UNIVERSITY OF CALABRIA

Department of Mathematics and Computer Science
Via P. Bucci
I-87036 Rende, Italy

OpenCAL User Guide

The Open Cellular Automata Library

Version 1.0

Donato D'Ambrosio, Alessio De Rango, Maurizio Macri', Marco Oliverio, Luigi Olivella, Davide Spataro, Rocco Rongo, and William Spataro

December, 2015

Acknowledgements

.....

Contents

Chapter 1

Introduction

Macroscopic Cellular Automata (MCA) represent a parallel computing methodology based on the Cellular Automata paradigm for modelling complex systems at a macroscopic level of description. Well known examples of applications include the simulation of natural phenomena such as lava and debris flows, forest fires, agent based social processes such as pedestrian evacuation and highway traffic problems, besides many others.

Many Cellular Automata software environments and libraries exist. However, when non-trivial modelling is needed, only not open source software are generally available. This is particularly true for Macroscopic Cellular Automata, for which only a significant example of non free software exists, namely the CAMELot Cellular Automata Simulation Environment.

In order to fill this deficiency in the world of free software, the `OpenCAL` C Library has been developed. Similarly to CAMELot, it allows for a simple and concise definition of both the transition function and the other characteristics of the cellular automaton definition. Moreover, it allows for both sequential and parallel execution, both on CPUs and GPUs (thanks to the adoption of the OpenMP and OpenCL, respectively), hiding most parallel implementation issues to the user.

The library has been tested on both CPUs anf GPUs by considering different Cellular Automata, including the well known Conway's Game of Life and the Macroscopic Cellular Automata model SciddicaT for the simulation of debris flows. Results have demonstrated the goodness the new library both in terms of usability and performance.

In the present release 1.0 of the library, 2D and 3D cellular automata can be defined. The library also offers diverse facilities (e.g. it provides many predefined cell's neighborhoods), allows to explicitate the simulation main cycle and provides a dynamic load balancing algorithm. Moreover, an interactive 2D/3D visualization system was developed, based on OpenGL compatibility profile.

Chapter 2

Quick Start

If you wish to get started by just typing a few lines and running an example, this section is for you. In any case, more details on the installation process are given in chapter Installation on page 4.

2.1 Download

OpenCAL source code is available on https://github.com/OpenCALTeam/opencal. To obtain a working copy of the library use the following commands:

```
1
2 user@machine:-$ cd <git root>
3 user@machine:-$ git clone https://github.com/OpenCALTeam/opencal
4 user@machine:-$ cd opencal
```

Listing 2.1: OpenCAL download

2.2 Build

OpenCAL requires *cmake*[1] and *make* for building the library (see section **??** on page **??** .

```
1
2 user@machine:-$ cd opencal && mkdir build && cd build
3 user@machine:-$ cmake ../
4 user@machine:-$ make
```

Listing 2.2: OpenCAL download

[1] Minimum required version: 2.8

Chapter 3

Installation

3.1 Introduction

This guide presents `OpenCAL`, an open source C/C++ library for implementing models based on the Cellular Automata (CA) paradighm. Specifically, the library was developed with the aim to permit a straightforward and simple implementation of Cellular Automata models, which are particularly suitable for the simulation of spatial extended dynamical systems. Key features of `OpenCAL` are the following:

- Parallel and Multiplatform.
- Support for GPU execution, using OpenCL and CUDA.
- Support for Complex Cellular Automata
- Other Key features here

3.2 Obtaining OpenCAL

3.3 Structure of the Distribution Directory

The distribution contains the following files and subdirectories:

- **AUTHORS**: Authors of `libautoti`.

3.4 Requirements and dependencies

To compile `libautoti`, you must have an ANSI C++ compiler that includes a full implementation of the Standard Library and related header files. Additionally, if you want to obtain a parallel version of the library, you must have

an implementation of the Message Passing Interface (MPI) for the parallel computer or workstation network you are running on. If you do not have a native version of MPI for your computer, several machine-independent implementations are available. Most of the testing and development of `libautoti` was done by using the MPICH2 implementation of MPI, which is freely available. Additional information about MPICH2 is available on the World Wide Web at http://www.mcs.anl.gov/research/projects/mpich2/.

3.4.1 Installing prerequisites

3.5 Build and installing

3.5.1 cmake options

3.6 Web Page and Bug Reporting

The World Wide Web page for `libautoti` is http://autoti.mat.unical.it and contains up-to-date news and a list of bug reports. For info or bug reports send an electronic mail to libautoti@mat.unical.it.

When reporting a bug, please include as much information and documentation as possible. Helpful information would include `libautoti` version, MPI implementation and version used, configuration options, type of computer system, problem description, and error message output.

Chapter 4

OpenCAL

With OpenCAL, we identify the sequential version of the software library, which runs on just a single core of your CPU, and represents the basis for the other parallel versions. Moreover, it allows for some *unsafe operations*, which can significantly speed up your application. Such unsafe operations can also be found in the OpenMP version, while they are not present to GPU one.

In the following sections, we will introduce OpenCAL by examples. In the first part of the Chapter, we will deal with the OpenCAL's safe mode, while in the last one, we will go deep inside OpenCAL, discussing unsafe operations.

4.1 Conway's Game of Life

In order to introduce you to Cellular Automata development with OpenCAL, we start this section by implementing the Conway's Game of Life. It represents one of the most simple, yet powerful examples of Cellular Automata, devised by mathematician John Horton Conway in 1970.

The Game of Life can be thought as an infinite two-dimensional orthogonal grid of square cells (the cellular space), each of which is in one of two possible states, dead or alive. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent to it (the Moore neighborhood). At each time step, one of the following transitions occur:

1. Any live cell with fewer than two alive neighbors dies, as if by loneliness.
2. Any live cell with more than three alive neighbors dies, as if by overcrowding.
3. Any live cell with two or three alive neighbors lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbors comes to life.

The initial configuration of the system specifies the state (dead or alive) of each cell into the cellular space. The evolution of the system is thus obtained by applying the above rules (the CA transition function) simultaneously to every cell in the cellular space, so that each new configuration is function of the one at the previous step. The rules continue to be applied repeatedly to create further generations. For more details on the Game of life you can check Wikipedia at the URL http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

The formal definition of the Life CA is reported below.

$$Life =< R, X, Q, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The firt coordinate, i, represents the row, while the second, j, the column. The cell at coodinates $(0, 0)$ is located at the top-left corner of the computational grid.

- $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0), (-1,-1), (1,-1), (1,1), (-1,1)\}$ is the Moore neighborhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

$$N(X, (i, j)) =$$

$$= \{(i, j) + (0, 0), (i, j) + (-1, 0), \ldots, (i, j) + (-1, 1)\} =$$

$$= \{(i, j), (i - 1, j), \ldots, (i - 1, j + 1)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighbourhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X, (i, j), n)$$

  represents the n^{th} neighbourhood of the cell (i, j). Thereby, $N(X, (i, j), 0) = (i, j)$, i.e. the central cell, $N(X, (i, j), 1) = (i - 1, j)$, i.e. the first neighbour, and so on.

- $Q = \{0, 1\}$ is the set of cell states.

- $\sigma : Q^9 \rightarrow Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the previously descripted transition rules.

```
1  // Conway's game of Life Cellular Automaton
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <stdlib.h>
7
8  // declare CA, substate and simulation objects
9  struct CALModel2D* life;
10 struct CALSubstate2Di* Q;
11 struct CALRun2D* life_simulation;
12
13 // The cell's transition function
14 void life_transition_function(struct CALModel2D* life, int i, int j)
15 {
16   int sum = 0, n;
17   for (n=1; n<life->sizeof_X; n++)
18     sum += calGetX2Di(life, Q, i, j, n);
19
20   if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
21     calSet2Di(life, Q, i, j, 1);
22   else
23     calSet2Di(life, Q, i, j, 0);
24 }
25
26 int main()
27 {
28   // define of the life CA and life_simulation simulation objects
29   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
30   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
31
32   // add the Q substate to the life CA
33   Q = calAddSubstate2Di(life);
34
35   // add transition function's elementary process
36   calAddElementaryProcess2D(life, life_transition_function);
37
38   // set the whole substate to 0
39   calInitSubstate2Di(life, Q, 0);
40
41   // set a glider
42   calInit2Di(life, Q, 0, 2, 1);
43   calInit2Di(life, Q, 1, 0, 1);
44   calInit2Di(life, Q, 1, 2, 1);
45   calInit2Di(life, Q, 2, 1, 1);
46   calInit2Di(life, Q, 2, 2, 1);
47
48   // save the Q substate to file
49   calSaveSubstate2Di(life, Q, "./life_0000.txt");
50
51   // simulation run
52   calRun2D(life_simulation);
53
54   // save the Q substate to file
55   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
56
57   // finalize simulation and CA objects
58   calRunFinalize2D(life_simulation);
59   calFinalize2D(life);
60
61   return 0;
62 }
```

Listing 4.1: An OpenCAL implementation of the Conway's game of Life.

```
struct CALModel2D* calCADef2D (
  int rows,
  int columns,
  enum CALNeighborhood2D CAL_NEIGHBORHOOD_2D,
  enum CALSpaceBoundaryCondition CAL_TOROIDALITY,
  enum CALOptimization CAL_OPTIMIZATION
)
```

Listing 4.2: Definition of the calCADef2D() function.

The program below shows a simple Game of Life sequential implementation in C with OpenCAL. As you can see, even if Listing 4.1 is very short, it completely defines the Conway's Game of Life CA and perform a simulation (actually, only one step in this example).

In order to use OpenCAL, you need to include some header files (lines 3-5). Specifically, `cal2D.h` (line 3) allows you to define the CA object (line 9) and the related substate (line 10), while `cal2DRun.h` (line 5) allows you to define a CA simulation object (line 11), needed to run the CA model. The `cal2DIO.h` header file (line 4) provides you some input/output functions for reading/writing substates from/to file.

While statements at lines 9-11 just declare the required objects, they are defined later in the `main` function. In particular, the life CA object is defined at line 29 by the `calCADef2D()` function. The first 2 parameters define the CA dimensions (the number of rows and columns, respectively), while the third the neighbourhood pattern. The fourth parameter specifies the boundary conditions. In this case, the CA cellular space is considered as a torus, with cyclic behaviour at boundaries. The last parameter allows you to specify if your model has to use the so called *active cells optimization*, that is able to restrict the computation to only *non-stationary cells*. In this case, no optimization is considered.

The complete definition of `calCADef2D()` is provided in Listing 4.2. In particular, the `CALNeighborhood2D` enum type (Listing 4.3) allows you to select one of the square or hexagonal predefined neighbourhoods, or a custom neighbourhood, whose pattern can be defined directly in your application. Custom neighbourhoods will be discussed later in this Chapter. Similarly, the `CALSpaceBoundaryCondition` enum type (Listing 4.4) allows you to set non-ciclic or cyclic behaviour at the boundaries of the cellular space. Eventually, the `CALOptimization` enum type (Listing 4.5) allows you to use or not the active cells optimization.

The CA simulation object is defined at line 30 by the `calRunDef2D()` function. The first parameter is a pointer to a CA object (`life` in our case), while the second and third parameters specify the initial and last simulation step, respectively. In this case, we just perform one step of computation, being both the first and last step set to 1. The last parameter allows you to specify the substate update policy. It can be implicit or explicit. In the first case, OpenCAL does substates' updates for you, while in the second case the substates' updates

```
enum CALNeighborhood2D {
  CAL_CUSTOM_NEIGHBORHOOD_2D,
  CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
  CAL_MOORE_NEIGHBORHOOD_2D,
  CAL_HEXAGONAL_NEIGHBORHOOD_2D,
  CAL_HEXAGONAL_NEIGHBORHOOD_ALT_2D
};
```

Listing 4.3: The CALNeighborhood2D enum type.

```
enum CALSpaceBoundaryCondition{
  CAL_SPACE_FLAT = 0,
  CAL_SPACE_TOROIDAL
};
```

Listing 4.4: The CALSpaceBoundaryCondition enum type.

is your responsibility. Note that, in case implicit update policy is applyied, all the CA substates are updated after the execution of each elementary process composing the CA transition function. We will discuss update policies later in this Chapter. The complete definition of `calRunDef2D()` is provided in Listing 4.6. The `CALUpdateMode` type (Listing 4.7) enumerates possible update policies.

Line 33 allocates memory and registers the substate `Q` to the `life` CA, while line 36 adds an elementary process to the cell transition function. The `calAddSubstate2Di()` function is very simple and self-explanatory. At the contrary, `calAddElementaryProcess2D()` must be discussed more in detail. It takes the handle to the CA model to which the elementary process must be attached and a pointer to a callback function, that defines the elementary process itself. In our example, we specified `life_transition_function` as second parameter, being it the name of a developer-defined function that you can find at lines 14-24. As you can see, the elementary process callback returns `void`. Moreover, it takes a pointer to a CA object as first parameter, followeb by a couple of integers, representing the coordinates of the generic cell in the CA space. This is the function prototype which is common to each elementary process you add to your application. Note that, each elementary process is applyed by OpenCAL simultaneously to each cell of the cellular space in a computational step. However, this is completely transparent to the user, so that he/she can concentrate his/her effort on the definition of single cell behaviour.

