 http://dx.doi.org/10.1109/PDP59025.2023.00045

OpenCAL++: An object-oriented architecture for transparent Parallel Execution of Cellular Automata models

Andrea Giordano*, Donato D'Ambrosio†, Davide Macrì *,
Rocco Rongo†, Gladys Utrera‡, Marisa Gil‡, William Spataro†
*ICAR-CNR, Rende, Italy
† University of Calabria, Department of Mathematics and Computer Science, Italy
‡Universitat Politècnica de Catalunya. BarcelonaTECH, Spain

***Abstract*—Cellular Automata (CA) models, initially studied by John von Neumann, have been developed by numerous researchers and applied in both academic and scientific fields. Thanks to their local and independent rules, simulations of complex systems can be easily implemented based on CA modelling on parallel machines. However, due to the heterogeneity of the components - from the hardware to the software perspective-the various possible scenarios running parallelism in today's architectures can pose a challenge in such implementations, making it difficult to exploit. This paper presents OpenCAL++, a transparent and efficient object-oriented platform for the parallel execution of cellular automata models. The architecture of OpenCAL++ ensures the modeller a fully transparent parallel execution and a strong "separation of concerns" between the execution parallelism issues and the model implementation. The code implementing the Cellular Automata model remains the same whether the execution performs in a shared-, distributed-memory or a GPGPU context, irrespective of the optimizations adopted. To this aim, the object-oriented paradigm has been intensely exploited. As well as the OpenCAL++ architecture, we present the description of a simple Cellular Automata model implementation for illustrative purposes.**

***Index Terms*—Parallel Computing, Cellular Automata, Modelling and Simulation**

I. Introduction

Complex systems modeling is in general a very compute-intensive task as their formalization usually relies on intractable differential equations. However, tasks that are involved in complex systems simulation can strongly benefit of alternative approaches in which the system is approximated by its decomposition in many interacting (simple) entities. Among these methods, Cellular Automata (CA) are one of the first parallel computing abstract models and have proved to be particularly suitable for systems whose behavior can be described in terms of local interactions [1]. Originally studied by John von Neumann to study self-reproduction problems [2], CA models have been developed by numerous researchers and applied in both theoretical and scientific fields (e.g., [3], [4], [5], [6], [7]), [8]). Thanks to their local and independent rules, simulations of complex systems that are based on CA modeling can be easily implemented on parallel machines. Even though Parallel computing [9] has undoubtedly proved its effectiveness in many application scenarios [10], overheads can arise due to the parallelization process itself, that can reduce the obtainable benefits (e.g., [11], [12], [13], [14]). To reduce this overhead, different strategies have been envisioned ([15], [16]. Moreover, the low-level implementations of CA execution must be devised for each parallel execution context, such as shared memory (e.g., OpenMP) and distributed memory (e.g., MPI) architectures and modern GPGPU (e.g., CUDA or OpenCL). The choice of the suitable execution context based on hardware availability is a key factor for providing dramatic computational improvements in computational results. On the other side, parallel programming requires strong technical expertise, let alone considering the different parallel execution contexts and adopted optimization strategies. Indeed, different high level CA modeling APIs were proposed in the literature in attempting to mitigate these issues. However, in general, these solutions [17]–[19] lack a full portability across different execution contexts and parallelization strategies requiring the model program to be adapted from case to case.

In this article, we present OpenCAL++, a platform for transparent and efficient parallel execution of CA models. Differently from the aforementioned solutions, OpenCAL++ makes parallelism transparent to the modeller and addresses many aspects of the underlying formal computational paradigms and optimization strategies. Moreover, it implements a set of load balancing strategies to accelerate the computation. The adoption of a fully object-oriented approach enables the full transparency of the code and a "plug-and-play" feature that allows to easily insert new parallel optimization strategies and new parallel execution contexts.

The paper is organized as follows. In Section II the parallel execution of CA models is discussed, while Section III presents optimization techniques for CA parallel execution. Section IV describes the OpenCAL++ platform object-oriented aspects, while Section V shows a simple

CA implementation for illustrative purposes. Eventually, conclusions and future developments are given in Section VI.

