graphical Interface, the Controller and the MRR. The UI is menu based. The commands received by the User Interface are read by the editor, then translated by the interpreter in a set of requests/commands to the other modules of the system. The run-time system can be in two states: *executing* or *waiting* for a command. During an *executing* phase the only allowed interaction is a suspension request that changes the system state. During a *waiting* phase three menu are available: I/O, Setup and Presentation.

The Setup commands permit the tuning of the simulation. The I/O commands permit the interaction between CAMEL and the host I/O subsystem. The commands of the Presentation menu involve the way how the state information are displayed on the graphic video.

Finally, during the *executing* phase, the system provides a set of useful information such as the *displayed substates*, the *current iteration*, the *step of visualization*, and the *step of saving*.

4.2.2. Graphical Interface

The GI shows the output of a simulation in a graphic format. This module receives raw data from the *Macrocells* on the network, commands from the UI and then performs the action needed to the on-line display. The GI displays the state of cells. To each substate is associated a range of values to be displayed and a corresponding range of colors to be used.

The user can zoom a displayed image or its portions to obtain the best definition level. Besides, it is also possible to visualize more substates, overlapped on the same window, or in different windows. This provides a more complete system evolution representation, but requires an higher visualization time.

It is important to note that the execution of the GI occurs in parallel with the system execution phase. In fact, when it receives the system state to be displayed, the Controller starts the next execution phase. If the *visualization step* takes sufficient time, the waiting time between two visualizations may be reduced only to the execution time.

4.2.3. Load balancing

A major problem to be solved when programming parallel computers is the task distribution and load balancing over the computing nodes of a parallel machine. Load balancing algorithms allow an effective use of resources of the nodes, avoiding overloaded nodes while others are underloaded [7].

Let consider the state transition function for a cellular automaton. If the state of a cell remains constant when its neighborhood's state does not change, the cell will remain in a given state until a change occurs in one or more cells of its neighborhood. We defined as *passive* this cells, *active* otherwise. On each step the sets of active and passive cells may change. For a large class of problems, as in flow diffusion, the areas of active cells are restricted to one, or few domains. The system evolution expands or reduces such domains over the entire automaton, causing the cells on the frontier to become active or passive. From an operational point of view is useless to compute the next state of the passive cells until they become active.

For these systems a simple static cell partitioning may be devised dividing the grid of cells in a set of partitions, each one formed by adjacent cells, and assigning each of them to a processor of the system. Although this method involves a simple matching between the partitions and the processors, it often produces a non optimal task distribution. In fact, a considerable part of the automaton and of the parallel system might remain idle for a long time.

As an example, figure 4 shows an automaton with a domain of active cells, and a simple partition over four processors. The load distribution is very unbalanced, The processors P0, P1, P2, P3 execute respectively the 16%, 57%, 18%, and 9% of active cells.

On the other hand, a dynamic task distribution on each iteration should have an updated snapshot of the system from which it may find out the new *active* cells and distribute them over the processors, maintaining the load balanced. Obviously, this strategy requires a high communication overhead due to the high dynamic and complex communication patterns.

The major goal for the task distribution is the assignment of the active cells on all the nodes in a balanced way. Due to the dynamicity of the active domains this requirement can be only partially met. However, the partitioning should distribute a generic area of the automaton on a high number of processors, and guarantee a simple communication pattern between adjacent partitions.

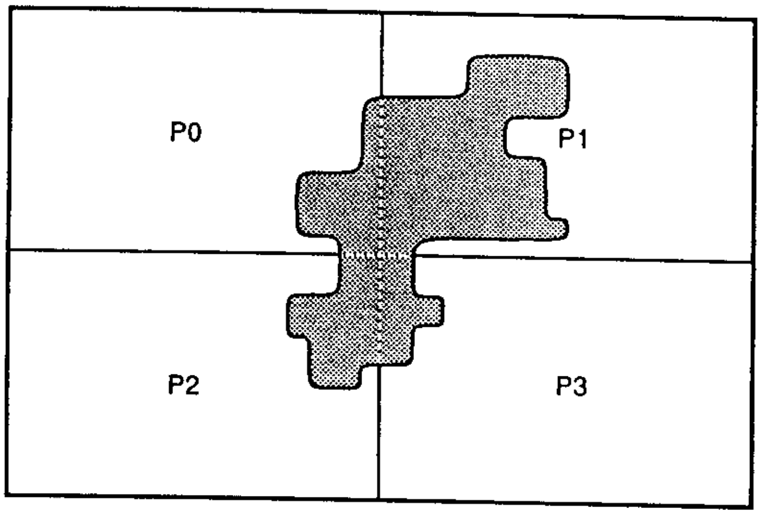

Figure 4. Simple task distribution on four processors.