When the user is going to implement an elementary process, by defining its

```
enum CALOptimization{
  CAL_NO_OPT = 0,
  CAL_OPT_ACTIVE_CELLS
};
```

Listing 4.5: The CALOptimization enum type.

```
struct CALRun2D* calRunDef2D (
  struct CALModel2D* ca2D,
  int initial_step,
  int final_step,
  enum CALUpdateMode UPDATE_MODE
)
```

Listing 4.6: Definition of the calRunDef2D() function.

```
enum CALUpdateMode {
  CAL_UPDATE_EXPLICIT = 0,
  CAL_UPDATE_IMPLICIT
};
```

Listing 4.7: The CALUpdateMode enum type.

callback function, he/she can rely on a set of OpenCAL functions that allow to get the substates values of both the central and the neighbouring cells, and to update the substates values of the central cell. In the specific case of the Game of Life, we used the `calGet2Di()` function to get the central cell's value of the substate `Q` (remember that the central cell is identified by the coordinates (i, j), coming from the callback parameters), the `calGetX2Di()` function to get the value of the n-th neighbour's substate `Q`, and the `calSet2Di()` function to update the value of the substate `Q` for the central cell. In the Game of Life example, we defined just one elementary process, that therefore represents the whole cell transition function. However, as we will see later, many elementary processes can be defined in OpenCAL by simply calling the `calAddElementaryProcess2D()` function many times. If you define more than one elementary process, they will be executed in the order they are added to the CA.

The `calInitSubstate2Di()` function at line 39 sets the whole substate `Q` to the value 0, i.e. the value of the substate `Q` is set to 0 in each cell of the cellular space. The following lines, from 42 to 46, set the value of the substate `Q` for some cells to 1, in order to define a well known *glider* pattern. In this case, we provided the cells coordinates as the third and fourth parameters. In this way, we define the initial condition of the system direcly inside the `main` function. However, as we will see later in this Chapter, such kind of initialization can be performed by means of a specific function.

The `calSaveSubstate2Di()` function (line 49) saves the substate `Q` to file, while the `calRun2D()` function (line 52) enters the simulation loop (actually, only one computational step in this example), and returns to the `main` function when the simulation is complete. The `calSaveSubstate2Di()` is thus called again at line 55 to save the new (last) configuration of the CA (represented by the only defined substate `Q`) to file, while the last two functions at lines 58 and 59 release previously allocated memory. The `return` statement at line 61 ends our first example.

Figures 4.1 and 4.2 show the initial and final configuration of Game of Life

```
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4.1: Initial configuration of Game of Life, as implemented in Listing 4.1.

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4.2: Final configuration of Game of Life (actually, just one step of computation), as implemented in Listing 4.1.

as implemented in Listing 4.1, respectively.

4.2 OpenCAL statement convention

As you can easily see from a rapid sight to the source code, all the OpenCAL statements are characterized by a prefix and a suffix. All the data types have the `CAL` prefix, and an optional suffix that identifies the CA dimension (e.g. 2D for a two-dimensional model) and the basic type. For instance, in the case of the Life's `Q` substate, the 2Di suffix of the `CALSubstate2Di` type specifies that it is a two-dimensional substate in whihc each element is of integer type.

More in detail, OpenCAL comes with three different basic numeric types, whihc are in lowercase (besides the prefix):

- `CALbyte`, corresponding to the `char` C data type;
- `CALint`, corresponding to the `int` C data type;
- `CALreal`, corresponding to the `long double` C data type;

while the possible substates types are:

- `CALSubstate2Db`, corresponding to a `CALbyte` based substate;
- `CALSubstate2Di`, corresponding to a `CALint` based substate;
- `CALSubstate2Dr`, corresponding to a `CALreal` based substate;

Also the OpenCAL constants have a prefix, namely the `CAL_` one, followd by at least one uppercase keyword. In case of more kewords, they are separated by the _ character. Eventually, all the OpenCAL functions start with the `cal` suffix, followed by at least one capitalized keyword, and end with a suffix specifying the CA dimension and the basic datatype.

4.3 SciddicaT

In the previous section we illustrated an OpenCAL implementation of a simple cellular automaton, namely the Conways Game of Life. Here, we will deal with a more complex example concerning the implementations of the SciddicaT Cellular Automata model for landslide simulation. Different versions will be presented, ranging from a naiv to a fully optimized implementation.

Sciddica is a family of bi-dimensional CCA debris flow models, successfully applied to the simulation of many real cases, such as the 1988 Mt. Ontake (Japan) landslide and the 1998 Sarno (Italy) disaster. An oversimplified toy-version of Sciddica (SciddicaT in the following) was here considered to be implemented in `OpenCAL`, and its application to the 1992 Tessina (Italy) landslide shown.

SciddicaT considers the surface over which the phenomenon evolves as subdivided in square cells of uniform size. Each cell changes its state by means of the transition function, which takes as input the state of the cells belonging to the von Neumann neighborhood. It is formally defined as:

$$SciddicaT =< R, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space over which the phenomenon evolves. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The firt coordinate, i, represents the row, while the second, j, the column. The cell at coodinates $(0, 0)$ is located at the top-left corner of the computational grid.
- $X = \{(0, 0), (-1, 0), (0, -1), (0, 1), (1, 0)\}$ is the von Neumann neighborhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

$$N(X, (i, j)) =$$

$$= \{(i, j) + (0, 0), (i, j) + (-1, 0), (i, j) + (0, -1), (i, j) + (0, 1), (i, j) + (1, 0)\} =$$

$$= \{(i, j), (i - 1, j), (i, j - 1), (i, j + 1), (i + 1, j)\}$$

Here, a subscript operator can be used to index cells belonging to the neighbourhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X, (i, j), n)$$

represents the n^{th} neighbourhood of the cell (i, j). Thereby, $N(X, (i, j), 0) = (i, j)$, i.e. the central cell, $N(X, (i, j), 1) = (i - 1, j)$, i.e. the first neighbour, and so on.

- Q is the set of cell states. It is subdivided in the following substates:
  - Q_z is the set of values representing the topographic altitude (i.e. elevation);
  - Q_h is the set of values representing the debris thickness;
  - Q_o^4 are the sets of values representing the debris outflows from the central cell to the neighboring ones.

  The Cartesian product of the substates defines the overall set of state Q:

$$Q = Q_z \times Q_h \times Q_o^4$$

  so that the cell state is specified by the following sixtuple:

$$q = (q_z, q_h, q_{o_0}, q_{o_1}, q_{o_2}, q_{o_3})$$

  In particular, q_{o_0} represents the outflows from the central cell towards the neighbour 1, q_{o_1} the outflow towards the neighbour 2, and so on.

- P is set of parameters ruling the CA dynamics:
  - p_ϵ is the parameter which specifies the thickness of the debris that cannot leave the cell due to the effect of adherence;
  - p_r is the relaxation rate parameter, which affects the size of outflows (cf. section above).

- $\sigma : Q^5 \to Q$ is the deterministic cell transition function. It is composed by two elementary processes, listed below in the same order they are applied:
  - $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to Q_o^4$ determines the outflows from the central cell to the neighboring ones by applying the *minimization algorithm of the differences*. In brief, a preliminary control avoids outflows computation for those cells in which the amount of debris is smaller or equal to p_ϵ, acting as a simplification of the adherence effect. Thus, by means of the minimization algorithm, outflows $q_o(0, m)$ $(m = 0, \ldots, 3)$ from the central cell towards its four adjecent

cells are evaluated, and the Q_o^4 substates accordingly updated. Note that, $q_o(0,0)$ represents the aoutflow from the central cell towards the neighbour 1, $q_o(0,1)$ the aoutflow towards the neighbour 2, and so on. In general, $q_o(0,m)$ represnets the outflows from the central cell towards the $n = (m+1)^{th}$ neighbouring cell. Eventually, a relaxation rate factor, $p_r \in]0,1]$, is considered in order to obtain the local equilibrium condition in more than one CA step. This can significantly improve the realism of model as, in general, more than one step may be needed to displace the proper amount of debris from a cell towards the adjacent ones. In this case, if $f(0,m)$ $(i = 0,\ldots,3)$ represent the outgoing flows towards the 4 adjacent cells, as computed by the minimization algorithm, the resulting outflows are given by $q_o(0,m) = f(0,m) \cdot p_r$ $(i = 0,\ldots,3)$.

- $\sigma_2 : Q_h \times (Q_o^4)^4 \to Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood: $h'(0) = h(0) + \sum_{m=0}^{3}(q_o(0,m) - q_o(m,0))$. Here, $h'(0)$ is the new debris thickness inside the cell, while $q_o(m,0)$ represents the inflow from the $n = (m+1)^{th}$ neighbouring cell. No parameters are involved in this elementary process.

In the following Listing 4.8, an OpenCAL implementation of SciddicaT is shown.

```
1  // The SciddicaT debris flows CCA simulation model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  // Some definitions...
10 #define ROWS 610
11 #define COLS 496
12 #define P_R 0.5
13 #define P_EPSILON 0.001
14 #define STEPS 4000
15 #define DEM_PATH "./data/dem.txt"
16 #define SOURCE_PATH "./data/source.txt"
17 #define OUTPUT_PATH "./data/width_final.txt"
18 #define NUMBER_OF_OUTFLOWS 4
19
20 // declare CCA model (sciddicaT), substates (Q), parameters (P),
21 // and simulation object (sciddicaT_simulation)
22 struct CALModel2D* sciddicaT;
23
24 struct sciddicaTSubstates {
25   struct CALSubstate2Dr *z;
26   struct CALSubstate2Dr *h;
27   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
28 } Q;
29
```

```
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37 // The sigma_1 elementary process
38 void sciddicaT_flows_computation(struct CALModel2D* sciddicaT, int i,
       int j)
39 {
40   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
41   CALbyte again;
42   CALint cells_count;
43   CALreal average;
44   CALreal m;
45   CALreal u[5];
46   CALint n;
47   CALreal z, h;
48
49   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
50     return;
51
52   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54   for (n=1; n<sciddicaT->sizeof_X; n++)
55   {
56     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58     u[n] = z + h;
59   }
60
61   //computes outflows
62   do{
63     again = CAL_FALSE;
64     average = m;
65     cells_count = 0;
66
67     for (n=0; n<sciddicaT->sizeof_X; n++)
68       if (!eliminated_cells[n]){
69         average += u[n];
70         cells_count++;
71       }
72
73       if (cells_count != 0)
74         average /= cells_count;
75
76       for (n=0; n<sciddicaT->sizeof_X; n++)
77         if( (average<=u[n]) && (!eliminated_cells[n]) ){
78           eliminated_cells[n]=CAL_TRUE;
79           again=CAL_TRUE;
80         }
81   }while (again);
82
83   for (n=1; n<sciddicaT->sizeof_X; n++)
84     if (eliminated_cells[n])
```

```
85        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
86      else
87        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
88  }
89
90  // The sigma_2 elementary process
91  void sciddicaT_width_update(struct CALModel2D* sciddicaT, int i, int j)
92  {
93    CALreal h_next;
94    CALint n;
95
96    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
97    for(n=1; n<sciddicaT->sizeof_X; n++)
98      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j,
            n) - calGet2Dr(sciddicaT, Q.f[n-1], i, j);
99
100   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
101 }
102
103 // SciddicaT simulation init function
104 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //initializing substates to 0
110   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
111   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
112   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
113   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
114
115   //sciddicaT parameters setting
116   P.r = P_R;
117   P.epsilon = P_EPSILON;
118
119   //sciddicaT source initialization
120   for (i=0; i<sciddicaT->rows; i++)
121     for (j=0; j<sciddicaT->columns; j++)
122     {
123       h = calGet2Dr(sciddicaT, Q.h, i, j);
124
125       if ( h > 0.0 ) {
126         z = calGet2Dr(sciddicaT, Q.z, i, j);
127         calSet2Dr(sciddicaT, Q.z, i, j, z-h);
128       }
129     }
130 }
131
132 // SciddicaT steering function
133 void sciddicaTSteering(struct CALModel2D* sciddicaT)
134 {
135   // set flow to 0 everywhere
136   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
137   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
138   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
139   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
140 }
```