II. Parallel Execution of Cellular Automata

The Cellular Automata (CA) computational paradigm can be easily adopted to model and simulate complex systems characterized by a high number of interacting elementary components. Due to their implicit parallel nature, CAs can be productively parallelized across multiple parallel machines to scale and speed up their execution. Execution of CA on both sequential and parallel computers consists in a step-by-step evaluation of the transition function for each cell of the cellular space. In what follows, implementation issues of the parallel execution of CA referred to the three main parallel execution contexts are summarized.

A. Distributed memory

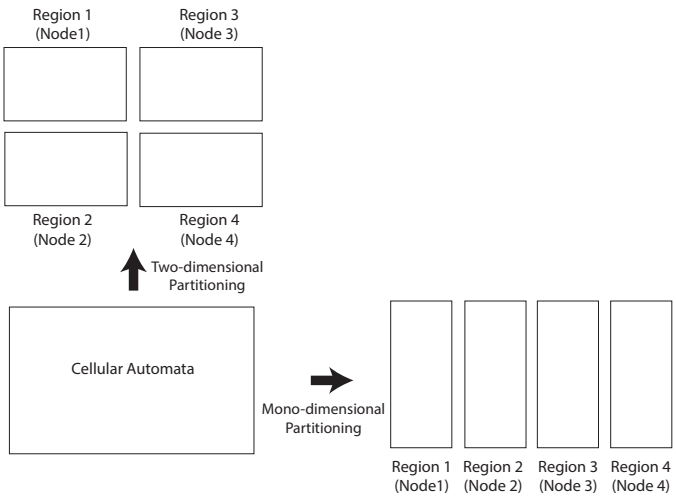

Fig. 1: Distributed context: the cellular space is partitioned into regions associated with parallel computing nodes. Two alternative types of partitioning are shown: uni-dimensional and two-dimensional.

The parallelization of CA execution on distributed memory systems can be efficiently achieved by partitioning the initial cellular space into different regions (or territories), which are assigned to the different processing elements (node) (e.g., [12], [20], [8]), as shown in Figure 1.

Each node is responsible for executing the transition function of all the cells belonging to the region it manages. As previously stated, the computation of a transition function of a cell is based on the states of the cell's neighbourhood. When a cell is located at the edge of a region, its neighbourhood can overlap more regions (see Figure 2). Hence, the transition function execution of these cells requires information belonging to adjacent computing nodes. For this reason, in a distributed memory context, the states of border cells (often called *halo* cells in CA literature) need to be exchanged, at each computing step, among adjacent nodes in order to keep the parallel execution consistent. The border area of a region (the halo cells) is divided into two different sub-areas: the *local border* and the mirror border (see Figure 3). The local border is managed by the local node and its content is replicated in the mirror border of the adjacent node.

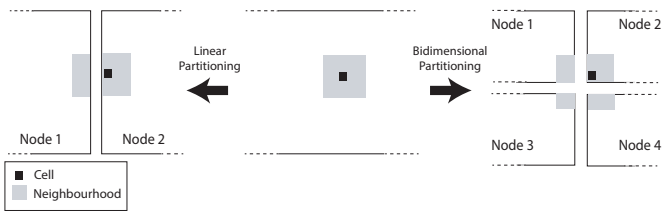

Fig. 2: A cell's neighbourhood overlapping more regions.

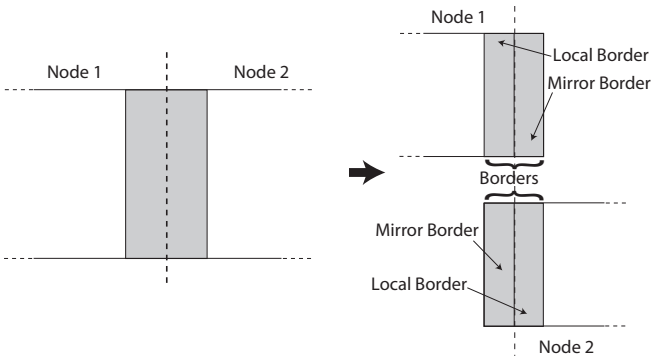

Fig. 3: Border areas of two adjacent nodes in the case of uni-dimensional partitioning