The load balancing strategy defined in CAMEL is a trade-off between the static and the dynamic approaches. In fact, the cells partitioning is static, whereas the amount of cells mapped on each partition is dynamic. In CAMEL, the grid of cells is first divided in n vertical *folds*, each fold is then partitioned into N strips, where N is the number of processors of the multicomputer. The i-th strip of each fold is assigned to the generic processor P_i. A similar technique known as *scatter decomposition* is discussed in reference [14].

Figure 5 shows the domain of figure 4 partitioned in six folds ($n=6$) each one split in four strips ($N=4$). The resulting assignment to the four processors is shown in figure 6. According to this strategy, the percentages of active cells mapped on the processors are 27%, 24%, 21%, and 28%.

To avoid useless computation the user may change, at run time, the set of folds on which the state transition function must be applied. Each *Macrocell* process will compute only the strips of the specified folds.

The set of active fold will be augmented or restricted just before some cells become active or passive. The choice of the active folds can be automatic including some tests. In the previous example, the active folds are *FOLD2*, *FOLD3*, and *FOLD4*; the state of cells included in *FOLD0*, *FOLD1*, and *FOLD5* are not computed because they are passive.

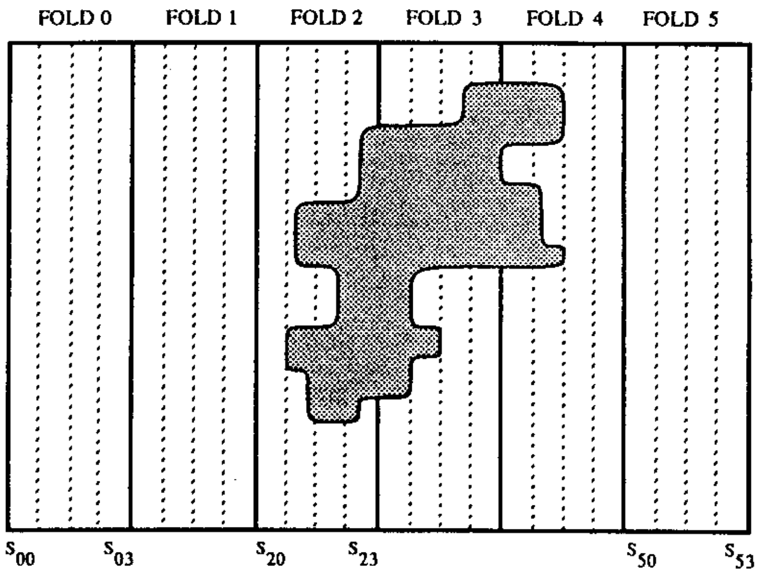

Figure 5. Domain partitioning in six folds.

The task distribution based on folds requires each processor *Pi* to communicate only with processors *Pi-1* and *Pi+1*, so a simple logical ring connecting all the processors is sufficient to accomplish all the communications. On the other hand, to obtain a good distribution in complex simulations it is necessary to increase the number of folds *n*.

This increases the complexity of each *Macrocell* that manages *n* disjoint and independent partitions. Since the number of folds *n* is a parameter of CAMEL, it is necessary to find a trade-off between a better distribution of the active cells and the complexity increase of the *Macrocell*. As an example, in the simulation of a lava flow of the last Etnean eruption (see section 6.1), using this load balancing strategy the execution time dropped by a 30% compared to the time obtained using a simple static strategy.

Finally, it is important to underline that the discussed strategy has improved the overall performance also in very dynamic problems with a high number of contemporary active domains.

