Cellular Automata on a Multi-GPU Architecture: A Technical Overview

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

***Abstract*—This work is focused on the transparent execution of Cellular Automata models on a multi-GPU architecture. Although Cellular Automata models can be easily parallelized on a single GPU, the domain size and transition function complexity may require the use of multiple GPUs. Our goal is to allow modellers to be completely unaware of the parallel execution context, i.e., the code implementing the Cellular Automata model remains the same regardless if the execution is performed on CPU, single GPU, or multi-GPU systems. This paper supplies meaningful technical insights on how to ensure both transparency and efficiency in multi-GPU execution of Cellular Automata models. In particular, an object-oriented approach is exploited in which a transparent layer is devised that abstracts the parallelization details and allows a strong "separation of concerns" between the execution parallelism issues and the model implementation. Preliminary experiments have been carried out on the multi-GPU cluster CTE-POWER available at the Barcelona Supercomputing Center (BSC), witnessing good speedups notwithstanding the transparency feature supplied by our approach.**

***Index Terms*—Multi-GPU 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], [9]). Thanks to their local and independent rules, simulations of complex systems that are based on CA modeling can be profitably implemented on parallel machines. Despite this, 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). In particular, GPGPU technology provides unprecedented computational capabilities paving the way for applicative scenarios previously considered to be intractable. As a matter of fact, GP-GPU technology strongly outperforms CPU-based approaches in software applications characterized by an inherent high degree of parallelism as in the case of cellular automata. In this case, thanks to the GPGPU many-cores architecture, it is possible to profitably assign different threads to different CA cells. However, When a typical one-cell-one-thread implementation is no longer feasible in a single GPU Card due, for instance, to particularly spatial-extended CA models, a Multi-GPU approach can be considered. In this case, GPU devices interact with each other by intra-node communication or even by inter-node network communications. The multi-GPU approach, thus, has to consider either the GPGPU peculiarities but also distributed computing issues. In addition, the CPU resources can be also exploited jointly with GPU facilities. In conclusion, a multi-GPU approach has to take into account together both shared/distributed memory and GPGPU issues. The choice of a suitable execution context based on hardware availability is a key factor for providing dramatic computational improvements in computational results. In literature, the parallel execution of cellular automata has been deeply studied and different solutions have been proposed in attempting to mitigate issues emerging from parallel programming in the different execution contexts [10]–[12]. Nevertheless, these solutions lack a fully portable execution across the different execution contexts and so the CA modeller is required to have low-level programming expertise in order to adapt the model program from case to case.

In this article, we present an approach for transparent and efficient parallel execution of CA models in a multi-GPU execution context. In particular, we present technical insights on how to make parallelism transparent to the modeller by addressing the aspects of the underlying computational paradigms thanks to the adoption of a fully object-oriented approach.

 https://dx.doi.org/10.1109/PDP62718.2024.00042

The paper is organized as follows. In the next Section II the parallel execution of CA models is discussed, Section III presents the proposed multi-gpu implementation for CA execution, while Section IV presents the achieved experimental results. Eventually, conclusions and future developments are given in Section V.

II. Transparent Parallel Execution of Cellular Automata

The inherent parallelism of CA models stems from their decentralized nature, where individual cells interact based on local rules, facilitating concurrent computations. This feature results in a full-fledged exploitation of the parallel processing capabilities of modern computing architectures, such as in the Multi-GPU case. The adoption of a multi-GPU strategy involves issues referred both to GPGPU programming and distributing computing. The latter comes to the forefront when considering the distribution of computational tasks across multiple GPUs requiring careful orchestration of communication, synchronization, and load balancing. In addition to the GPU-centric aspects, the integration of Central Processing Unit (CPU) resources into the multi-GPU paradigm introduces another layer of complexity and optimization opportunities. Leveraging CPU capabilities alongside GPU facilities can result in a balanced utilization of computational resources, enhancing overall system performance. The necessity of adopting different parallel execution contexts asks for expertise for each one of the parallel execution contexts, and this goes along with the specific model-wise expertise required to design the cellular automata. This complexity can be reduced by supplying the CA modeller with a software platform for transparent and efficient parallel execution. In this scenario, the modeller is completely unaware of the parallel execution context and has to focus only on the cellular automata design.

For instance, let's assume we want to design a simple CA model 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.