To summarize, the parallel execution of a CA model consists in each parallel node executing the following execution loop:

Algorithm 1: CA parallel execution

```
1 for each step do
2     SendBorder()
3     ReceiveBorder()
4     for each cell C do
5         N=getNeighbourhood(C)
          C-newState=transitionFunction(C,N)
6     swapReadWriteMatrices()
```

B. Shared memory

The parallel execution of CA in a shared memory context can also exploit space partitioning as for the case of distributed memory one. In this case, though, the processing elements are now parallel thread, instead of the computing nodes, each managing a different portion of CA space, while halo border exchanges are no longer needed since each thread has access to the entire memory. As a consequence, the parallel loop for the shared memory case

is the same as in Algorithm 1 except for the `SendBorder()` and `ReceiveBorder()` calls.

C. GPGPU execution

The GPGPU paradigm enables an efficient parallel execution of CA models by assigning each cell to a different GPGPU thread ([21]). The CA space domain is initially transferred from the host memory to the device (global) memory. Afterwards, no host-memory transfers, which is a typical bottleneck of GPGPU computations, are further required. A further improvement of computational performances can be obtained by exploiting the device shared memory [22]. In addition, especially when the CA domain is particularly large, a multi-gpu approach can be considered, consisting in different GPGPU devices communicating among each other. In both shared memory and multi-gpu approaches, the space partitioning and border exchange methodology described for the distributed memory case (see section II-A) can also efficiently exploited.

III. Parallel Execution Optimization strategies

The execution time of a parallel application executed on N nodes, T_N , can be expressed as [23]:

$$T_N = \frac{T_1}{N} + T_O = \frac{T_1}{N} + T_{comm} + T_{idle} + T_{conf} \quad (1)$$

where T_1 is the sequential time, i.e., the time to execute the computation on a single node, and T_O is the overhead time added by the parallelization process itself. The overhead time can be seen as the sum of three main contributions related, respectively, to the time needed for setting up and transmitting data between nodes (T_{comm}), the idle time experienced by faster nodes that need to wait for slower nodes (T_{idle}) and the time needed to solve conflicts, i.e., manage contentions on shared data (T_{conf}). It is easy to realize that the time needed to solve conflicts can be considered negligible in the case of parallel execution of CA, while T_{comm} and T_{idle}, related to the communication and synchronization burden, respectively, can strongly impact on the overall performance resulting in poor scalability. The optimization strategies presented in what follows are aimed for improving parallel performances by attempting to reduce, to different extents, the communication burden and the synchronization burden.

A. Communication/Computation interleaving strategy

The Communication/Computation interleaving [24] strategy aims to reduce the synchronization burden by moving the transition function computation of a large set of cells between the `SendBorder()` and `ReceiveBorder()` operations (see Algorithm 1). This is possible by considering that only cells located at the edge part of a region actually have neighbour cells falling into the halo border area, and so the transition function for all the other cells can be computed *before* receiving borders from neighbour regions.

Moving the transition function computation before the `ReceiveBorder()` operation results in having the exchange of halo borders and the transition function computation occurring at the same time, enhancing the degree of parallelism of the system and, consequently, the performances.

B. Multi-border strategy

The Multi-border strategy [24] reduces the number of synchronization points by adding some redundant computation. In particular, the exchange of borders is carried out each k steps, with $k > 1$ (instead of each step). At each border exchange operation, k borders are actually exchanged. After this multi-border exchange operation, each region has the border located at the more external part working as halo replica as usual, and so the transition function for all the cells can be computed, *including* the remaining borders. In this way, the second more external border can now work as halo replica for the third more external border, and so forth. After k steps the k border has to be exchanged again.