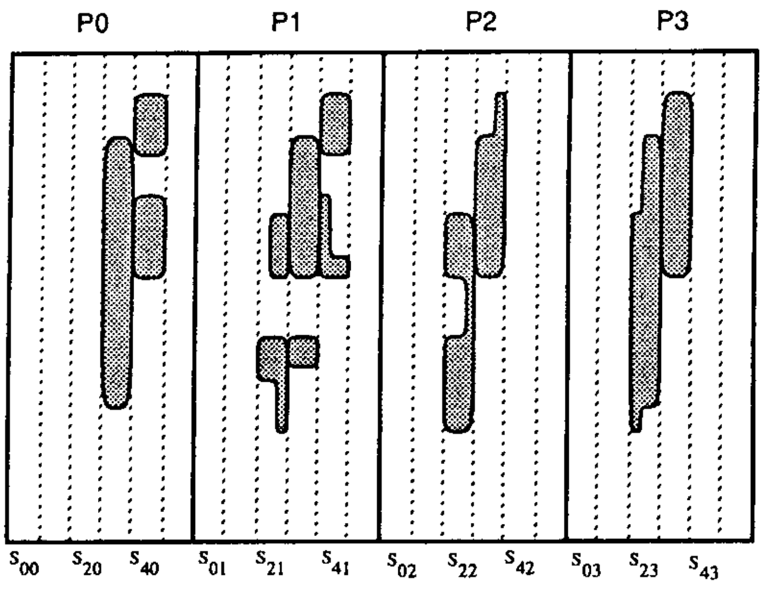

Figure 6. Task distribution on CAMEL.

5. PERFORMANCE RESULTS

Scalability is a common goal in the design of parallel computers. It shows the potential ability of parallel computers to speedup their performance by adding more processing elements.

Performance of CAMEL has been evaluated running several applications. In all cases the system demonstrates to be scalable. The scalability of CAMEL has been measured increasing the number of PEs used to solve the same problem in a shorter time (*speedup*), and increasing the number of PEs used to solve bigger problems in the same time (*scaleup)*.

Figure 7 shows the speedup obtained on an automaton composed of 256 x 256 cells which simulated a combustion system. The speedup on 32 PEs is 28.11 (87.8% efficient). Similar performances has been obtained with different automata.

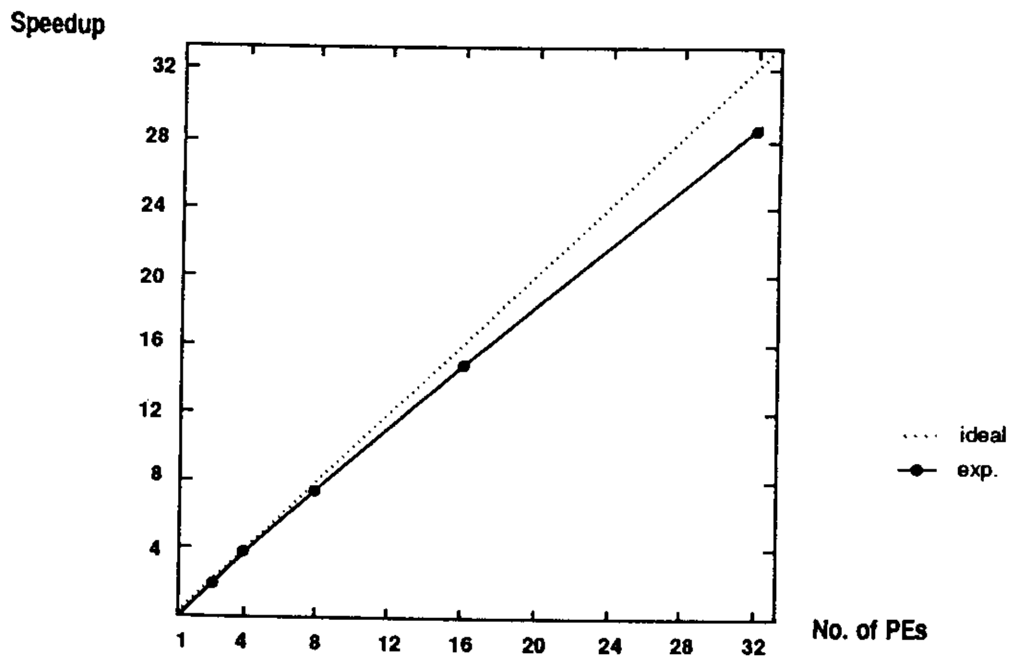

Figure 7. Speedup of CAMEL.