A possible high-level implementation, which hides the underlying parallel programming issues, can be the one illustrated in the following code snippet:

```
class Ball : public Model2D<BallCell>{
    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);
        c.setState(cNeigh.getState());
        space->setCell(x, y, c);
    }
};
```

As evident, the code implements the particular CA model by defining its behaviour through the `transitionFunction` and `init` functions implementation without any reference to the specific issues related to the different parallel execution contexts. The underlying software platform allows to transparently execute this code without the need to adapt it to the chosen execution context. This is achieved by leveraging object-oriented programming. Indeed, as can be seen in the code, the model is implemented by defining the `Ball` class, which inherits the `Models2D` platform class using the C++ template features, as will be better clarified in the following. The `space` object is used to read/write the CA spatial domain. In particular, the `initCell` method is used in the `init` function for setting the initial value of the cells, i.e., setting the 1 value for cells falling in the circle defined by `centerX`, `centerY` and `radious`[1]. The `getCell`/`setCell` methods are used in the `transitionFunction` function for read and write the cell status, respectively, i.e., for moving the "ball" by simply copying the state of the neighbour cell located in the opposite direction with respect to the moving direction (`dirX`,`dirY`).

A. The Platform Software Architecture

The underlying software platform, which enables the transparent parallel execution described so far, 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 platform objects defining how the parallel execution must be carried out. The architecture of the platform ensures a fully transparent parallel execution and a strong "separation of concerns" between the parallel execution issues and the model implementation. 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 involved software entities. Such entities have

[1]The `centerX`, `centerY` and `radious` declarations are here omitted for code brevity.

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>`.

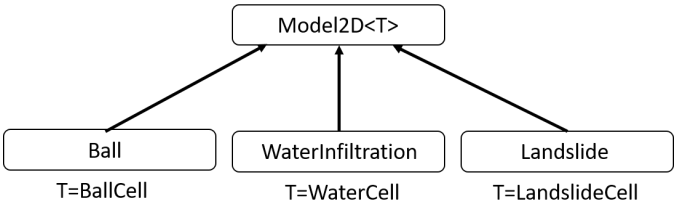

Fig. 1: Examples of Model2D subclasses.

- **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. A model implementation thus consists of a subclass of `Model2D`, further specified by a subclass of Cell as template class (see Figure 1).

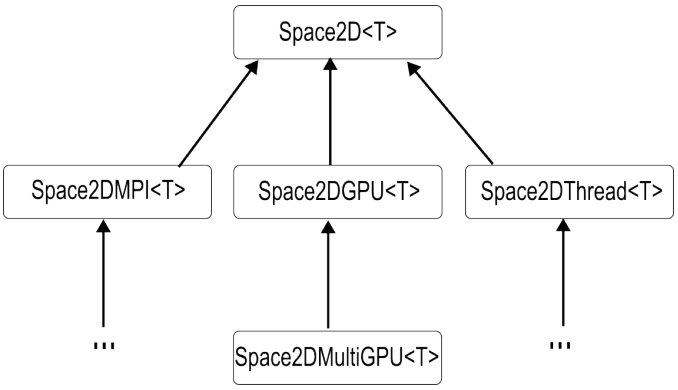

Fig. 2: Space2D subclasses hierarchy.

- **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 they are referred to. As a consequence, these subclasses are naturally arranged in a hierarchical manner (see Figure 2). 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. `Space2DMultiGPU` extends `Space2DGPU` by supplying the required distributed memory features.
- **Engine2D** class. Its subclasses are in charge of implementing the basic control loop for the specific adopted execution context.

B. Virtual Function Overhead

The Cell, Model, Space, and Engine objects robustly embrace the C++ Virtual Function mechanism. This mechanism facilitates dynamic binding among C++ objects, providing flexibility in the execution of functions across different contexts. However, it is essential to acknowledge that this flexibility introduces a slight overhead, due to the necessity of accessing the virtual function table and especially to the inherent inability to exploit the C++ inline function features effectively.

The impact of this overhead becomes particularly noteworthy when considering frequent virtual function calls, as occurs in our case. Indeed, the CA execution is characterized by the repeated call of the `transitionFunction` function, which, in turn, results in many calls to `getCell`, `setCell` functions.

However, by making further considerations of the actual system requirements, it turns out that a static binding mechanism could be enough to fulfill our needs. Indeed, the choice of the parallel execution context and the model to be executed can occur at compile-time and so the dynamic aspects of virtual functions are indeed unnecessary.