C. Look-ahead strategy

As previously stated, in the CA context synchronization among different nodes is mandatory at each computational step in order to maintain consistency. The aim of the Look-ahead strategy [16] is to relax the synchronization burden by exploiting the so-called lookahead concept, which is derived from the Discrete Event Simulation research field [25]. In particular, the algorithm reduces the number of border exchange phases, which otherwise would be required at each computational step, by computing at runtime the (minimum) number of CA steps that a node can carry out without affecting the border cells that are of interest of adjacent nodes.

D. Load Balancing strategy

One of the worst source of performance degradation in parallel systems is due to the synchronization burden related to the not balanced workload assignment to the processing elements. In such a scenario, the least loaded processing elements have to wait the most loaded ones. This is particularly significant in CA execution scenario where the processing elements need to synchronize each other at each step. In addition, in CA modelling natural phenomenon, such as lava flow, fire spreading, and so forth, the unbalanced workload condition comes directly from the unpredictability of the natural phenomenon evolution resulting in different regions of the space domain being more affected by a given physical condition (e.g. presence of lava, fire, water, etc.). In the Load Balancing (LB) strategy [26], the computational load is dynamically (and optimally) balanced among the parallel processing elements during the simulation.

IV. The OpenCAL++ platform

The OpenCAL++ is a C++ platform for transparent and efficient parallel execution of cellular automata models. A cellular automata model executed on the OpenCAL++ platform is made up of 4 main software entities: the *cell* class, the *model* object, the *space* object and the *engine* object. The first two entities are those of interest for the cellular automata modeller, while the latter two are OpenCAL++ own objects defining how the parallel execution must be carried out. The architecture of OpenCAL++ ensures a full transparent parallel execution and a strong "separation of concerns" between the parallel execution issues and the model implementation. Roughly speaking, OpenCAL++ allows the modeller to be completely unaware of the parallel execution context, that is, the code implementing the cellular automata model remains the same regardless if the parallel execution is based on a shared or distributed memory context, and regardless of the parallel execution optimization adopted, such as the multi-border strategy or load balancing. To this aim, the object-oriented paradigm has been strongly exploited. In particular, four "abstract" base classes are envisioned defining the basic features of the 4 main software involved entities. Such entities have to be implemented by defining subclasses of the four base classes. In the following, a brief description of such base classes is reported.

- **Cell** class: This is the base class that represents the status of a single cell of the cellular automata. For instance, in a landslide CA model, a subclass of `Cell` could host the status of the landslide (i.e., the debris amount and the altitude) in a generic point of space. The specific `Cell` subclass is used to define the other relevant classes through the C++ template feature. For example, in a landslide model supposing the Cell subclass is called `LandslideState`, the other base classes are specified as follows: `Model2D<LandslideState>`, `Space2D<LandslideState>`, `Engine2D<LandslideState>`. The abstract function of Cell must be implemented by subclasses in order to specify how to serialize/deserialize the status of a cell (for file writing purposes) and how to visualize it in terms of RGB coloring. This issue will be better clarified in the following (see Section IV-A and V).

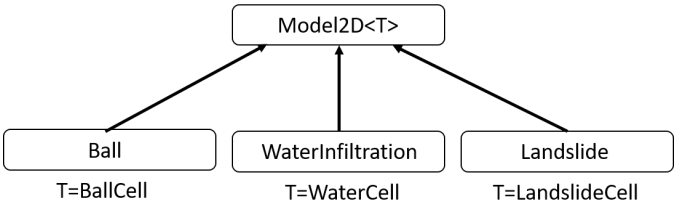

Fig. 4: Examples of Model2D subclasses.

- **Model2D** class: This is the base class for the definition of the CA model. The modeller is in charge of defining a subclass of Model2D by implementing its abstract function `init` and `transitionFunction`. The `init` implementation defines the initialization of the CA, while the `transitionFunction` implementation defines the generic cell transition function at a generic CA time step. The code of `init` and `transitionFunction` consists in regular C/C++ code where the reading and writing operations on CA domain space transparently occur by means of a specific high level API supplied by the `Space2D` base class. An OpenCAL++ application thus consists of a subclass of `Model2D`, furtherly specified by a subclass of Cell as template class (see Figure 4).