Table 1 shows the execution times (in seconds) obtained running the same combustion system simulation on 2, 8, and 32 PEs using three different grid sizes.

Times in table 1 represent a measure of the *scaleup* of the system. It shows that if the number of PEs is increased proportionally to the size of the problem, bigger problems might be solved in the same time using a larger configuration.

Table 1. Times for different grid sizes on three different configurations.

Grid size	Number of PEs	Time for 1 step	Time for 100 steps	Time for 500 steps
128 x 128	2	0.539	54.12	269.44
256 x 256	8	0.544	54.41	271.77
512 x 512	32	0.548	54.85	274.14

6. RELATED WORK

With regard to the applications of CA, a rough classification concerning the CA models which have been developed for the simulation of complex dynamic systems could be defined. Such a classification can be expressed by a bipolarity.

On one hand, there are models with *ea* having few states (less than hundred) and a simple transition function. They usually describe the systems from a microscopic viewpoint and often need an optical control, step by step, on the evolution of the CA. On the other hand, a broader application of CA needs more detailed descriptions in order to capture the relevant elements of the system to be simulated which can involve million states with a complex transition function of the *ea*.

No real parallel environment can represent a practical solution to different types of models, so CAM-6, a specialized machine for CA, constitutes an instrument which is difficult to surpass when the CA model can be formulated in a simple way (it doesn't mean trivial), however it does not allows the implementation of CA with a large number of states. In fact, CAM is well suited for CA with a number of states not higher than sixteen and with a combination of the few neighborhood indexes. It also permits a complete visualization of the CA evolution, step by step, up to the limits of perception of the human eye.

CAMEL covers requirements diametrically opposite, and several advantages can be evidenced from various viewpoints considering its features.

- There is no practical limit to the number of states of the *ea* (the RAM of a single processing element is up to 8 Mbytes), so it is possible to reduce every other possible neighborhood to the directly available neighborhoods by a trade off with the states. Furthermore, when the model must be validated, it is important to experiment different levels of detail.
- The MIMD architecture permits, by load balancing, to introduce time optimization for a stationary region, if any exists.
- The flexibility of the environment permits to balance a larger size of cellular space increasing the number of processors, with a little proportional slowing down of the computation.

CAMEL partially shares these listed characters with other CA environments; a short comparison may be tried.

Pussycat [16] is a parallel simulation system for cellular automata developed on Transputers which can support *ea* with no more than 256 states. Tommasini [19] discusses an implementation of CA on a shared memory MIMD machine, a Sequent Symmetry parallel computer. This approach shows the limits in the performance because of the memory access contention when the number of processors needs to be increased. Wilding et alii [20] exhibit CA models implementations on a SIMD computer, a DAP, i.e. an Active Memory Technology Distributed Array Processor with interesting results, but obviously no time optimization for the stationary region can be introduced using the DAP system.

At last, we consider CAPE, a CA environment implemented on a transputer network too [15]. A first difference is given by the lack in CAPE of a load balancing strategy, not crucial (to our knowledge) for the applications developed for CAPE with *ea* having few states, but very important for several other applications. Such a feature could be introduced easily; however an opportune choice is not trivial because a load balancing algorithm could increase excessively the inter-processor communication overhead for particular cases of neighborhood indexes and/or when a large number of states is defined. Furthermore, the policy of CAPE is centered on a quick export of Fortran programs, running on a sequential machine, to a parallel machine.

The CAMEL approach assumes a user with a little programming background sufficient to roughly understand the CAMEL philosophy, to learn in a few time the CAMEL utilities and the small sequential subset of the Occam language, necessary to define the transition function. In such a way, CAMEL becomes very transparent to a user, which benefits from a greater flexibility and deeper interaction features, which seems not to be available in CAPE.

7. AN APPLICATION OF CAMEL

This section presents a significant application of modeling and simulation of lava flows developed using CAMEL. We first describe how the problem can be defined using the CA model, then discuss how it can be programmed by CAMEL.