```
141
142 int main()
143 {
144   time_t start_time, end_time;
145
146   // define of the sciddicaT CA and sciddicaT_simulation simulation
        objects
147     sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D
          , CAL_SPACE_TOROIDAL, CAL_NO_OPT);
148   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
        CAL_UPDATE_IMPLICIT);
149
150   // add transition function's sigma_1 and sigma_2 elementary processes
151   calAddElementaryProcess2D(sciddicaT, sciddicaT_flows_computation);
152   calAddElementaryProcess2D(sciddicaT, sciddicaT_width_update);
153
154   // add substates
155   Q.z = calAddSubstate2Dr(sciddicaT);
156   Q.h = calAddSubstate2Dr(sciddicaT);
157   Q.f[0] = calAddSubstate2Dr(sciddicaT);
158   Q.f[1] = calAddSubstate2Dr(sciddicaT);
159   Q.f[2] = calAddSubstate2Dr(sciddicaT);
160   Q.f[3] = calAddSubstate2Dr(sciddicaT);
161
162   // load configuration
163   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
164   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
165
166   // simulation run
167   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
168   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
169   printf ("Starting simulation...\n");
170   start_time = time(NULL);
171   calRun2D(sciddicaT_simulation);
172   end_time = time(NULL);
173   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
        start_time);
174
175   // saving configuration
176   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
177
178   // finalizations
179   calRunFinalize2D(sciddicaT_simulation);
180   calFinalize2D(sciddicaT);
181
182   return 0;
183 }
```

Listing 4.8: An OpenCAL implementation of the SciddicaT debris flows simulation model.

As for the case of Game of Life, the CA model and the simulation objects are declared as global variables (lines 22 and 35, respectively), and defined later into the main function (lines 147 and 148, respevctively). As you can see, the 2D cellular space is a grid of `ROWS` rows times `COLS` columns cells, corresponding to i_{max} and j_{max} of the formal definition, respectively (cf. lines 10-11), while the

von Neumann neighbourhood is adopted. The cellular space is still toroidal, as in Life, and no optimization is considered. Regarding the simulation object, a total of `STEPS` steps (i.e. 4000 steps - cf. line 14) are set, and implicit substates updating considered.

Substates and parameters are grouped into two different C structures (lines 24-28 and 30-33, respectively). Substates are therefore bound to the CA context by means of the `calAddSubstate2Dr()` function (lines 155-160), as well as elementary processes are defined as collback functions by means of the `calAddElementaryProcess2D()` function (lines 151-152).

The topographic altitude and debris thickness substates are initialized from files through the `calLoadSubstate2Dr()` function (lines 163-164), while the remaining initial state of the CA is set by means of the `calRunAddInitFunc2D()` function. It registers the `sciddicaT_simulation_init()` callback, whic is executed once before the execution of the simulation loop, in which the elementary processes are applyed to the whole set of cells of the cellular space. Such a callback function must return void and take a pointer to a simulation obect as parameter. Differently to an elementary process, that can only access state values of cells belonging to the neighbourhood, this function can perform global operations on the whole cellular space. In the specific case of the SciddicaT model, the `sciddicaT_simulation_init()` function (lines 104-130) sets the values of all the outflows from the central cell to its neighbours to zero, by means of the function `calInitSubstate2Dr()` (lines 110-113). Moreover, it sets the values of the P.r and P.epsilon parameters (lines 116-117) and initializes the debris flow source by simply subtracting the source's debris thickness to the topographic altitude. For this purpose, a nested double for is executed to check the debris thickness in each cell of the cellular space. Here, the `sciddicaT->rows` and `sciddicaT->cols` members of the CA object are used, which represent the cellular spece's numbers of rows and columns, respectively. Still, the `calGet2Dr()` and `calSet2Dr()` functions are here employed to read/update substates' values inside the cells.

Line 168 defines a *steering* callback by the `calRunAddSteeringFunc2D()` function. Steering is executed at the end of each computational step (i.e. after all the elementary processe have been applied to each cell of the cellular space), and can perform global operations over the cellular space. In this case, the `sciddicaT_simulation_init()` callback is registered; it must return void and takes a pointer to a simulation object as function parameter. It simply reset (to zero) the outflows everywere through the `calInitSubstate2Dr()` function.

The function `calRun2D()` (line 171) eneters the OpenCAL simulation loop, which exectues a totoal of 4000 steps (cf. lines 14 and 148). Eventually, the final debris flow path is saved to file by means of the `calSaveSubstate2Dr()` function (line 176) and previously allocated memery is released (lines 179-180).

As regards the elementary processes, the first one, σ_1, is defined at lines 38-88, while the second, σ_2, al lines 91-101. In both cases, the `calGet2Dr()` `calGetX2Dr()` functions are employed to get substes' values for the central cell and its neighbours, respectively. Moreover, the `calSet2Dr()` function, updates the central cell's state.

Figure 4.3 shows the SciddicaT simulation of the 1992 Tessina (Italy) landslide. Both the initial landslide source and the final flow path configruation are shown.

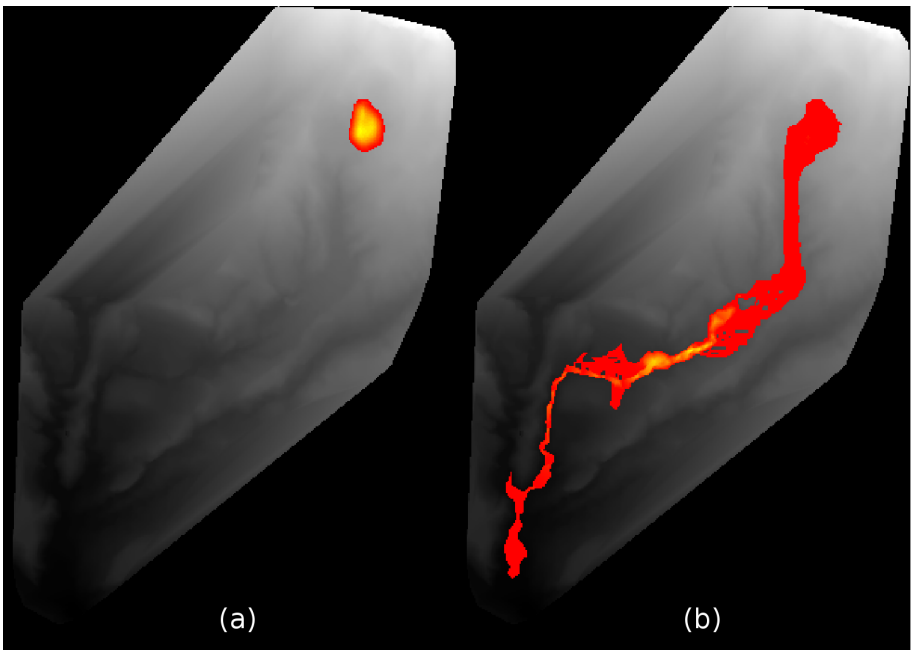

Figure 4.3: SciddicaT simulation of the 1992 Tessina (Italy) landslide. Topographic altitudes are represented in gray scale. Black represents the lower altitude, while the white color is used for the highest elevation in the study area. Debris thickness is represented with colours ranging from red (for lower values) to yellow (for higher values). (a) Initial configuration. (b) Final debris flow path. Note that the graphic output was generated by using the `cal_sciddicaT-glut` application, that implements the SciddicaT model and provides a minimal visualization system. You can find it in the examples directoy.

As regards computational preformace, the simulation shown in Figure 4.3 was executed on a Intel Core i7-4702HQ CPU @ 2.20GHz by exploiting only a single core. The simulation lasted a total of 172 seconds for executing a total of 4000 compuational steps.

Figure 4.4 shows the OpenCAL main loop. Before entering the loop, if defined, the init function is executed. Afterwards, while the current step is lower or equal to the final step of computation (or this latter is set to `CAL_RUN_LOOP`), elementary processes are executed cocurrently [1]. In this cycle, substates are

[1]On the serial version of OpenCAL, implicit parallelism is obtained by exploiting the two different computing planes built into OpenCAL's substates. The first one, that we will call *current*,

updated after each elementary process has been applyed, while just before the end of the computational step, if defined, the steering function is executed. At the end of the computational step, a stop condition is checked, which can stop the simulation before the last step is reached. In order to define such a stop condition, the user can use the `stopCondition()` function, which registers a callback in which the stop condition can be defined.

4.4 SciddicaT with active cells optimization

Here we present a computationally improved version of SciddicaT, which takes advantage of the built-in OpenCAL active cells optimization. As stated above, this optimization is able to restrict computation to a subset of cells which are actually involved in computation, by neglecting those cells for which is sure they will not change state to the next step (stationary cells).

In the case of SciddicaT, only cells containing debris and their neighbours can change state to the next step, as they can be interestefd in mass variation due to otflows and inflows. At the beginning of the simulation, we can simply initialize the set of active cells to those cells containing debris (i.e. those cells forming the initial landslide source). Moreover, we can add to this set new cells or remove some ones from it. Specifically, if an outflow is computed from an active cell towards a neighbouring stationary cell, this latter can be added to the set of active cells and considered for state change by the remaining elementary processes in the current step of cmputation (if any), or by the next computational step. Similarly, if a given active cell looses a sufficient amount of debris, it can be eliminated from the set of active cells. In the case of SciddicaT, this appens when its thickness becomes lower than or euqal to a given threshold (i.e. p_ϵ).

In order to account for these processes, we have to slightly revise the SciddicaT definition. In particular we have to add the set of active cells, A. The optimized SciddicaT model is now defined as

$$SciddicaT =< R, A, X, Q, P, \sigma >$$

where $A \subseteq R$ is the set of active cells, while the other components are defned as before. The transition function is now defined as:

$$\sigma : A \times Q^5 \rightarrow Q \times A$$

denoting that it is applied to only the cells in A and that it can add/remove active cells. More in detail, the σ_1 elementary process have to be modified, as

is used to read substates's values for the central cell and its neighbours, while the second, that we will call *next*, is used to update the new state for the central cell. When all the cells have been processed and the new state values updated, computing planes are switched, i.e. the *next* plane is assumed as *current* and the *current* as *next*, and the process is reiterated. In this manner, the *current* computing plane is not *corrupted* during a computational step, being new values written to the *next* plane. Note that, even in the case more processing units are used to compute the next CA state and more cells are updated simltaneously, which is the case of OpenCAL-OMP and OpenCAL-CL, the two computing planes are manteined

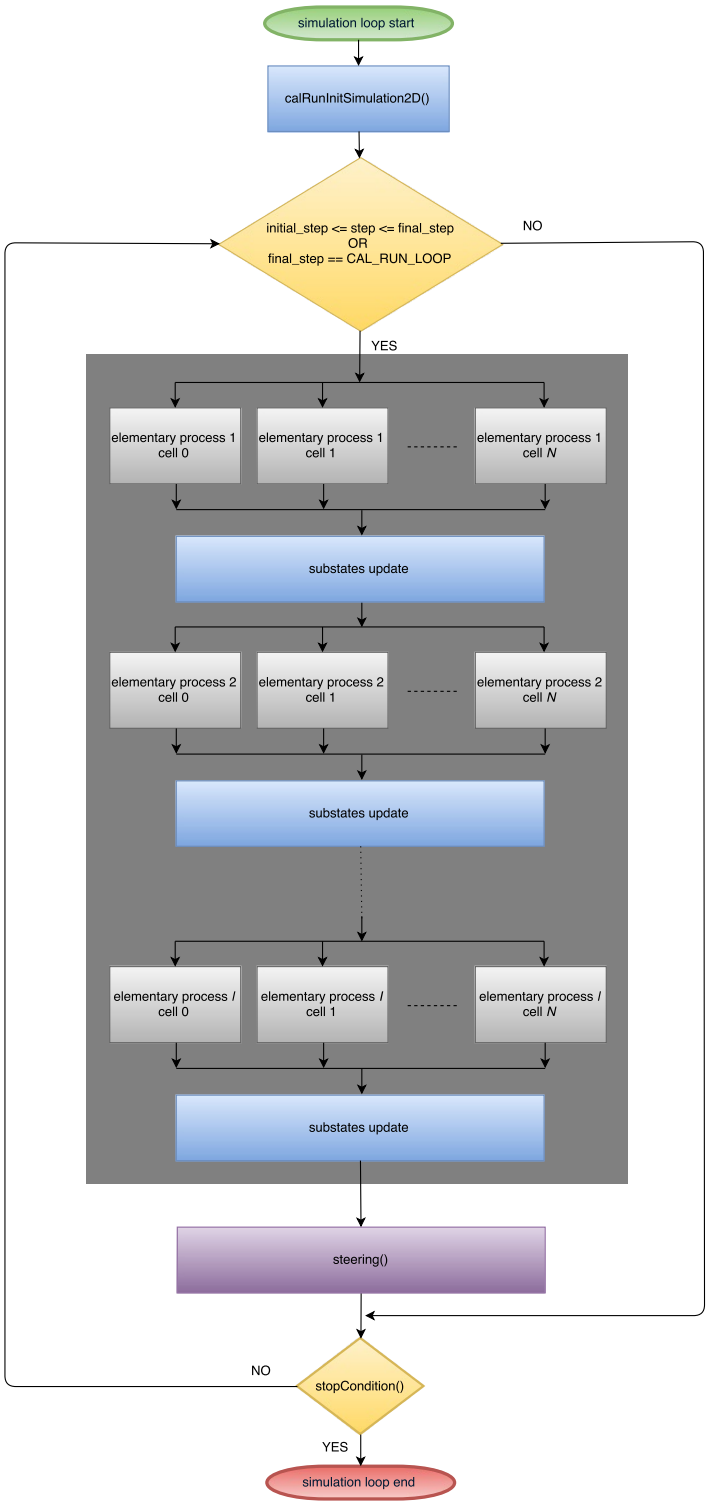

Figure 4.4: OpenCAL main loop chart.

it can activate new cell. Moreover, a new elementary process, σ_3, have to be added in order to remove cells that cannot produce outflows during the next computational step due to the fact that their debris thickness is negligible. The new sequence of elementary processes is listed below, in the same order they are applied.

- $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4 \times A$ determines the outflows from the central cell to the neighboring ones, as before. In addition, each time an outflow is computed, the neighbour receiving the flow is added to the set of active cells.
- $\sigma_2 : A \times Q_h \times (Q_o^4)^4 \rightarrow Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood. This elementary process does not change with respect to the original version of SciddicaT.
- $\sigma_3 : A \times Q_h \times p_\epsilon \rightarrow A$ removes a cell from A if its debris thickness is lower than or equal to the p_ϵ threshold.

In order to implement the SciddicaT debris flows model in OpenCAL by exploiting the active cells optimization, we have to chage the definition of the CA objet, by also adding the third σ_3 elementary process to it. Moreover, the σ_1 elementary process have to also be changed. A complete implementation of the sactive cells optimized version of SciddicaT is shown in Listing 4.9 for the sake of completeness, even if only the differences with respect to the original impleme ntation are commented.