In order to implement the required static binding mechanism, we resort to the so-called *curiously recurrent template pattern* (CRTP).

In the CRTP technique, whose name was coined by Jim Coplien in 1995 [13], there are a base and a set of derived classes, as for virtual functions, for which a polymorphism feature must be ensured. Actually, there is a function defined in the Base class that is also redefined in the derived classes. The polymorphism feature, in this case, thus consists in choosing the *right* function implementation among the ones of the derived classes. For example, let's consider the following C++ code excerpt where the polymorphism feature is achieved through the typical virtual function feature:

```
class Base{
```

```
    virtual void func()=0;
};

class Derived1: public Base{
    void func(){
    ...
    }
};

class Derived2: public Base{
    void func(){
    ...
    }
};

void F(Base* b){
    b->func();
}
```

As known, the `func` call in `F` function is dynamically binded to the specific implementation of the `Derived1`/`Derived2` class by the virtual function feature, based on which of these classes was used in instantiating the object passed to the `F` function.

By using CRTP, we are able to reproduce such a mechanism at compile time, thus providing a *static polymorphism*. In CRTP there is a base class taking a template parameter and each one of the derived classes uses itself as the template parameter for the base class. For example, let's consider the following code excerpt in which the polymorphism of the previous code example, is reproduced in a static (compile time) fashion:

```
template<class C>
class Base{
    void func(){
        ((C*)this->C::func();
    }
};

class Derived1: public Base<Derived1>{
    void func(){
    ...
    }
};

class Derived2: public Base<Derived2>{
    void func(){
    ...
    }
};

void F(Base* b){
    b->func();
}
```

As it can be seen, the `Base` class takes the `C` template class, which is intended to be the specific derived class to be used. Indeed, the polymorphism is achieved by casting itself to the specific derived class passed as template parameter.

III. Multi-GPU execution of Cellular Automata

In this section, the implementation of the parallel execution, both on a single GPU card and on a multi-GPU scenario, is introduced, which results in being transparent thanks to the platform and the concepts described in the previous section.

In Algorithm 1 the parallel execution of cellular automata is described for the case of a single GPU card. Firstly, the domain space, i.e., the `Space2D` object, and the `Model2D` object are transferred from host to device memory (line 1). Subsequently, for each step, the transition function is called to be executed on the device through the typical CUDA kernel invocation (line 3). Afterward, the read and write matrix pointers are swapped both on host and device memory[2]. Eventually, the final CA domain state is copied back to the host memory for visualization purposes (line 6).

Algorithm 1: CA parallel execution on GPU

```
1 CopyHostToDevice()
2 for each step do
3 |   <<<transFunc()>>>()
4 |   <<<swapPtr()>>>()
5 |   swapPtr()//Host
6 CopyDevToHost()
```

When the transition function kernel is executed, the model object interacts with the space object completely inside the device part, as can be appreciated in Figure 3, through the `getCell`/`setCell` space object methods calls.

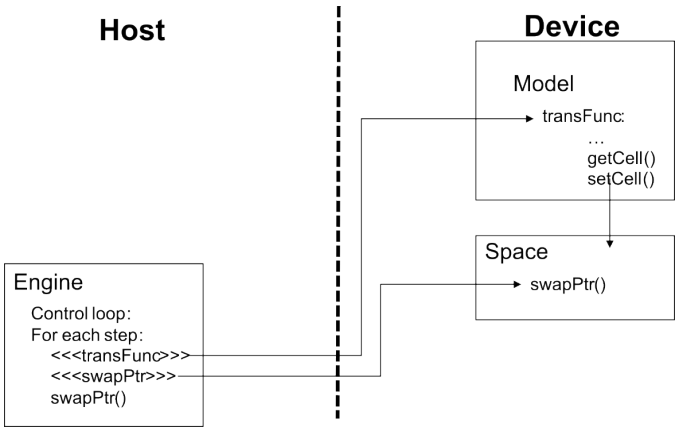

Fig. 3: CA Execution on a single GPU card.

In the multi-GPU execution, the CA domain is partitioned into different regions (or territories), which are assigned to the different nodes/GPU cards as occurs in a typical distributed memory scenario (e.g., [14], [15], [8]), as shown in Figure 4.