The model of a CA for lava flow is given following the specifications of reference [1] with little changes:

$$A_{lava} = (R_2, X, S, \sigma)$$

where

- $R_2 = \{(x, y) | x, y \in N, 0 \le x \le l_x, 0 \le x \le l_y\}$ is the set of points with integer coordinates in the finite region, where the phenomenon evolves. N is the set of natural numbers.
- $X = \{(0,0), (0,1), (0,-1), (1,0), (-1,0)\}$ identifies the neighboring cells, i.e., respectively, the cell itself and the *north*, *south*, *east* and *west* cells.
- The finite set S of states of the *ea* is defined as follows:

$$S = S_a \times S_h \times S_T \times S_f^4$$

where

S_a is correlated to the altitude of the cell;

S_h is correlated to the lava thickness of the cell;

S_T is correlated to the lava temperature of the cell;

S_f^4 is correlated to the lava flows toward the only four neighborhood directions.

The elements of S_a are integers, which express the value of the altitude (dm). The elements of S_h are integers, which represent the lava quantity inside the cell, expressed as lava thickness in dm. The elements of S_T are integers, which express the temperature as tenths of Kelvin degree. The elements of S_f are integers, which express the outflow rate as lava thickness in dm.

- $\sigma : S^5 \rightarrow S$ is the deterministic state transition for the cells in R_2, it is executed at each step to obtain a new state for each cell.

At the start time we specify the states of the cells in R_2, defining the initial configuration of the CA. Initial values of substates S_h and S_f are null. At each step the function σ is applied to all cells in R_2, and the A_{lava} evolves.

The behavior of the state transition function σ can be shortly described by the following statements:

a) is computed the outflow from a cell at the previous step and the remaining lava thickness in the cell;

b) if the lava temperature drops below the solidus the cell's altitude is increased by the thickness of lava, then the lava thickness of the cell is reset to zero;

c) is computed the inflow to the cell coming from its neighbors and the new lava thickness of the cell adding to the inflow to the previous lava thickness;

d) if the new lava thickness is not null two steps are performed: the first step takes the average temperature of lava passing through a cell, the second step then estimates the temperature drop due to thermal energy losses at the surface A applying the following formula:

$$T_2 = T_1 \sqrt[3]{1 + (T_1^3 p\, A/V)}$$

where T_1 and T_2 are respectively the old and new average temperature, V is the volume of lava in the cell, and p is a "cooling parameter" determined semiempirically.

e) the outflow from a cell towards its neighbor cells is computed; the outflow from a cell depends on the hydrostatic pressure gradients across the cell due to differences in lava thicknesses compared with neighboring cells, and on the adherence of the lava in the cell.

To program this application using CAMEL we only needed to design the procedure `compute_state` described in figure 8 which performs the operations of the state transition function defined above. The procedure has been linked to the CAMEL environment. CAMEL mapped its code automatically on each computing node and executed it in parallel to update the state of each cell. A user may then observe the system evolution through the Graphical Interface and eventually change the parameters of the simulation by the User Interface.

In figure 8 `NS` defines the number of substates. The parameters `i` and `j` indicate the position of a cell in the grid and the arrays `curr_state` and `next_state` are the current and the next state of the automaton. The vector `macro_feat` contains the macroscopic features of the system which is simulated, for example, the cooling parameter (`p`) and the temperature of solidification ($\mathtt{T}_{\mathtt{solid}}$).

```
PROC compute_state ([NS][grid_width][grid_height] INT curr_state,
                    next_state, INT i, j, [NoOfFeat] REAL macro_feat)
  INT lavathick, inflow, outflow, $T_1$;
  BEGIN
   FOR  ind= 1 TO neighbors
a:   outflow:= curr_state[$S_f$(ind)][i][j];
   lavathick:= curr_state[$S_h$][i][j] - outflow;
   IF curr_state[$S_T$][i][j] < Tsolid
     THEN
b:     next_state[$S_a$][i][j]:= curr_state[$S_a$][i][j] + lavathick;
       lavathick:= 0;
   ENDIF
   inflow:= curr_state[$S_f$][i][j-1] + curr_state[$S_f$][i][j+1]+
c:          curr_state[$S_f$][i-1][j] + curr_state[$S_f$][i+1][j];
   next_state[$S_h$][i][j]:= inflow + lavathick;
   IF next_state[$S_h$][i][j] > 0
     THEN
       $T_1$:= average value of neighbors temperature ;
       next_state[$S_T$][i][j]:= $T_1(1 + (T_1^3\ p\ A/V))^{1/3}$;
       FOR ind= 1 TO neighbors
d:       next_state[$S_f$(ind)][i][j]:= f(curr_state[$S_h$],
                                        curr_state[$S_a$], i, j);
     ELSE
       next_state[$S_T$][i][j]:= 0;
       FOR ind= 1 TO neighbors
         next_state[$S_f$(ind)][i][j]:= 0;
   ENDIF
  END
```

Figure 8. The procedure **compute_state** for the *lava flow* problem.