```
1  // The SciddicaT debris flows model with the active cells optimization
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
```

```
27 } Q;
28
29 struct sciddicaTParameters {
30   CALParameterr epsilon;
31   CALParameterr r;
32 } P;
33
34
35 // The sigma_1 elementary process
36 void sciddicaT_transition_function(struct CALModel2D* sciddicaT, int i,
       int j)
37 {
38   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
       CAL_FALSE};
39   CALbyte again;
40   CALint cells_count;
41   CALreal average;
42   CALreal m;
43   CALreal u[5];
44   CALint n;
45   CALreal z, h;
46   CALreal f;
47
48
49   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
50   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
51   for (n=1; n<sciddicaT->sizeof_X; n++)
52   {
53     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
54     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
55     u[n] = z + h;
56   }
57
58   //computes outflows and updates debris thickness
59   do{
60     again = CAL_FALSE;
61     average = m;
62     cells_count = 0;
63
64     for (n=0; n<sciddicaT->sizeof_X; n++)
65       if (!eliminated_cells[n]){
66         average += u[n];
67         cells_count++;
68       }
69
70       if (cells_count != 0)
71         average /= cells_count;
72
73       for (n=0; n<sciddicaT->sizeof_X; n++)
74         if( (average<=u[n]) && (!eliminated_cells[n]) ){
75           eliminated_cells[n]=CAL_TRUE;
76           again=CAL_TRUE;
77         }
78
79   }while (again);
80
81   for (n=1; n<sciddicaT->sizeof_X; n++)
```

```
82      if (eliminated_cells[n])
83        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
84      else
85      {
86        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
87        calAddActiveCellX2D(sciddicaT, i, j, n);
88      }
89  }
90
91  // The sigma_2 elementary process
92  void sciddicaT_width_update(struct CALModel2D* sciddicaT, int i, int j)
93  {
94    CALreal h_next;
95    CALint n;
96
97    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
98    for(n=1; n<sciddicaT->sizeof_X; n++)
99      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j,
           n) - calGet2Dr(sciddicaT, Q.f[n-1], i, j);
100
101   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
102 }
103
104 // The sigma_3 elementary process
105 void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
      i, int j)
106 {
107   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
108     calRemoveActiveCell2D(sciddicaT,i,j);
109 }
110
111
112 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
113 {
114   CALreal z, h;
115   CALint i, j;
116
117   //sciddicaT parameters setting
118   P.r = P_R;
119   P.epsilon = P_EPSILON;
120
121   //initializing substates to 0
122   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
123   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
124   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
125   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
126
127   //sciddicaT source initialization
128   for (i=0; i<sciddicaT->rows; i++)
129     for (j=0; j<sciddicaT->columns; j++)
130     {
131       h = calGet2Dr(sciddicaT, Q.h, i, j);
132
133       if ( h > 0.0 ) {
134         z = calGet2Dr(sciddicaT, Q.z, i, j);
135         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
136
```

```
137         //adds the cell (i, j) to the set of active ones
138         calAddActiveCell2D(sciddicaT, i, j);
139       }
140     }
141 }
142
143 // SciddicaT steering function
144 void sciddicaTSteering(struct CALModel2D* sciddicaT)
145 {
146   // set flow to 0 everywhere
147   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
148   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
149   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
150   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
151 }
152
153
154 int main()
155 {
156   time_t start_time, end_time;
157
158   // define of the sciddicaT CA and sciddicaT_simulation simulation
        objects
159   struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
        CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
        CAL_OPT_ACTIVE_CELLS);
160   struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1,
        STEPS, CAL_UPDATE_IMPLICIT);
161
162   // add transition function's sigma_1 and sigma_2 elementary processes
163   calAddElementaryProcess2D(sciddicaT, sciddicaT_transition_function);
164   calAddElementaryProcess2D(sciddicaT, sciddicaT_width_update);
165   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
        ;
166
167   // add substates
168   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
169   Q.h = calAddSubstate2Dr(sciddicaT);
170   Q.f[0] = calAddSubstate2Dr(sciddicaT);
171   Q.f[1] = calAddSubstate2Dr(sciddicaT);
172   Q.f[2] = calAddSubstate2Dr(sciddicaT);
173   Q.f[3] = calAddSubstate2Dr(sciddicaT);
174
175   // load configuration
176   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
177   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
178
179   // simulation run
180   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
181   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
182   printf ("Starting simulation...\n");
183   start_time = time(NULL);
184   calRun2D(sciddicaT_simulation);
185   end_time = time(NULL);
186   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
        start_time);
187
```

```
188   // saving configuration
189   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
190
191   // finalizations
192   calRunFinalize2D(sciddicaT_simulation);
193   calFinalize2D(sciddicaT);
194
195   return 0;
196 }
```

Listing 4.9: An OpenCAL implementation of the SciddicaT debris flows simulation model with the active cells optimization.

As you can easily see, few modifications to the original source code are needed to add the active cells optimization to SciddicaT. In particular, the active cells optimization is enableb by means of the `CAL_OPT_ACTIVE_CELLS` parameter at line 159, while the third elementary process added at line 165. As regards the elementary processe σ_1, it is the same of the one of the basic SciddicaT version, with the exception that when an outflow is generated, the cell receiving the flow is added to the set A of the active cells (line 88). Moreover, an active cell is eliminated by the set A by means of the σ_3 elementary process in the case its debris thickness becomes lower than or equal to the P_ϵ threshold parameter (lines 107-108).

Regarding the computational preformace, the same simulation shown in Figure 4.3 was executed using the new SciddicaT implementation adoptig the active cells implementation. Still, only a single core of the same Intel Core i7-4702HQ CPU was used, as we did before. The simulation lasted a total of 22 seconds, versus 172 seconds obtained for the basic (non-optimized) version, whihc is about 8 times faster. Given the small required implementatice effort, we think it can be condidered a very good result.

4.5 SciddicaT as eXtended CA

OpenCAL allows for further optimization of the SciddicaT debris flows simulation model by means of the so called *unsafe operations*. In fact, in some cases, it is possible to consider an extended definition of the computational model, allowing for operations that are not strictly permitted by the formal definition of Cellular Automata. In particular, we will allow the transition function to update the state of the neighbouring cells, while the CA only allows for state change for of the central one. When we will permit such a kind of unsafe operations, we will talk about *XCA eXtended Cellular Automata*. Obviously, the extended CA must be equivalent to the original one in terms of computational results.

An XCA equivalent version of SciddicaT can be obtained by obseving that, when an outflow is computed from the central cell towards a neighbour, the flow can be immediatly subtracted from the central cell and added to the neighbour. This does not change the state of the system at the current step, which is represented by means of the *current* computational plane, as updated

values are written to the *next* plane. Thus, the *current* computational plane is not corrupted by the extended operation, while the *next* plane is used for immediatly accounting mass variations inside the cells. By introducing such feature, ouflows don't need to be saved into otflows substates anymore, as they are used to account mass exchange directly during ouflows computation. As you can figure out, this can give rise to a further performace improvement of the application. The SciddicaT XCA model is formally defined as:

$$SciddicaT =< R, A, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 2-dimensional cellular space over which the phenomenon evolves. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The firt coordinate, i, represents the row, while the second, j, the column. The cell at coodinates $(0, 0)$ is located at the top-left corner of the computational grid.
- $A \subseteq R$ is the set of active cells, i.e. those cells actually involved in computation.
- $X = \{(0, 0), (-1, 0), (0, -1), (0, 1), (1, 0)\}$ is the von Neumann neighborhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i, j) is given by

  $$N(X, (i, j)) =$$

  $$= \{(i, j) + (0, 0), (i, j) + (-1, 0), (i, j) + (0, -1), (i, j) + (0, 1), (i, j) + (1, 0)\} =$$

  $$= \{(i, j), (i - 1, j), (i, j - 1), (i, j + 1), (i + 1, j)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighbourhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

  $$N(X, (i, j), n)$$

  represents the n^{th} neighbourhood of the cell (i, j). Thereby, $N(X, (i, j), 0) = (i, j)$, i.e. the central cell, $N(X, (i, j), 1) = (i - 1, j)$, i.e. the first neighbour, and so on.
- Q is the set of cell states; it is subdivided in the following substates:
  - Q_z is the set of values representing the topographic altitude (i.e. elevation);
  - Q_h is the set of values representing the debris thickness;

The Cartesian product of the substates defines the overall set of state Q:

$$Q = Q_z \times Q_h$$

so that the cell state is specified by:

$$q = (q_z, q_h)$$

- P is set of parameters ruling the CA dynamics:
  - p_ϵ is the parameter which specifies the thickness of the debris that cannot leave the cell due to the effect of adherence;
  - p_r is the relaxation rate parameter, which affects the size of outflows (cf. section above).
- $\sigma : A \times Q^5 \to Q$ is the deterministic cell transition function. It is composed by two elementary processes:
  - $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to (A \times Q_h)^5$ determines the outflows from the central cell to the neighboring ones and updates debris thickness inside the central cell and its neighbours accordingly. It also adds the neighbouring cells receining a flow to the set A of the active cells.
  - $\sigma_2 : A \times Q_h \times p_\epsilon \to A$ removes the cell from the set A of the active cells if the debris thickness inside the cell is lower than or equal to the p_ϵ threshold.

Note that, only the topographic altitude and the debris thicness are now considered as model's substates, as the four outflows substates are no longer needed. Moreover, the number of elementary process now considered is two, instead of three for the previous versions of SciddicaT. The OpenCAL implementation of the further optimized SciddicaT debris flows model is shown in Listing 4.10.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
```

```
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26 } Q;
27
28 struct sciddicaTParameters {
29   CALParameterr epsilon;
30   CALParameterr r;
31 } P;
32
33
34 // The sciddicaT transition function
35 void sciddicaT_transition_function(struct CALModel2D* sciddicaT, int i,
       int j)
36 {
37   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
38   CALbyte again;
39   CALint cells_count;
40   CALreal average;
41   CALreal m;
42   CALreal u[5];
43   CALint n;
44   CALreal z, h;
45   CALreal f;
46
47
48   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
49   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
50   for (n=1; n<sciddicaT->sizeof_X; n++)
51   {
52     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
53     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
54     u[n] = z + h;
55   }
56
57   //computes outflows and updates debris thickness
58   do{
59     again = CAL_FALSE;
60     average = m;
61     cells_count = 0;
62
63     for (n=0; n<sciddicaT->sizeof_X; n++)
64       if (!eliminated_cells[n]){
65         average += u[n];
66         cells_count++;
67       }
68
69       if (cells_count != 0)
70         average /= cells_count;
71
72       for (n=0; n<sciddicaT->sizeof_X; n++)
```

```
73          if( (average<=u[n]) && (!eliminated_cells[n]) ){
74            eliminated_cells[n]=CAL_TRUE;
75            again=CAL_TRUE;
76          }
77
78    }while (again);
79
80    for (n=1; n<sciddicaT->sizeof_X; n++)
81      if (!eliminated_cells[n])
82      {
83        f = (average-u[n])*P.r;
84        calSet2Dr (sciddicaT,Q.h,i,j,   calGetNext2Dr (sciddicaT,Q.h,i,j)
                - f );
85        calSetX2Dr(sciddicaT,Q.h,i,j,n, calGetNextX2Dr(sciddicaT,Q.h,i,j,
              n) + f );
86
87        //adds the cell (i, j, n) to the set of active ones
88        calAddActiveCellX2D(sciddicaT, i, j, n);
89      }
90  }
91
92
93  void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
        i, int j)
94  {
95    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
96      calRemoveActiveCell2D(sciddicaT,i,j);
97  }
98
99
100 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
101 {
102   CALreal z, h;
103   CALint i, j;
104
105   //sciddicaT parameters setting
106   P.r = P_R;
107   P.epsilon = P_EPSILON;
108
109   //sciddicaT source initialization
110   for (i=0; i<sciddicaT->rows; i++)
111     for (j=0; j<sciddicaT->columns; j++)
112     {
113       h = calGet2Dr(sciddicaT, Q.h, i, j);
114
115       if ( h > 0.0 ) {
116         z = calGet2Dr(sciddicaT, Q.z, i, j);
117         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
118
119         //adds the cell (i, j) to the set of active ones
120         calAddActiveCell2D(sciddicaT, i, j);
121       }
122     }
123 }
124
125
126 int main()
```

```
127 {
128   time_t start_time, end_time;
129
130   // define of the sciddicaT CA and sciddicaT_simulation simulation
          objects
131   struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
          CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
          CAL_OPT_ACTIVE_CELLS);
132   struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1,
          STEPS, CAL_UPDATE_IMPLICIT);
133
134   // add transition function's sigma_1 and sigma_2 elementary processes
135   calAddElementaryProcess2D(sciddicaT, sciddicaT_transition_function);
136   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
          ;
137
138   // add substates
139   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
140   Q.h = calAddSubstate2Dr(sciddicaT);
141
142   // load configuration
143   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
144   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
145
146   // simulation run
147   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
148   printf ("Starting simulation...\n");
149   start_time = time(NULL);
150   calRun2D(sciddicaT_simulation);
151   end_time = time(NULL);
152   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
153
154   // saving configuration
155   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
156
157   // finalizations
158   calRunFinalize2D(sciddicaT_simulation);
159   calFinalize2D(sciddicaT);
160
161   return 0;
162 }
```

Listing 4.10: An OpenCAL implementation of the SciddicaT further optimized debris flows simulation model.