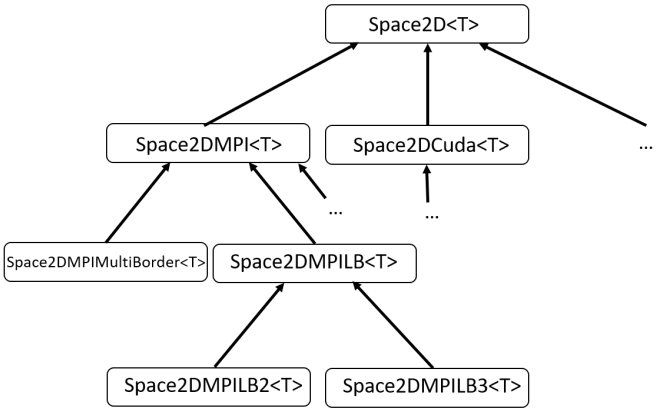

Fig. 5: Space2D subclasses hierarchy.

- **Space2D** class: This is the ancestor of all the classes managing the CA domain. Each subclass of Space2D implements all the abstract functions of Space2D that represent an API to be used by the modeller to read/write the CA domain. The Space2D subclasses are in charge of implementing all the technicalities of the specific parallel paradigm and optimization strategy they are referred to. As a consequence, these subclasses are naturally arranged in a hierarchical manner (see Figure 5). For example, `Space2DMpi` is a direct subclass of `Space2D` and is in charge of managing the halo border exchange by MPI message exchanges, as typically required in a distributed memory context. `Space2DMpiLB` is a subclass of `Space2DMpi` class and it adds, to the `Space2DMpi` features, the code required for achieving load balancing. It implements the basic strategy for load balancing, while its subclasses, `Space2DMpiLB2` and `Space2DMpiLB3`, refer to more advanced load balancing strategies.
- **Engine2D** class. Its subclasses are in charge of implementing the basic control loop for the specific paradigm/strategy adopted.

A. The Visualizer module

OpenCAL++ comes with a built in tool for "offline" visualization. In particular, during the parallel execution, the Space2D subclasses can optionally dump the state of

all the CA space domain into a file at each step. Each computing node writes its own file. After the parallel execution is terminated, the OpenCAL++ visualizer is in charge of reading the output files produced by all the nodes in order to "reconstruct" the CA space and then visualize it. The Cell virtual methods have been devised specifically for these purposes. In particular, the `stringEncoding` method has to be implemented in `Cell` subclasses in order to supply a string representation of the state of a cell. This method will be called by the `Space2D` subclasses when it has to write the CA space domain into a file. The method `composeCell` does the opposite operation reconstructing the state of a cell starting from a cell state string representation. Finally, the method `outputValue` returns an RGB color based on the cell state. The visualizer reconstructs the whole CA space domain by reading from the output files of all the nodes and properly calling `composeCell` for each cell. Afterwards, all cells are displayed on the screen based on the returned values of `outputValue`. In the following, we introduce a simple example of OpenCAL++ CA model that possibly better clarifies these issues.

V. A simple CA model Implementation: The Ball CA Model

In this section, we introduce a straightforward CA model example consisting in a "ball" moving through space. In particular, the CA cell state can be either 'active' or 'inactive'. The CA initial configuration concerns of a set of 'active' cells, that are the ones falling inside a given circle, surrounded by all the other cells set to be 'inactive'. The transition function, applied to a generic cell c, simply copies the state of a neighbour cell, i.e., the one in the opposite direction w.r.t. the moving direction on the state of the cell c. The overall effect is to move the entire ball of active cells throughout the CA space domain along the given moving direction. This simple CA model can be used as a "Hello World" test for a CA execution platform (as OpenCAL++). However, it can be also very useful in reproducing a load imbalanced condition when the computational load changes dynamically during the advancement of the CA execution.