[2]The host and device read and write matrix pointers must be kept aligned throughout the execution for visualization purposes

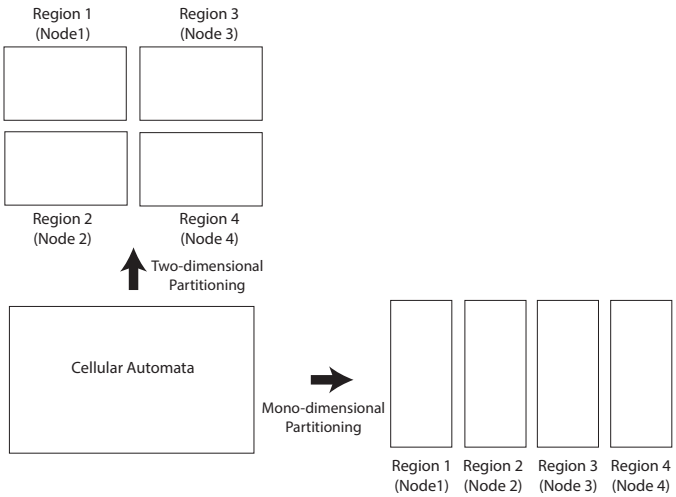

Fig. 4: Distributed context: the cellular space is partitioned into regions associated with parallel computing nodes. Two alternative types of partitioning are shown: uni-dimensional and two-dimensional.

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 5). Hence, the transition function execution of these cells requires information belonging to adjacent computing nodes. For this reason, 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 subareas: the *local border* and the mirror border (see Figure 6). The local border is managed by the local node and its content is replicated in the mirror border of the adjacent node.

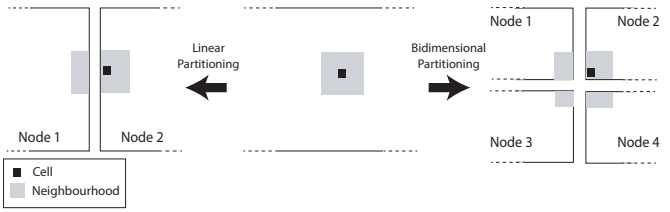

Fig. 5: A cell's neighbourhood overlapping more regions.

The border exchange in the multi-GPU context is managed as shown in Figure 7.

In particular, the local border content is first transferred from the device memory of the sender node to its host memory (CopyDevToHost). At this point, the actual network operation of exchanging the border takes place, e.g., by using an MPI send/receive operation. Finally, the receiving node can transfer the received content into

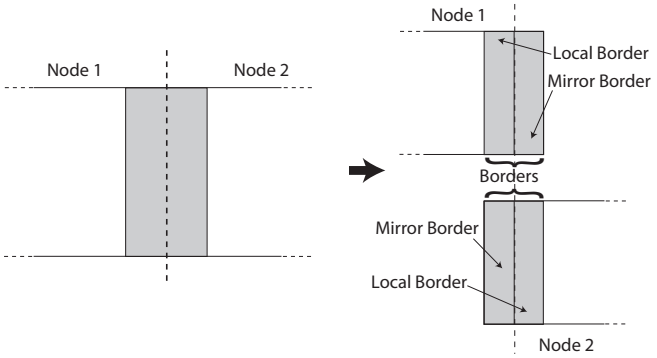

Fig. 6: Border areas of two adjacent nodes in the case of uni-dimensional partitioning

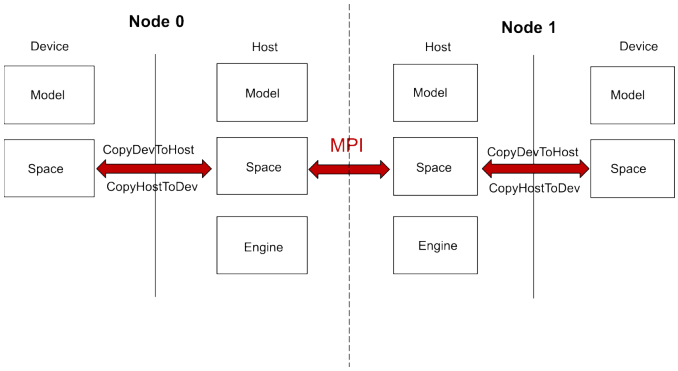

Fig. 7: Halo exchange in the CA Execution for a 2 nodes multi-GPU scenario.

the device memory (CopyHostToDev) to be located as a mirror border.

To summarize the CA parallel execution in the multi-GPU context follows the step described in Algorithm 2.