As you can see, the definitions of CA and simulation objects doesn't change from the previous implementation (lines 131-132), while only two elementary processes are considered (lines 135-136). In particular, the firt call to `calAddElementaryProcess2D()` registers the callbak function implementing the σ_1 elementary process. It computes outflows from the (active) central cell to its neighbours (line 83) and updates the debris tickness in both the central cell and the neighbouring cell receiving a flow (lines 84-85). Moreover, neighbouring cells receiving a flow are added to the set A of active cells (line 88) and therefore will be considered for elaboration by the subsuequent elementary

process (σ_2) in the current step of computation[2] and, if it is not removed by σ_2), by the subsequent computational steps. In particular, the `calSetX2Dr()` *unsafe* function is used to update the derbis thickess of the neighbouring cells receiving a flow, while the `calAddActiveCellX2D()` function is used to add a neighbouring cells receiving a flow to the set A of active cells. The σ_2 elementary process, simply remove inactive cells from A (lines 95-86), as in the previous example.

Substates are added to the CA at lines 139-140. Here, note that the firt substate, Q_z, is added by menas of the `calAddSingleLayerSubstate2Dr()` function. It is here considered to allocate memory only for the *current* computing plane. In fact, topographic altitutde only changes at the simulation initialization stage (cf. lines 147 and 117), while it remains inalterated during computation as its value is never updated by the transition function. This allows for memory space allocation optimization and possibly for computational performance improvements. Note that, at line 117 we used the `calSetCurrent2Dr()` function, instead of the usual `calSet2Dr()`. The `calSetCurrent2Dr()` function allows for updating the *current* computational plane (the only present in the Q_z substate), while `calSet2Dr()` would update the *next* computational plane, by producing an access violation error.

Regarding the computational preformace, the same simulation shown in Figure 4.3 was executed by considering the XCA implementation of SciddicaT on a single core of the same processor. The simulation lasted a total of 11 seconds, versus 22 seconds obtained for the active cells optimized version and 172 seconds for the basic (non-optimized) version. The XCA implementation resulted 2 times faster than the active cells optimized version and about 16 times faster with than the basic one.

4.6 SciddicaT with explicit simulation loop

Even if results obtained so far are more than satisfying, it is further possibile to improve computational performance of SciddicaT by avoiding unnecessary substates updating. In fact, in some cases, elementary processes don't affect one or more model's substates and therefore their updating becomes only a waste of time.

As we stated above, when we use the implicit `calRun2D()` simulation loop, an update of all the defined substates is executed at the end of each elementary process. However, this behaviour can be changed by making the OpenCAL simulation loop explicit.

In the specific case of the SciddicaT XCA model, the second elementary process, σ_2, just remove cells that became inactive from the set A of active

[2] This is due to the fact that a substates' update is performed after the first elementary process has been applied to all the (active) cells of the cellular space. This behaviour is set by menas of the `CAL_UPDATE_IMPLICIT` parameter used in the definition of the simulation object at line 132 of Listing 4.10.

cells and don't affect the mode's substates[3]. As a consequence, no substates updating should be executed after σ_2 application, in order to avoid unnecessary operations.

A new OpenCAL implementation of SciddicaT is presented in Listing 4.11. It is based on an explicit simulation loop and also defines a stopping criterion for the simulation termination. The complete implementation is shown for the sake of completeness.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL/cal2D.h>
4  #include <OpenCAL/cal2DIO.h>
5  #include <OpenCAL/cal2DRun.h>
6  #include <OpenCAL/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct CALModel2D* sciddicaT;
24
25 struct sciddicaTSubstates {
26   struct CALSubstate2Dr *z;
27   struct CALSubstate2Dr *h;
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37
38 // The sciddicaT transition function
39 void sciddicaT_flows(struct CALModel2D* sciddicaT, int i, int j)
40 {
41   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
42   CALbyte again;
43   CALint cells_count;
44   CALreal average;
45   CALreal m;
```

[3] Actually, only the $Q.h$ substate can be update by the transition function, since the other, $Q.z$, is of single-lyer type.

```
46    CALreal u[5];
47    CALint n;
48    CALreal z, h;
49    CALreal f;
50
51
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows and updates debris thickness
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81
82    }while (again);
83
84    for (n=1; n<sciddicaT->sizeof_X; n++)
85      if (!eliminated_cells[n])
86      {
87        f = (average-u[n])*P.r;
88        calSet2Dr (sciddicaT,Q.h,i,j,   calGetNext2Dr (sciddicaT,Q.h,i,j)
                - f );
89        calSetX2Dr(sciddicaT,Q.h,i,j,n, calGetNextX2Dr(sciddicaT,Q.h,i,j,
              n) + f );
90
91        //adds the cell (i, j, n) to the set of active ones
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
        i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
```

```
100     calRemoveActiveCell2D(sciddicaT,i,j);
101 }
102
103
104 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
123         //adds the cell (i, j) to the set of active ones
124         calAddActiveCell2D(sciddicaT, i, j);
125       }
126     }
127
128   calUpdate2D(sciddicaT);
129 }
130
131
132 void sciddicaTransitionFunction(struct CALModel2D* sciddicaT)
133 {
134   // active cells must be updated first becouse outflows
135   // have already been sent to (pheraps inactive) the neighbours
136   calApplyElementaryProcess2D(sciddicaT, sciddicaT_flows);
137     calUpdateActiveCells2D(sciddicaT);
138     calUpdateSubstate2Dr(sciddicaT, Q.h);
139
140   // here you don't need to update Q.h
141   calApplyElementaryProcess2D(sciddicaT,
          sciddicaT_remove_inactive_cells);
142   calUpdateActiveCells2D(sciddicaT);
143 }
144
145
146 CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT)
147 {
148   if (sciddicaT_simulation->step >= STEPS)
149     return CAL_TRUE;
150   return CAL_FALSE;
151 }
152
153
154 int main()
155 {
```

```
156   CALbyte again;
157   time_t start_time, end_time;
158
159   // define of the sciddicaT CA and sciddicaT_simulation simulation
          objects
160   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
161   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
          CAL_UPDATE_EXPLICIT);
162
163   // add transition function's sigma_1 and sigma_2 elementary processes
164   calAddElementaryProcess2D(sciddicaT, sciddicaT_flows);
165   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
          ;
166
167   // add substates
168   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
169   Q.h = calAddSubstate2Dr(sciddicaT);
170
171   // load configuration
172   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
173   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
174
175   // simulation run
176   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
177   calRunAddGlobalTransitionFunc2D(sciddicaT_simulation,
          sciddicaTransitionFunction);
178   calRunAddStopConditionFunc2D(sciddicaT_simulation,
          sciddicaTSimulationStopCondition);
179
180   printf ("Starting simulation...\n");
181   start_time = time(NULL);
182   // applies the callback init func registered by calRunAddInitFunc2D()
183   calRunInitSimulation2D(sciddicaT_simulation);
184   // the do-while explicitates the calRun2D() implicit looop
185   do{
186       again = calRunCAStep2D(sciddicaT_simulation);
187       sciddicaT_simulation->step++;
188   } while (again);
189   calRunFinalizeSimulation2D(sciddicaT_simulation);
190   end_time = time(NULL);
191   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
192
193   // saving configuration
194   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
195
196   // finalizations
197   calRunFinalize2D(sciddicaT_simulation);
198   calFinalize2D(sciddicaT);
199
200   return 0;
201 }
```

Listing 4.11: An OpenCAL implementation of the SciddicaT XCA debris flows simulation model with explicit simulation loop.

As you can see, the `calRunAddGlobalTransitionFunc2D()` function is called

CA model	Elapsed time [s]	Speedup
SciddicaT	240	1
SciddicaT with active cells	23	10.43
SciddicaT XCA (eXtended CA)	13	18.46
SciddicaT XCA explicit loop	12	20

Table 4.1: Computational performace of the four different implementations of the SciddicaT debris flows model.

to register a custom transition function (line 177). In the specific case of SciddicaT, the `sciddicaTransitionFunction()` callback (lines 132-143) is used to make the elementary processes application and the substates update explicit. Here, the elementary processes are applyied in the same order they are defined (which is the default behaviour of OpenCAL). In particular, the `sciddicaT_flows()` elementary process is applyied to each (active) cell into the computational domain by means of the `calApplyElementaryProcess2D()` function. After that, the set A of the active cells and the Q_h substate are updated by means of the `calUpdateActiveCells2D()` and `calUpdateSubstate2Dr()`, respectively.

Table 5.4 resumes the computational performace of the above illustraed versions of SciddicaT.

4.7 A three-dimensional example

In order to introduce you to development with of three-dimensional CA with OpenCAL, we start this section by implementing a simple 3D model, namely the mod2 3D CA. Cells can be in one of two differnt states, 0 or 1, as in Game of Life. The cellular space is a parallelepiped made by cubic cells, while the cell's neighbourhood is the 3D Moore one, consisting of the central cell and its adjecent cells. The transition function simply evaluates the quantity s as the number of neighbouring cells which are in the state 1 and sets the new state for the central cell as $s\%2$.

$$mod2 =< R, X, Q, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the 3-dimensional cellular space. The generic cell in R is individuated by means of a triple of integer coordinates (i, j, k), where $0 \leq i < i_{max}$, $0 \leq j < j_{max}$, and $0 \leq k < k_{max}$. The firt coordinate, i, represents the row, the second, j, the column, while the third coordinate represents the slice. The cell at coodinates $(0, 0, 0)$ is located at the top-left-far corner of the computational grid.

- $X = \{(0,0,0),\ldots,(-1,1,0),(0,0,-1),\ldots,(-1,1,-1),(0,0,1),\ldots,(-1,1,1)\}$ is the Moore neighborhood relation, a geometrical pattern which identifies the cells influencing the state transition of the central cell. The neighborhood of the generic cell of coordinate (i,j) is given by

$$N(X,(i,j,k)) =$$

$$= \{(i,j,k)+(0,0,0),\ldots,(i,j,k)+(-1,1,-1)\} =$$

$$= \{(i,j,k),\ldots,(i-1,j+1,-1)\}$$

  Here, a subscript operator can be used to index cells belonging to the neighbourhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X,(i,j),n)$$

  represents the n^{th} neighbourhood of the cell (i,j). Thereby, $N(X,(i,j,k),0) = (i,j,k)$, i.e. the central cell, $N(X,(i,j,k),1) = (i-1,j,k)$, i.e. the first neighbour, and so on.

- $Q = \{0,1\}$ is the set of cell states.

- $\sigma : Q^{27} \rightarrow Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the previously descripted transition rules.

As you can imagine, the OpenCAL implementation of the mod2 3D CA is very simple, as the Conway's game of Life is. The complete source code is shown in Listing 4.12.