This model was validated with the simulation of 1986-1987 Etnean lava flows on a sequential computer with computation time of the same order of the actual event (some days). But, only the parallel implementation by CAMEL has permitted to use it on a real emergency in Sicily when a lava flow threatened the town of Zafferana Etnea in April and May of 1992. In this case the time necessary to complete the simulation was about 1 hour and 45 minutes without the use of the load balancing strategy and 1 hour and 10 minutes using load balancing.

Various hypotheses were followed, considering different lava flow rates from the initial event, simulating the most unfavorable conditions imaginable and the effects of artificial obstacles in order to deviate the lava flow. Figure 9 shows such a simulation, and figure 10 shows the same case with the outcome that the construction of an artificial barrier has on the flow.

8. CONCLUSIONS AND FUTURE WORK

We have presented a scalable parallel environment based on the cellular automata theory implemented on a Transputer-based multicomputer. CAMEL is a development environment which allows a user to solve real complex problems using efficiently the computing power offered by a highly parallel computer in a transparent way.

CAMEL implements CA as SPMD programs. According to this approach, CA are implemented as a number of processes each one mapped on a distinct PE which executes the same code on different data. A load balancing strategy has been defined to distribute the cells over the processing nodes, minimizing the occurrence of overloaded nodes while other nodes are underloaded.

The behavior of the current CAMEL prototype has been very satisfying for the various applications implemented on it. However, more enhancements could be provided for a better and broader usage of the system. In particular, we identified the following features to be added.

- To define functions which allow to transform a generic neighborhood relation in one of the neighborhood relations which has been implemented in CAMEL. In order to avoid that, this must be performed by a user when he defines a CA. Together with these functions which represent an encoding of the CA states must be defined the correspondent decoding functions for the graphical processing.
- To allow the definition of facilities in the state transition function for calculating and storing statistical values on regular intervals of time.
- To introduce simple graphical utilities and a general support to create data files to be used for complex graphical facilities for CA with more than two dimensions. Naturally, such complex graphical facilities must be let to specialized programs, which process the data files created by CAMEL

according to the necessity of the user. However, a first rough optical control is necessary during the running of simulations, so that it is important to represent the state evolution on the screen.

The research work on the CAMEL environment will continue along these directions. At the same time, the system is utilized for the simulation and modeling of complex phenomena such as landslides and floods, and for low and medium level processing in computer vision and image understanding.

REFERENCES

[1] D. Barca, G.M. Crisci, S. Di Gregorio, and F.P. Nicoletta, Cellular automata methods for modeling lava flow: simulation of the 1986-1987 etnean eruption, in: C. Kilburn and G. Luongo G., eds., *Active Lavas* (UCL Press London, 1993).

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

[3] P. Brinch Hansen, Model programs for computational science: a programming methodology for multicomputers, *Concurrency: Practice and Experience* 5 (5) (1993) 407-423.

[4] A.W. Burks ed., *Essays on Cellular Automata* (University of Illinois Press, 1970).

[5] M. Cannataro, E. Gallizzi, G. Spezzano, and D. Talia, Design, implementation and evaluation of a deadlock-free routing algorithm for concurrent computers, *Concurrency: Practice and Experience* 4 (2) (1992) 143-161.

[6] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, CAMEL : A Cellular Automata Environment on a Highly Parallel Computer, *Proc. of the International Section of AICA'93* (1993) 143-158.

[7] T.L. Casavant and J.G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Transactions on Software Engineering* **SE-14** (2) (1988) 141-154.

[8] J.H. Conway, E.R. Berlekamp, and R.K. Guy, *Winning Ways for your Mathematical Plays* (Academic Press, New York, 1982).

[9] S. Di Gregorio and G. Trautteur, On reversibility in cellular automata, *J. of Computer and System Sciences* **11** (3) (1975) 382-391.