In the following, the code implementing the Ball CA model using OpenCAL++ is reported. As previously stated, the code of a OpenCAL++ model is made up of 2 main parts: a subclass of the OpenCAL++ base class `Cell`, and a subclass of the OpenCAL++ base class `Model2D` where the subclass of `Cell` is used as template class. First, let's have a look at the `Cell` subclass code:

```
#ifndef BallCell_H
#define BallCell_H

#include "../../Cell.h"

rgb outputColor(0, 0, 0);

class BallCell : public Cell
{
private:
    int state;
public:

    BallCell(int state)
    {
        this->state = state;
    }

    void setState(int s)
    {
        state = s;
    }

    int getState()
    {
        return state;
    }

    void composeCell(char *str)
    {
        this->state = atoi(str);
    }

    char *stringEncoding()
    {
        char *zstr = new char[1];
        sprintf(zstr, "%d", state);
        return zstr;
    }

    rgb *outputValue()
    {
        if (state == 0)
            outputColor = rgb(255, 0, 0);
        if (state == 1)
            outputColor = rgb(0, 0, 0);

        return &outputColor;
    }


};
#endif
```

One can easily see that the cell state is just an integer variable, 'state', hosting the value 0 for an inactive cell and 1 for an active one. The method `stringEncoding` converts this state variable to a one-character string for output dumping purposes. The method `composeCell` does the opposite operation reconstructing the state variable starting from a one-character string. Finally, the method `outputValue` returns an RGB color to be used in the visualization phase by assigning the red color for active cells and the black color for the inactive ones.

The code below implements the subclass of `Model2D`:

```
#include "ballCell.h"
```

```
#include "../../Model2D.h"
#include "dummy.h"
#include <math.h>


int centerX;
int centerY;
int radious;

int dirX;
int dirY;

const int dummyIteration = 100;

class Ball : public Model2D<BallCell>
{

public:
    void init()
    {
        RangeCoord d = space->getDimension();
        for (int y = d.minY; y < d.maxY; y++)
        {
            for (int x = d.minX; x < d.maxX; x++)
            {
                int state = 0;
                if (sqrt(pow(x - centerX, 2) +
                         pow(y - centerY, 2)) < radious)
                    state = 1;

                BallCell c(state);
                space->initCell(x, y, c);
            }
        }
    }

    void transitionFunction(int x, int y)
    {
    BallCell cNeigh = space->getCell(x - dirX, y - dirY);
    BallCell c = space->getCell(x, y);

    if(c.getState() == 1)
    c.setState(cNeigh.getState());
    space->setCell(x, y, c);
    }


};
```

Notice that the previously illustrated `BallCell` is used as a template class. The `init` function supplies the CA initial configuration by setting, as active cells, the cells falling inside the circle defined by `centerX`, `centerY` and `radious`. The setting of these latter variables, not reported here for brevity issues, is carried out in the `main` function where also the `space` object is created and linked the to `Ball` class, and specifically to its variable `space`, which can be used for reading/writing the space through the well-defined `Space2D` API. In particular, in the `init` method, the `getDimension` space method is used for retrieving the CA space dimension, and the `initCell` is used to initialize a CA space cell. The `transitionFunction` uses the space methods `getCell` and `setCell` for copying the `neighbour` cell state in the cell for which the transition function is called. It is important to emphasize here that both the `BallCell` and `Ball` codes do not contain any reference to the specific execution context or optimization strategy adopted. Thus, the same code can be used, for example, with the load balancing feature or multiborder strategy without even changing a line of code. The last part worth of discussion is the 'main' function, i.e., the code where the execution begins. In the main function, all the relevant objects are created and linked together and the parallel execution is started. This is the only part of the code where the "user" has to indicate what execution context/strategy they want to exploit.