```
1  // mod2 3D Cellular Automaton
2
3  #include <OpenCAL/cal3D.h>
4  #include <OpenCAL/cal3DIO.h>
5  #include <OpenCAL/cal3DRun.h>
6
7  #define ROWS 5
8  #define COLS 7
9  #define LAYERS 3
10
11 // declare CA, substate and simulation objects
12 struct CALModel3D* mod2;
13 struct CALSubstate3Db* Q;
14 struct CALRun3D* mod2_simulation;
15
16 // The cell's transition function
17 void mod2_transition_function(struct CALModel3D* ca, int i, int j, int
       k)
18 {
19   int sum = 0, n;
20
21   for (n=0; n<ca->sizeof_X; n++)
```

```
22      sum += calGetX3Db(ca, Q, i, j, k, n);
23
24    calSet3Db(ca, Q, i, j, k, sum%2);
25  }
26
27  int main()
28  {
29    // define of the mod2 CA and mod2_simulation simulation objects
30    mod2 = calCADef3D(ROWS, COLS, LAYERS, CAL_MOORE_NEIGHBORHOOD_3D,
          CAL_SPACE_TOROIDAL, CAL_NO_OPT);
31    mod2_simulation = calRunDef3D(mod2, 1, 1, CAL_UPDATE_IMPLICIT);
32
33    // add the Q substate to the mod2 CA
34    Q = calAddSubstate3Db(mod2);
35
36    // add transition function's elementary process
37    calAddElementaryProcess3D(mod2, mod2_transition_function);
38
39    // set the whole substate to 0
40    calInitSubstate3Db(mod2, Q, 0);
41
42    // set a seed at position (2, 3, 1)
43    calInit3Db(mod2, Q, 2, 3, 1, 1);
44
45    // save the Q substate to file
46    calSaveSubstate3Db(mod2, Q, "./mod2_0000.txt");
47
48    // simulation run
49    calRun3D(mod2_simulation);
50
51    // save the Q substate to file
52    calSaveSubstate3Db(mod2, Q, "./mod2_LAST.txt");
53
54    // finalize simulation and CA objects
55    calRunFinalize3D(mod2_simulation);
56    calFinalize3D(mod2);
57
58    return 0;
59  }
```

Listing 4.12: An OpenCAL implementation of the mod2 3D CA.

As you can see, even if Listing 4.1 is very short, it completely defines the mod2 3D CA and perform a simulation (actually, only one step in this example). Lines 3-5 include some header files for the 3D version of OpenCAL, while lines 12-14 declare CA, substate and simulation objects. They are therefore defined later in the main function. In particular, line 30 defines the CA as a parallelepiped having `ROWS` rows, `COLS` columns and `SLICES` slices (cf. lines 7-9). Moreover, the 3D Moore neighbourhood is here set as well as cyclic conditions at boundaries. Eventually, no optimizations are considered. Line 31 defines the simulation object by setting just one step of computation and implicit substate's update. Finally, the only substate, Q, is defined at line 34. Note that, since it was defined by means of the `calInitSubstate3Db()` function, each element $q \in Q$ results to be of the CALbyte type. Line 37 registers the only CA's elementary process, namely the `mod2_transition_function()`

```
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  1  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
```

Figure 4.5: Initial configuration of mod2 3D CA, as implemented in Listing 4.12.

function, which is then implemnented at lines 17-25. Line 43 initializes the cell's substated Q at coordinates (2, 3, 1) to the state 1. The obtained initial configuration is then saved to disk at line 46, and the simulation ran at line 49. The final configuration is therefore saved to disk at line 52 and, eventually, memory previously and implicitly allocated is released at lines 55-56. Note that, diffrently to the previous examples, almost all the OpenCAL functions come in the 3D flower. For instace, this is the case of the `alGetX3Db()` and `calSet3Db()` functions at lines 22 and 24, respectively, which that take the `k` parameter inorder to specify the slice coordinate for the cell.

Figures 4.5 and 4.7 show the initial and final configuration of mod2 3D CA as implemented in Listing 4.12, respectively.

4.8 Custom Neighbourhoods

xxx

```
0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0

0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0
```

Figure 4.6: Final configuration of mod2 3D CA (actually, just one step of computation), as implemented in Listing 4.12.

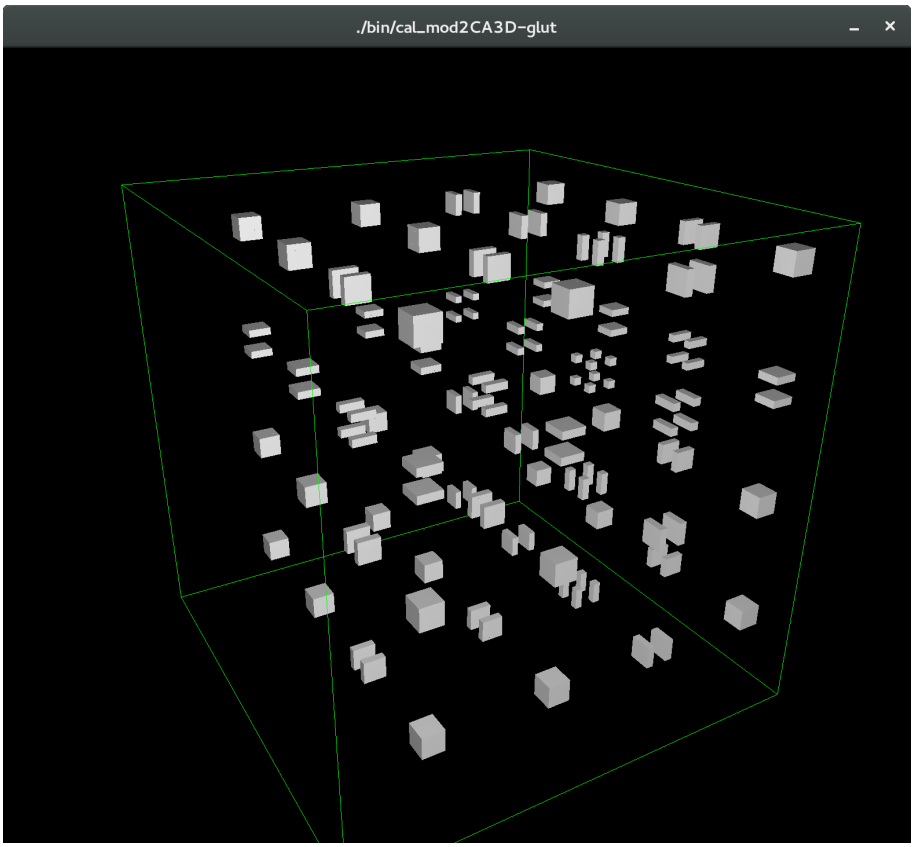

Figure 4.7: Graphical representation of the mod2 3D CA after 77 computational steps, as implemented in Listing 4.12. CA dimensions were set to (rows, cols, slices) = (65, 65, 65), while the initial seed located at coordinates (12, 12, 12). Cells in black are in the state 0, cell in white are in the state 1.

Chapter 5

OpenCAL OpenMP version

With the name OpenCAL-OMP, we identify the OpenMP implementation of the software library, which can run on all the cores for your CPU. If you are luky and have a shared memory multiptocessor system, OpenCAL-OMP can also exploit all the cores of all your CPUs.

Similarly to the serial version, OpenCAL-OMP allows for some *unsafe operations*, which can significantly speed up your application. Howevewr, it must be given the utmost attention due to possible problems related to race condions. In particular, when many theads perform concurrent operations on the same memory locations, if such operation are made by more than one basic machine instructions, it can appen they can interleave, giving rice to wrong results. Furthermore, even in the case of atomic operations, race conditions can occur if the logic order of execution is not respected. For instance, in a sequence of write-read atomic operations, it can occur that the read is performed before the write due to the fact that the thread performing the write is executed first.

In the following sections, we will introduce OpenCAL-OMP by comparing examples source code differences with respect to the serial implemebntations. In the first part of the Chapter, we will deal with the OpenCAL's safe mode, while in the last one, we will discuss unsafe operations.

5.1 Conway's Game of Life in OpenCAL-OMP

In Section **??**, we described Conway's Game of Life and shown a possible implementation using the `OpenCAL` serial library. Here, we present a `OpenCAL-OMP` implementation of the same cellular automaton in listing 5.1, by discussing the differences with respect the serial implementation.

As you can see, the OpenMP-based implementation of Life, which uses only safe operations, is alsmost identical to the serial one (listing 4.1). The only differences can be found at lines 3-5 where, instead of including the OpenCAL header files, you can find the OpenCAL-OMP headers. All the remaining source code is unchanged. Note that also the OpenCAL-OMP statements' prefix does

```
1  // Conway's game of Life Cellular Automaton
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6
7  // declare CA, substate and simulation objects
8  struct CALModel2D* life;
9  struct CALSubstate2Di* Q;
10 struct CALRun2D* life_simulation;
11
12 // The cell's transition function
13 void life_transition_function(struct CALModel2D* life, int i, int j)
14 {
15   int sum = 0, n;
16   for (n=1; n<life->sizeof_X; n++)
17     sum += calGetX2Di(life, Q, i, j, n);
18
19   if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
20     calSet2Di(life, Q, i, j, 1);
21   else
22     calSet2Di(life, Q, i, j, 0);
23 }
24
25 int main()
26 {
27   // define of the life CA and life_simulation simulation objects
28   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
         CAL_SPACE_TOROIDAL, CAL_NO_OPT);
29   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
30
31     //put OpenCAL - OMP in unsafe state execution(to allow unsafe
           operation to be used)
32   calSetUnsafe2D(life);
33
34   // add the Q substate to the life CA
35   Q = calAddSubstate2Di(life);
36
37   // add transition function's elementary process
38   calAddElementaryProcess2D(life, life_transition_function);
39
40   // set the whole substate to 0
41   calInitSubstate2Di(life, Q, 0);
42
43   // set a glider
44   calInit2Di(life, Q, 0, 2, 1);
45   calInit2Di(life, Q, 1, 0, 1);
46   calInit2Di(life, Q, 1, 2, 1);
47   calInit2Di(life, Q, 2, 1, 1);
48   calInit2Di(life, Q, 2, 2, 1);
49
50   // save the Q substate to file
51   calSaveSubstate2Di(life, Q, "./life_0000.txt");
52
53   // simulation run
54   calRun2D(life_simulation);
55
56   // save the Q substate to file
57   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
58
59   // finalize simulation and CA objects
60   calRunFinalize2D(life_simulation);
61   calFinalize2D(life);
62
63   return 0;
64 }
```

Listing 5.1: An OpenCAL-OMP implementation of the Conway's game of Life.

not change with respect to the OpenCAL's one (i.e. `cal` for the functions, `CAL` for the data types, and `CAL_` for the constants). In conclusion, if you only use OpenCAL-OMP in safe mode, besides including the proper OpenCAL-OMP header files instead of the OpenCAL ones, you don't need to change the serial code at all.

5.2 SciddicaT

As for the case of Conway's Game of Life, even the OpenCAL-OMP implementation of SciddicaT cellular automaton, shown in Lsting 5.2, does not significantly differ from the serial implementation that you can find in the previous Chapter. As before, the only differences regard the included headers (lines 3-5). In conclusion, as for the Life cellular automaton, due to the fact we used only OpenCAL-OMP safe operations, besides including the proper OpenCAL-OMP header files instead of the OpenCAL ones, you don't need to change the serial code at all.

```
1  // The SciddicaT debris flows CCA simulation model
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  // Some definitions...
10 #define ROWS 610
11 #define COLS 496
12 #define P_R 0.5
13 #define P_EPSILON 0.001
14 #define STEPS 4000
15 #define DEM_PATH "./data/dem.txt"
16 #define SOURCE_PATH "./data/source.txt"
17 #define OUTPUT_PATH "./data/width_final.txt"
18 #define NUMBER_OF_OUTFLOWS 4
19
20 // declare CCA model (sciddicaT), substates (Q), parameters (P),
21 // and simulation object (sciddicaT_simulation)
22 struct CALModel2D* sciddicaT;
23
24 struct sciddicaTSubstates {
25   struct CALSubstate2Dr *z;
26   struct CALSubstate2Dr *h;
27   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
```

```
37 // The sigma_1 elementary process
38 void sciddicaT_flows_computation(struct CALModel2D* sciddicaT, int i,
       int j)
39 {
40   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
41   CALbyte again;
42   CALint cells_count;
43   CALreal average;
44   CALreal m;
45   CALreal u[5];
46   CALint n;
47   CALreal z, h;
48
49   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
50     return;
51
52   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54   for (n=1; n<sciddicaT->sizeof_X; n++)
55   {
56     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58     u[n] = z + h;
59   }
60
61   //computes outflows
62   do{
63     again = CAL_FALSE;
64     average = m;
65     cells_count = 0;
66
67     for (n=0; n<sciddicaT->sizeof_X; n++)
68       if (!eliminated_cells[n]){
69         average += u[n];
70         cells_count++;
71       }
72
73       if (cells_count != 0)
74         average /= cells_count;
75
76       for (n=0; n<sciddicaT->sizeof_X; n++)
77         if( (average<=u[n]) && (!eliminated_cells[n]) ){
78           eliminated_cells[n]=CAL_TRUE;
79           again=CAL_TRUE;
80         }
81   }while (again);
82
83   for (n=1; n<sciddicaT->sizeof_X; n++)
84     if (eliminated_cells[n])
85       calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
86     else
87       calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
88 }
89
90 // The sigma_2 elementary process
91 void sciddicaT_width_update(struct CALModel2D* sciddicaT, int i, int j)
```