[10] S. Di Gregorio, F. P. Di Maio, P. G. Lignola, A. Zumpano, Simulation of

heterogeneous reactive processes by Cellular Automata, *Atti del I Convegno Nazionale di Informatica Chimica* (1991) 118-120.

[11] Y. Feldman and E. Shapiro, Spatial machines: a more realistic approach to parallel computation, *Communications of the ACM* **35** (10) (1992) 60-73.

[12] Inmos, *The Transputer Products Overview Manual*, SGS-Thomson Microsystems (1991).

[13] A. Lindenmayer, Cellular automata, formal languages and development systems, *Fourth Int. Congress for Logic, Methodology and Philosophy of Science*, Bucarest (1971).

[14] D. M. Nicol and J.H. Saltz, An Analysis of Scatter Decomposition, *IEEE Transactions on Computers* **C-39** (11) (1990) 1337-1345.

[15] M.G. Norman, J.R. Henderson, I.G. Main, and D.J. Wallace, The use of the CAPE environment in the simulation of rock fracturing, *Concurrency: Practice and Experience* **3** (6) (1991) 687-698.

[16] E. Pauwels, Pussycat: a parallel simulation system for cellular automata on transputers, in: *Parallel Processing and Artificial Intelligence* (Wiley, 1989) 233-247.

[17] J.W. Thatcher, Universality in the Von Neumann cellular model, in: A.W. Burk, ed., *Essays on Cellular Automata* (University of Illinois Press, 1970).

[18] T. Toffoli and N. Margolus, *Cellular Automata Machines - A New Enviroment for Modeling*, (MIT Press, 1987).

[19] M. Tomassini, Cellular automata calculations on a shared memory MIMD machine, *Supercomputing review* (1990) 2-6.

[20] N.B. Wilding, *et al.*, Scientific modeling with massively parallel SIMD computers, *Proceedings of IEEE* **79** (4) (1991) 574-585.

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

[22] S. Wolfram, Computation theory of cellular automata, *Comm. Math. Phys.* **96**, (1984) 15-57.

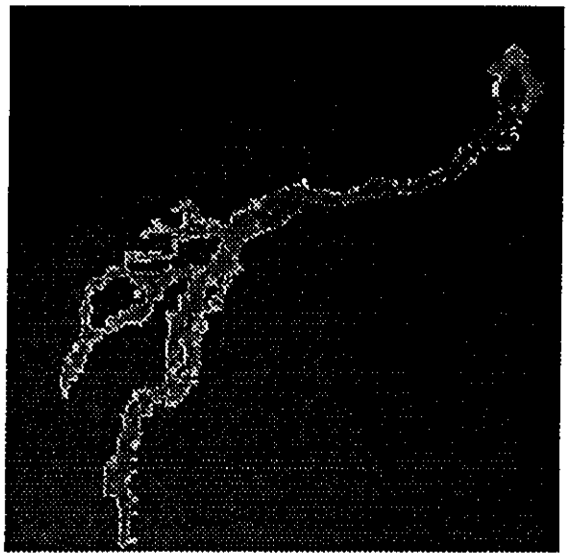

Figure 9. Lava flow simulation.

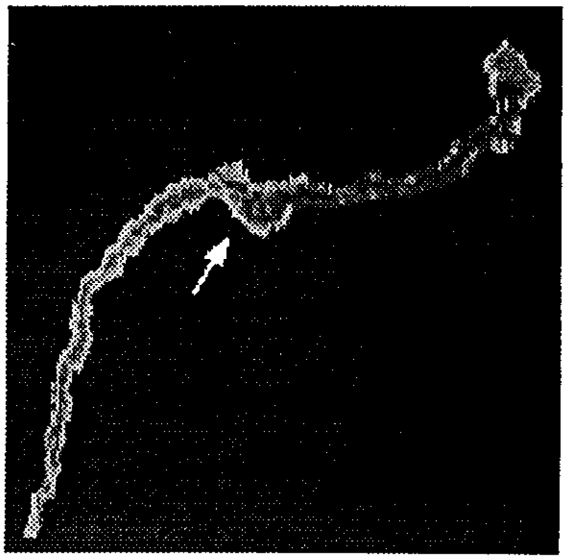

Figure 10. The same lava flow with an artificial barrier.