In the code below, one can see a main function code example:

```
int main(int argc, char *argv[])
{

    int infoFromFile[8];
    char *outputFileName = new char[256];
    readConfigurationFile(infoFromFile, outputFileName);

    int dimX = infoFromFile[0];
    int dimY = infoFromFile[1];
    int borderSizeX = infoFromFile[2];
    int borderSizeY = infoFromFile[3];
    int numBorders = infoFromFile[4];
    int nNodeX = infoFromFile[5];
    int nNodeY = infoFromFile[6];
    int nsteps = infoFromFile[7];

    int stepLB = 100;

    Space2DMpiLB<BallCell> space(dimX, dimY, borderSizeX,
        borderSizeY, nNodeX, nNodeY, nsteps, outputFileName);

    Ball ball;
    ball.setSpace(&space);

    Engine2DMPILB<BallCell> engine;
    engine.setModel(&ball);
    engine.setSpace(&space);
    engine.setNsteps(nsteps);
    engine.setStepLB(stepLB);
    engine.start();

    delete[] outputFileName;

    return 0;
}
```

In this example, the chosen `Space2D` subclass is the one that also implements load balancing, `Space2DMpiLB`, and the same applies for the `Engine2D` subclass (`Engine2DMPILB`). For both `Space2DMpiLB` and `Engine2DMPILB`, the `BallCell` class is used as a template class. Once all the objects are created, they need to be linked together. In particular, the space object is linked to the model object (`ball`), and both the space and the model objects are linked to the engine object that is finally used to start the CA execution through the calling of its start method.

VI. Conclusions

In this paper, we present the first release of OpenCAL++, an object-oriented platform for the parallel execution of cellular automata. In OpenCAL++, the parallel execution context is entirely transparent to the modeller, who can concentrate on the code development regardless of whether the execution runs on a shared-, distributed-memory or GPGPU context. Moreover, OpenCAL++ can transparently exploit some optimizations, such as Load Balancing features, which have proven to be proper support for speeding up simulations.

Future work will extend the platform to multi-dimensional models (e.g., 3D) besides enhancing the multi-GPU version, which can fully exploit the computational power of current parallel machines. OpenCAL++ is currently freely available on GitHub at https://github.com/alessioderango/oopencal

Acknowledgements

The authors gratefully acknowledge Alessio De Rango from the Department of Environmental Engineering of the University of Calabria (Italy) for his support in the implementation and testing phases of the OpenCAL++ library.

This research was funded by the Italian "ICSC National Center for HPC, Big Data and Quantum Computing" Project, CN00000013 (approved under the Call M42C – Investment 1.4 – Avvisto "Centri Nazionali" – D.D. n. 3138 of 16.12.2021, admitted to financing with MUR Decree n. 1031 of 06.17.2022)

References

[1] A. De Rango, L. Furnari, A. Giordano, A. Senatore, D. D'Ambrosio, W. Spataro, S. Straface, and G. Mendicino, "Opencal system extension and application to the three-dimensional richards equation for unsaturated flow," *Computers and Mathematics with Applications*, vol. 81, pp. 133–158, 2021.

[2] J. von Neumann, *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[3] S. Wolfram, "Universality and complexity in cellular automata," *Physica D*, no. 10, pp. 1–35, 1984.

[4] C. Aidun and J. Clausen, "Lattice-boltzmann method for complex flows," *Annual Review of Fluid Mechanics*, vol. 42, pp. 439–472, 2010.

[5] V. Ntinas, B. Moutafis, G. Trunfio, and G. Sirakoulis, "Parallel fuzzy cellular automata for data-driven simulation of wildfire spreading," *Journal of Computational Science*, 2016.

[6] M. Macri, A. De Rango, D. Spataro, D. D'Ambrosio, and W. Spataro, "Efficient lava flows simulations with opencl: A preliminary application for civil defence purposes," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015, pp. 328–335.

[7] L. Furnari, A. Senatore, A. De Rango, M. De Biase, S. Straface, and G. Mendicino, "Asynchronous cellular automata subsurface flow simulations in two- and three-dimensional heterogeneous soils," *Advances in Water Resources*, vol. 153, p. 103952, 2021.

[8] A. De Rango, D. Spataro, W. Spataro, and D. D'Ambrosio, "A first multi-gpu/multi-node implementation of the open computing abstraction layer," *Journal of Computational Science*, vol. 32, pp. 115–124, 2019.

[9] V. Kumar, "Introduction to parallel computing, 2nd ed," *Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.,*, 2002.