```
92  {
93    CALreal h_next;
94    CALint n;
95
96    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
97    for(n=1; n<sciddicaT->sizeof_X; n++)
98      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j,
            n) - calGet2Dr(sciddicaT, Q.f[n-1], i, j);
99
100   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
101 }
102
103 // SciddicaT simulation init function
104 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //initializing substates to 0
110   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
111   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
112   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
113   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
114
115   //sciddicaT parameters setting
116   P.r = P_R;
117   P.epsilon = P_EPSILON;
118
119   //sciddicaT source initialization
120   for (i=0; i<sciddicaT->rows; i++)
121     for (j=0; j<sciddicaT->columns; j++)
122     {
123       h = calGet2Dr(sciddicaT, Q.h, i, j);
124
125       if ( h > 0.0 ) {
126         z = calGet2Dr(sciddicaT, Q.z, i, j);
127         calSet2Dr(sciddicaT, Q.z, i, j, z-h);
128       }
129     }
130 }
131
132 // SciddicaT steering function
133 void sciddicaTSteering(struct CALModel2D* sciddicaT)
134 {
135     //initializing substates to 0
136   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
137   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
138   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
139   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
140 }
141
142 int main()
143 {
144   time_t start_time, end_time;
145
146   //cadef and rundef
```

```
147   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_NO_OPT);
148   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
          CAL_UPDATE_IMPLICIT);
149
150   //add transition function's elementary processes
151   calAddElementaryProcess2D(sciddicaT, sciddicaT_flows_computation);
152   calAddElementaryProcess2D(sciddicaT, sciddicaT_width_update);
153
154   //add substates
155   Q.z = calAddSubstate2Dr(sciddicaT);
156   Q.h = calAddSubstate2Dr(sciddicaT);
157   Q.f[0] = calAddSubstate2Dr(sciddicaT);
158   Q.f[1] = calAddSubstate2Dr(sciddicaT);
159   Q.f[2] = calAddSubstate2Dr(sciddicaT);
160   Q.f[3] = calAddSubstate2Dr(sciddicaT);
161
162   //load configuration
163   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
164   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
165
166   //simulation run
167   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
168   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
169   printf ("Starting simulation...\n");
170   start_time = time(NULL);
171   calRun2D(sciddicaT_simulation);
172   end_time = time(NULL);
173   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
174
175   //saving configuration
176   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
177
178   //finalizations
179   calRunFinalize2D(sciddicaT_simulation);
180   calFinalize2D(sciddicaT);
181
182   return 0;
183 }
```

Listing 5.2: An OpenCAL-OMP implementation of the SciddicaT debris flows simulation model.

5.3 SciddicaT with active cells optimization

Here we present an OpenCAL-OMP implemenation of SciddicaT, which takes advantage of the built-in OpenCAL active cells optimization. You can find the complete source code in Listing 5.3, while the corresponding serial implementation can be found in previous Chapter.

```
1  // The SciddicaT debris flows model with the active cells optimization
2
3  #include <OpenCAL-OMP/cal2D.h>
```

```
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <OpenCAL-OMP/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct sciddicaTSubstates {
24   struct CALSubstate2Dr *z;
25   struct CALSubstate2Dr *h;
26   struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
27 } Q;
28
29 struct sciddicaTParameters {
30   CALParameterr epsilon;
31   CALParameterr r;
32 } P;
33
34
35 // The sigma_1 elementary process
36 void sciddicaT_transition_function(struct CALModel2D* sciddicaT, int i,
       int j)
37 {
38   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
39   CALbyte again;
40   CALint cells_count;
41   CALreal average;
42   CALreal m;
43   CALreal u[5];
44   CALint n;
45   CALreal z, h;
46   CALreal f;
47
48
49   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
50   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
51   for (n=1; n<sciddicaT->sizeof_X; n++)
52   {
53     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
54     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
55     u[n] = z + h;
56   }
57
58   //computes outflows and updates debris thickness
```

```
59    do{
60      again = CAL_FALSE;
61      average = m;
62      cells_count = 0;
63
64      for (n=0; n<sciddicaT->sizeof_X; n++)
65        if (!eliminated_cells[n]){
66          average += u[n];
67          cells_count++;
68        }
69
70        if (cells_count != 0)
71          average /= cells_count;
72
73        for (n=0; n<sciddicaT->sizeof_X; n++)
74          if( (average<=u[n]) && (!eliminated_cells[n]) ){
75            eliminated_cells[n]=CAL_TRUE;
76            again=CAL_TRUE;
77          }
78
79    }while (again);
80
81    for (n=1; n<sciddicaT->sizeof_X; n++)
82      if (eliminated_cells[n])
83        calSet2Dr(sciddicaT, Q.f[n-1], i, j, 0.0);
84      else
85      {
86        calSet2Dr(sciddicaT, Q.f[n-1], i, j, (average-u[n])*P.r);
87        calAddActiveCellX2D(sciddicaT, i, j, n);
88      }
89  }
90
91  // The sigma_2 elementary process
92  void sciddicaT_width_update(struct CALModel2D* sciddicaT, int i, int j)
93  {
94    CALreal h_next;
95    CALint n;
96
97    h_next = calGet2Dr(sciddicaT, Q.h, i, j);
98    for(n=1; n<sciddicaT->sizeof_X; n++)
99      h_next +=  calGetX2Dr(sciddicaT, Q.f[NUMBER_OF_OUTFLOWS - n], i, j,
           n) - calGet2Dr(sciddicaT, Q.f[n-1], i, j);
100
101   calSet2Dr(sciddicaT, Q.h, i, j, h_next);
102 }
103
104 // The sigma_3 elementary process
105 void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
      i, int j)
106 {
107   if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
108     calRemoveActiveCell2D(sciddicaT,i,j);
109 }
110
111
112 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
113 {
```

```
114   CALreal z, h;
115   CALint i, j;
116
117   //sciddicaT parameters setting
118   P.r = P_R;
119   P.epsilon = P_EPSILON;
120
121   //initializing substates to 0
122   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
123   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
124   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
125   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
126
127   //sciddicaT source initialization
128   for (i=0; i<sciddicaT->rows; i++)
129     for (j=0; j<sciddicaT->columns; j++)
130     {
131       h = calGet2Dr(sciddicaT, Q.h, i, j);
132
133       if ( h > 0.0 ) {
134         z = calGet2Dr(sciddicaT, Q.z, i, j);
135         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
136
137         //adds the cell (i, j) to the set of active ones
138         calAddActiveCell2D(sciddicaT, i, j);
139       }
140     }
141 }
142
143 // SciddicaT steering function
144 void sciddicaTSteering(struct CALModel2D* sciddicaT)
145 {
146   // set flow to 0 everywhere
147   calInitSubstate2Dr(sciddicaT, Q.f[0], 0);
148   calInitSubstate2Dr(sciddicaT, Q.f[1], 0);
149   calInitSubstate2Dr(sciddicaT, Q.f[2], 0);
150   calInitSubstate2Dr(sciddicaT, Q.f[3], 0);
151 }
152
153
154 int main()
155 {
156   time_t start_time, end_time;
157
158   // define of the sciddicaT CA and sciddicaT_simulation simulation
        objects
159   struct CALModel2D* sciddicaT = calCADef2D (ROWS, COLS,
        CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
        CAL_OPT_ACTIVE_CELLS);
160   struct CALRun2D* sciddicaT_simulation = calRunDef2D(sciddicaT, 1,
        STEPS, CAL_UPDATE_IMPLICIT);
161
162   //put OpenCAL - OMP in unsafe state execution(to allow unsafe
        operation to be used)
163   calSetUnsafe2D(sciddicaT);
164
165   // add transition function's sigma_1 and sigma_2 elementary processes
```

```
166   calAddElementaryProcess2D(sciddicaT, sciddicaT_transition_function);
167   calAddElementaryProcess2D(sciddicaT, sciddicaT_width_update);
168   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
          ;
169
170   // add substates
171   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
172   Q.h = calAddSubstate2Dr(sciddicaT);
173   Q.f[0] = calAddSubstate2Dr(sciddicaT);
174   Q.f[1] = calAddSubstate2Dr(sciddicaT);
175   Q.f[2] = calAddSubstate2Dr(sciddicaT);
176   Q.f[3] = calAddSubstate2Dr(sciddicaT);
177
178   // load configuration
179   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
180   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
181
182   // simulation run
183   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
184   calRunAddSteeringFunc2D(sciddicaT_simulation, sciddicaTSteering);
185   printf ("Starting simulation...\n");
186   start_time = time(NULL);
187   calRun2D(sciddicaT_simulation);
188   end_time = time(NULL);
189   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
190
191   // saving configuration
192   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
193
194   // finalizations
195   calRunFinalize2D(sciddicaT_simulation);
196   calFinalize2D(sciddicaT);
197
198   return 0;
199 }
```

Listing 5.3: An OpenCAL-OMP implementation of the SciddicaT debris flows simulation model with the active cells optimization.

With respect to the Sciddica implementation in Listing 5.2, which is exclusively based on safe OpenCAL-OMP operations, the active cells management, as implemented here, requires an unsafe operation. Such an unsafe operation is performed by means of the `calAddActiveCellX2D()` function (line 87), which adds a cell belonging to the neighbourhood to the set A of active cells. Such an operation both breasks the formal definition of Omogeneous Cellular Automanta, and also can give rise to race condition. In fact, if more threads try to add the same cell to the set A at the same time, being this a non-atomic operation, the result can be wrong. In order to avoid this possible error, OpenCAL-OMP is able to *lock* the memory locations involved in the operations so that each thread can entirely perform its own task, without the risk other threads interfere. In order to do that, it is sufficient to put OpenCAL-OMP in *unsafe* state by calling the `calSetUnsafe2D()`, as done al line 163. No other modifications to the serial source code are required.

5.4 SciddicaT as eXtended CA

Here we present an OpenCAL-OMP implemenation of SciddicaT, which takes advantage of the built-in OpenCAL unsafe operations. They represents an extension of the Omogeneous Cellular Automata definition, allowing for further computational optimizations. Using unsafe operation, give rise to an eXtended CA, as described in the previous chapter. You can find the complete source code of SciddicaT implemented as an eXtended CA in Listing 5.4, while the corresponding serial implementation can be found in Listing 4.10, in previous Chapter.

```
1  // The SciddicaT further optimized CCA debris flows model
2
3  #include <OpenCAL-OMP/cal2D.h>
4  #include <OpenCAL-OMP/cal2DIO.h>
5  #include <OpenCAL-OMP/cal2DRun.h>
6  #include <OpenCAL-OMP/cal2DUnsafe.h>
7  #include <stdlib.h>
8  #include <time.h>
9
10 // Some definitions...
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct CALModel2D* sciddicaT;
24
25 struct sciddicaTSubstates {
26   struct CALSubstate2Dr *z;
27   struct CALSubstate2Dr *h;
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37
38 // The sciddicaT transition function
39 void sciddicaT_flows(struct CALModel2D* sciddicaT, int i, int j)
40 {
41   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
42   CALbyte again;
43   CALint cells_count;
44   CALreal average;
```

```
45    CALreal m;
46    CALreal u[5];
47    CALint n;
48    CALreal z, h;
49    CALreal f;
50
51
52    m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53    u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54    for (n=1; n<sciddicaT->sizeof_X; n++)
55    {
56      z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57      h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58      u[n] = z + h;
59    }
60
61    //computes outflows and updates debris thickness
62    do{
63      again = CAL_FALSE;
64      average = m;
65      cells_count = 0;
66
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81
82    }while (again);
83
84    for (n=1; n<sciddicaT->sizeof_X; n++)
85      if (!eliminated_cells[n])
86      {
87        f = (average-u[n])*P.r;
88        calAddNext2Dr(sciddicaT,Q.h,i,j,-f);
89        calAddNextX2Dr(sciddicaT,Q.h,i,j,n,f);
90
91        //adds the cell (i, j, n) to the set of active ones
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
        i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
100     calRemoveActiveCell2D(sciddicaT,i,j);
```

```
101 }
102
103
104 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
123         //adds the cell (i, j) to the set of active ones
124         calAddActiveCell2D(sciddicaT, i, j);
125       }
126     }
127 }
128
129
130 int main()
131 {
132   time_t start_time, end_time;
133
134   // define of the sciddicaT CA and sciddicaT_simulation simulation
        objects
135   sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
        CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
136   sciddicaT_simulation = calRunDef2D(sciddicaT, 1, STEPS,
        CAL_UPDATE_IMPLICIT);
137
138   //put OpenCAL - OMP in unsafe state execution(to allow unsafe
        operation to be used)
139   calSetUnsafe2D(sciddicaT);
140
141
142   // add transition function's sigma_1 and sigma_2 elementary processes
143   calAddElementaryProcess2D(sciddicaT, sciddicaT_flows);
144   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
        ;
145
146   // add substates
147   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
148   Q.h = calAddSubstate2Dr(sciddicaT);
149
150   // load configuration
151   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
152   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
```

```
153
154   // simulation run
155   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
156   printf ("Starting simulation...\n");
157   start_time = time(NULL);
158   calRun2D(sciddicaT_simulation);
159   end_time = time(NULL);
160   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
161
162   // saving configuration
163   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
164
165   // finalizations
166   calRunFinalize2D(sciddicaT_simulation);
167   calFinalize2D(sciddicaT);
168
169   return 0;
170 }
```

Listing 5.4: An OpenCAL-OMP implementation of the SciddicaT debris flows eXtended CA simulation model with unsafe optimization.

Note that, only the topographic altitude and the debris thicness are now considered as model's substates (lines 25-28, 147-148), as the four outflows substates are no longer needed. Moreover, the number of elementary process now considered is two (lines 143-144), instead of three for the previous versions of SciddicaT.

The call to the `calSetUnsafe2D()` function (line 139) puts OpenCAL-OMP in unsafe state, allowing to lock memory locations (i.e. cells) that can be simultaneously accessed by more threads. In order to lock a cell, you have to use spacific functions, which are provided by the `OpenCAL-OMP/cal2DUnsafe.h` header file (line 6). In the specific case, besides `calAddActiveCellX2D()`, the `calAddNext2Dr()` and `calAddNextX2Dr()` functions are employed (lines 88-89). In fact, a combination of get-set operations, as done in the corresponding serial implementation (Listing 4.10, can not be considerd. In fact, let considered the snippet of code in Listing 5.5. As you can see, for each not-eliminated cell, the algorithm computes a flow, f (line 5), and then subtract it from the central cell (line 6) and add it to the corresponding neigbour (line 7), in order to perform the mass balance. In both cases (flow subtraction and adding), a flawor of `calGet` function is called to read the current value of the Q_h substate at the next working plane. Subsequently, a flawor of `calSet` function is used to update the previously read value. When a single thread is used to perform such operations, no race conditions can occur. At the contrary, even in the case of two concurrent threads, different undesirable situations can take place, which give rise to wrong resutls. For instance, let suppose both the threads read the value first, and then they write their updated values; in this case, the resulting value will correspond to the one written by the thread that write the value for last, and the contribution of the other thread will be lost.

In order to avoid such kind of problems whed dealing with more threads, the above mentioned `calAddNext2Dr()` and `calAddNextX2Dr()` functions lock

```
1   // <snip>
2   for (n=1; n<sciddicaT->sizeof_X; n++)
3   if (!eliminated_cells[n])
4   {
5     f = (average-u[n])*P.r;
6     calSet2Dr (sciddicaT,Q.h,i,j,  calGetNext2Dr (sciddicaT,Q.h,i,j)  -
          f);
7     calSetX2Dr(sciddicaT,Q.h,i,j,n,calGetNextX2Dr(sciddicaT,Q.h,i,j,n)+
          f);
8     // <snip>
9   }
10  // <snip>
```

Listing 5.5: Example of non atomic operation made of a combination of get-set calls.

the cell under condideration and then perform the get-set oprations. In this way, is ensured that only a thread can work at the same time and no race conditions can occur. Obviously, this can give rise to a lack of performance.

5.5 SciddicaT with explicit simulation loop

As for the serial version, aslo for the OpenMP based realease of OpenCAL is further possibile to improve computational performance of SciddicaT by avoiding unnecessary substates updating.

As we already stated, the `calRun2D()` function used so far to run the simulation loop updates all the defined substates at the end of each elementary process. However, in the specific case of the SciddicaT XCA model, no substates updating should be executed after the application of the second elementaty process, as it just remove inactive cells from the set A.

A new OpenCAL implementation of SciddicaT is presented in Listing 5.6. It is based on an explicit global transition function, defined by means of `calRunAddGlobalTransitionFunc2D()`. registers a callback functions within which you can both reorder the sequence of elementary processes to be applyed in the generic computational step, and also chouse which substates have to be updated. It also explicitates the simulation loop and also defines a stopping criterion for the simulation termination. The complete implementation is shown for the sake of completeness.

```
1   // The SciddicaT further optimized CCA debris flows model
2
3   #include <OpenCAL-OMP/cal2D.h>
4   #include <OpenCAL-OMP/cal2DIO.h>
5   #include <OpenCAL-OMP/cal2DRun.h>
6   #include <OpenCAL-OMP/cal2DUnsafe.h>
7   #include <stdlib.h>
8   #include <time.h>
9
10  // Some definitions...
```

```
11 #define ROWS 610
12 #define COLS 496
13 #define P_R 0.5
14 #define P_EPSILON 0.001
15 #define STEPS 4000
16 #define DEM_PATH "./data/dem.txt"
17 #define SOURCE_PATH "./data/source.txt"
18 #define OUTPUT_PATH "./data/width_final.txt"
19 #define NUMBER_OF_OUTFLOWS 4
20
21 // declare CCA model (sciddicaT), substates (Q), parameters (P),
22 // and simulation object (sciddicaT_simulation)
23 struct CALModel2D* sciddicaT;
24
25 struct sciddicaTSubstates {
26   struct CALSubstate2Dr *z;
27   struct CALSubstate2Dr *h;
28 } Q;
29
30 struct sciddicaTParameters {
31   CALParameterr epsilon;
32   CALParameterr r;
33 } P;
34
35 struct CALRun2D* sciddicaT_simulation;
36
37
38 // The sciddicaT transition function
39 void sciddicaT_flows(struct CALModel2D* sciddicaT, int i, int j)
40 {
41   CALbyte eliminated_cells[5]={CAL_FALSE,CAL_FALSE,CAL_FALSE,CAL_FALSE,
         CAL_FALSE};
42   CALbyte again;
43   CALint cells_count;
44   CALreal average;
45   CALreal m;
46   CALreal u[5];
47   CALint n;
48   CALreal z, h;
49   CALreal f;
50
51
52   m = calGet2Dr(sciddicaT, Q.h, i, j) - P.epsilon;
53   u[0] = calGet2Dr(sciddicaT, Q.z, i, j) + P.epsilon;
54   for (n=1; n<sciddicaT->sizeof_X; n++)
55   {
56     z = calGetX2Dr(sciddicaT, Q.z, i, j, n);
57     h = calGetX2Dr(sciddicaT, Q.h, i, j, n);
58     u[n] = z + h;
59   }
60
61   //computes outflows and updates debris thickness
62   do{
63     again = CAL_FALSE;
64     average = m;
65     cells_count = 0;
66
```

```
67      for (n=0; n<sciddicaT->sizeof_X; n++)
68        if (!eliminated_cells[n]){
69          average += u[n];
70          cells_count++;
71        }
72
73        if (cells_count != 0)
74          average /= cells_count;
75
76        for (n=0; n<sciddicaT->sizeof_X; n++)
77          if( (average<=u[n]) && (!eliminated_cells[n]) ){
78            eliminated_cells[n]=CAL_TRUE;
79            again=CAL_TRUE;
80          }
81
82    }while (again);
83
84    for (n=1; n<sciddicaT->sizeof_X; n++)
85      if (!eliminated_cells[n])
86      {
87        f = (average-u[n])*P.r;
88        calAddNext2Dr(sciddicaT,Q.h,i,j,-f);
89        calAddNextX2Dr(sciddicaT,Q.h,i,j,n,f);
90
91        //adds the cell (i, j, n) to the set of active ones
92        calAddActiveCellX2D(sciddicaT, i, j, n);
93      }
94  }
95
96
97  void sciddicaT_remove_inactive_cells(struct CALModel2D* sciddicaT, int
        i, int j)
98  {
99    if (calGet2Dr(sciddicaT, Q.h, i, j) <= P.epsilon)
100     calRemoveActiveCell2D(sciddicaT,i,j);
101 }
102
103
104 void sciddicaT_simulation_init(struct CALModel2D* sciddicaT)
105 {
106   CALreal z, h;
107   CALint i, j;
108
109   //sciddicaT parameters setting
110   P.r = P_R;
111   P.epsilon = P_EPSILON;
112
113   //sciddicaT source initialization
114   for (i=0; i<sciddicaT->rows; i++)
115     for (j=0; j<sciddicaT->columns; j++)
116     {
117       h = calGet2Dr(sciddicaT, Q.h, i, j);
118
119       if ( h > 0.0 ) {
120         z = calGet2Dr(sciddicaT, Q.z, i, j);
121         calSetCurrent2Dr(sciddicaT, Q.z, i, j, z-h);
122
```

```
123           //adds the cell (i, j) to the set of active ones
124           calAddActiveCell2D(sciddicaT, i, j);
125         }
126       }
127
128     calUpdateActiveCells2D(sciddicaT);
129   }
130
131
132   void sciddicaTransitionFunction(struct CALModel2D* sciddicaT)
133   {
134     // active cells must be updated first becouse outflows
135     // have already been sent to (pheraps inactive) the neighbours
136     calApplyElementaryProcess2D(sciddicaT, sciddicaT_flows);
137       calUpdateActiveCells2D(sciddicaT);
138       calUpdateSubstate2Dr(sciddicaT, Q.h);
139
140     // here you don't need to update Q.h
141     calApplyElementaryProcess2D(sciddicaT,
          sciddicaT_remove_inactive_cells);
142     calUpdateActiveCells2D(sciddicaT);
143   }
144
145
146   CALbyte sciddicaTSimulationStopCondition(struct CALModel2D* sciddicaT)
147   {
148     if (sciddicaT_simulation->step >= STEPS)
149       return CAL_TRUE;
150     return CAL_FALSE;
151   }
152
153   void run(struct CALRun2D* simulation)
154   {
155     CALbyte again;
156
157     calRunInitSimulation2D(simulation);
158
159     do{
160         again = calRunCAStep2D(simulation);
161         simulation->step++;
162     } while (again);
163
164     calRunFinalizeSimulation2D(simulation);
165   }
166
167   int main()
168   {
169     time_t start_time, end_time;
170
171     // define of the sciddicaT CA and sciddicaT_simulation simulation
          objects
172     sciddicaT = calCADef2D (ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
          CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS);
173     sciddicaT_simulation = calRunDef2D(sciddicaT, 1, CAL_RUN_LOOP,
          CAL_UPDATE_EXPLICIT);
174
175     //put OpenCAL - OMP in unsafe state execution(to allow unsafe
```

T version	Serial [s]	1thr	2thr	4thr	6thr	8thr
naive	240s	0.82 (293s)	1.22 (196s)	1.53 (157s)	1.64 (146s)	1.6 (150s)
active cells	23s	0.77 (30s)	1.36 (17s)	1.77 (13s)	2.09 (11s)	2.3 (10s)
eXtended CA	13s	0.77 (17s)	1.86 (7s)	2.6 (5s)	2.17 (6s)	2.6 (5s)
explicit loop	12s	0.75 (16s)	1.2 (10s)	2.4 (5s)	2.4 (5s)	3.0 (4s)

Table 5.1: Speedup of the four different implementations of the SciddicaS3hex debris flows model accelerated by OpenMP.

```
        operation to be used)
176   calSetUnsafe2D(sciddicaT);
177
178   // add transition function's sigma_1 and sigma_2 elementary processes
179   calAddElementaryProcess2D(sciddicaT, sciddicaT_flows);
180   calAddElementaryProcess2D(sciddicaT, sciddicaT_remove_inactive_cells)
          ;
181
182   // add substates
183   Q.z = calAddSingleLayerSubstate2Dr(sciddicaT);
184   Q.h = calAddSubstate2Dr(sciddicaT);
185
186   // load configuration
187   calLoadSubstate2Dr(sciddicaT, Q.z, DEM_PATH);
188   calLoadSubstate2Dr(sciddicaT, Q.h, SOURCE_PATH);
189
190   // simulation run
191   calRunAddInitFunc2D(sciddicaT_simulation, sciddicaT_simulation_init);
192   calRunAddGlobalTransitionFunc2D(sciddicaT_simulation,
          sciddicaTransitionFunction);
193   calRunAddStopConditionFunc2D(sciddicaT_simulation,
          sciddicaTSimulationStopCondition);
194
195   printf ("Starting simulation...\n");
196   start_time = time(NULL);
197   run(sciddicaT_simulation);
198   end_time = time(NULL);
199   printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
          start_time);
200
201   // saving configuration
202   calSaveSubstate2Dr(sciddicaT, Q.h, OUTPUT_PATH);
203
204   // finalizations
205   calRunFinalize2D(sciddicaT_simulation);
206   calFinalize2D(sciddicaT);
207
208   return 0;
209 }
```

Listing 5.6: An OpenCAL-OMP implementation of the SciddicaT XCA debris flows simulation model with explicit simulation loop.

S3-hex version	Serial [s]	1thr	2thr	4thr	6thr	8thr
naive	1030s	0.52 (1982s)	0.9 (1142s)	1.03 (998s)	1.13 (913s)	1.3 (781s)
active cells	55s	0.86 (64s)	1.57 (35s)	2.75 (20s)	2.5 (22s)	3.06 (18s)
eXtended	27s	0.87 (31s)	1.42 (19s)	2.7 (10s)	2.46 (11s)	3.38 (8s)
explicit loop	16s	0.8 (20s)	1.33 (12s)	2.67 (6s)	2.29 (7s)	3.2 (5s)

Table 5.2: Speedup of the four different implementations of the SciddicaS3hex debris flows model accelerated by OpenMP.

mbusu version	Serial [s]	OpenMP 6th	OpenCL (Quadro FX 1100M)
naive	7796s	3.57 (2185s)	3.52 (2213s)

Table 5.3: mbusu Speedup.

5.6 A three-dimensional example

Threads	Elapsed time [s]	Speedup
1	7308	1
2		
4		
6	2185	3.34
8		

Table 5.4: mbusu Speedup.

Chapter 6

OpenCAL OpenCL version

Chapter 7

OpenCAL OpenGL version