[10] M. Macri, A. Rango, D. Spataro, D. D'Ambrosio, and W. Spataro, "Efficient lava flows simulations with opencl: A preliminary application for civil defence purposes," *Proceedings - 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2015*, pp. 328–335, 2015.

[11] A. Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 3, pp. 12–21, 1993.

[12] F. Cicirelli, A. Forestiero, A. Giordano, and C. Mastroianni, "Parallelization of space-aware applications: Modeling and performance analysis," *Journal of Network and Computer Applications*, vol. 122, pp. 115–127, 2018.

[13] J. Was, H. Mróz, and P. Topa, "Gpgpu computing for microscopic simulations of crowd dynamics," *Computing and Informatics*, vol. 34, no. 6, pp. 1418–1434, 2016.

[14] I. Gerakakis, P. Gavriilidis, N. I. Dourvas, I. G. Georgoudas, G. A. Trunfio, and G. C. Sirakoulis, "Accelerating fuzzy cellular automata for modeling crowd dynamics," *Journal of Computational Science*, vol. 32, pp. 125–140, 2019.

[15] A. Giordano, A. De Rango, D. D'Ambrosio, R. Rongo, and W. Spataro, "Strategies for parallel execution of cellular automata in distributed memory architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 406–413.

[16] A. Giordano, D. D'Ambrosio, A. De Rango, A. Portaro, W. Spataro, and R. Rongo, "Exploiting distributed discrete-event simulation techniques for parallel execution of cellular automata," in *Artificial Life and Evolutionary Computation*, F. Cicirelli, A. Guerrieri, C. Pizzuti, A. Socievole, G. Spezzano, and A. Vinci, Eds. Cham: Springer International Publishing, 2020, pp. 66–77.

[17] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia, "A parallel cellular automata environment on multicomputers for computational science," *Parallel Computing*, vol. 21, no. 5, pp. 803–823, 1995.

[18] G. Spingola, D. D'Ambrosio, W. Spataro, R. Rongo, and G. Zito, "Modeling complex natural phenomena with the libautoti cellular automata library: An example of application to lava flows simulation." in *PDPTA*, 2008, pp. 277–283.

[19] D. D'Ambrosio, A. De Rango, M. Oliverio, D. Spataro, W. Spataro, R. Rongo, G. Mendicino, and A. Senatore, "The open computing abstraction layer for parallel complex systems modeling on many-core systems," *Journal of Parallel and Distributed Computing*, vol. 121, pp. 53–70, 2018.

[20] A. Giordano, A. De Rango, D. Spataro, D. D'Ambrosio, C. Mastroianni, G. Folino, and W. Spataro, "Parallel execution of cellular automata through space partitioning: the landslide simulation sciddicas3-hex case study," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2017, pp. 505–510.

[21] P. Renc, T. Pecak, A. De Rango, W. Spataro, G. Mendicino, and J. Was, "Towards efficient gpgpu cellular automata model implementation using persistent active cells," *Journal of Computational Science*, vol. 59, p. 101538, 2022.

[22] D. Spataro, D. D'Ambrosio, G. Filippone, R. Rongo, W. Spataro, and D. Marocco, "The new sciara-fv3 numerical model and acceleration by gpgpu strategies," *The International Journal of High Performance Computing Applications*, vol. 31, no. 2, pp. 163–176, 2017.

[23] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency: measuring the scalability of parallel algorithms and architectures," *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, no. 3, pp. 12–21, 1993.

[24] A. Giordano, A. De Rango, D. D'Ambrosio, R. Rongo, and W. Spataro, "Strategies for parallel execution of cellular automata in distributed memory architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 406–413.

[25] B. Zeigler, Y. Moon, D. Kim, and G. Ball, "The devs environment for high-performance modeling and simulation," *IEEE*

Computational Science and Engineering, vol. 4, no. 3, pp. 61–71, 1997.

[26] A. Giordano, A. De Rango, R. Rongo, D. D'Ambrosio, and W. Spataro, "Dynamic load balancing in parallel execution of cellular automata," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 470–484, 2020.