Algorithm 2: CA parallel execution on multi-GPU

```
1 CopyHostToDevice()//all the region
2 for each step do
3     CopyDevToHost()//border to send
4     ExchBorder()
5     CopyHostToDevice()//border received
6     <<<transFunc()>>>()
7     <<<swapPtr()>>>()
8     swapPtr()//Host
```

In line 1, the initial configuration of the CA is transferred from the host to the device memory, i.e., each node transfers its own portion of interest. The rest of the code is analogous to the one referred to single GPU case, except for the exchange border operation that consists in the sequence of (i) transferring the local border from device to host memory, (ii) network exchange of the borders, and (iii) copying back the received content to the mirror border area in the device memory.

TABLE I: Configuration parameters of the moving ball CA model

Parameter	Value
Total steps	8000
CA size	10000 × 10000 cells
Ball radius	125 cells

IV. Experimental results

To evaluate the performance of the Multi-GPU system, the quantitative experiment of the moving ball CA model was considered. The ball CA model consists of a set of active and inactive cells. The active cells are those cells falling inside a given ball area, while inactive cells are the remaining ones. The transition function is designed in order to move the ball horizontally over the domain. The configuration parameter of the moving ball CA model used for the experiments is illustrated in Table I: All these preliminary experiments were carried out on the cluster supplied by the Barcelona Supercomputing Center (BSC), called CTE-POWER, composed of 52 compute nodes, each equipped with 2 x IBM Power9 8335-GTH processors at 2.4GHz, an Infiniband interconnection network, and four Nvidia V100 GPUs. The speed-up achieved by the MultiGPU system of the moving ball CA model is shown in Fig. 8, demonstrating good speed-up and scalability despite the CA model code being written in a completely abstract form without the need to take care of any low-level technologies and paradigms thanks to the proposed transparent object-oriented approach.

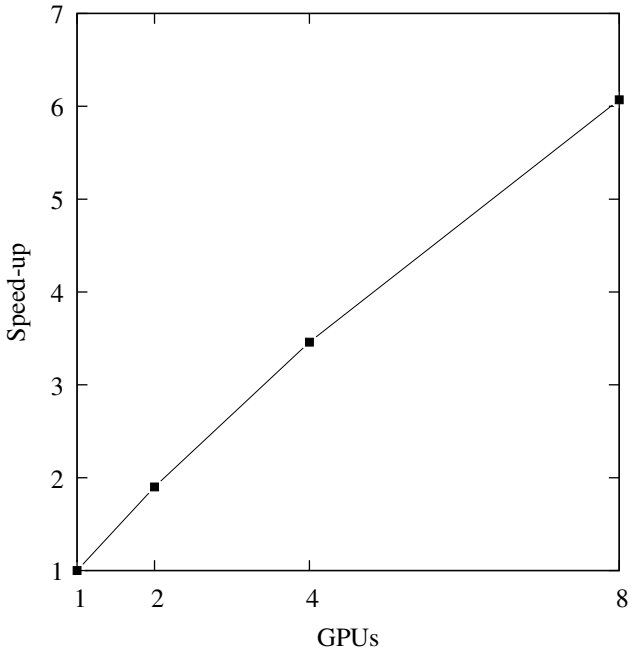

Fig. 8: Speed-up of the moving ball CA model using the MultiGPU system by considering up to 8 GPUs.

V. Conclusions

In this paper, an object-oriented platform for the multi-GPU parallel execution of cellular automata is presented. This platform ensures that the parallel execution context (single GPU, multi-GPU, or even CPU) is entirely transparent to the modeller, who can concentrate on the code development regardless of any technical issue related to the specific execution context.

Future work will extend the platform to multi-dimensional models (e.g. 3D) besides enhancing it with some optimization features, which can fully exploit the computational power of current parallel machines.

Acknowledgements

This research was partially funded by the Italian "ICSC National Center for HPC, Big Data and Quantum Computing" Project, CN00000013 (approved under the Call M42C – Investment 1.4 – Avviso "Centri Nazionali" - D.D. n. 3138 of 16.12.2021, admitted to financing with MUR Decree n. 1031 of 06.17.2022).
This research was partially funded by the Spanish Ministry of Education (PID2019-107255GB-C22) and the Generalitat de Catalunya (2021-SGR-01007).

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] A. De Rango, P. Napoli, D. D'Ambrosio, W. Spataro, A. Di Renzo, and F. Di Maio, "Structured grid-based parallel simulation of a simple dem model on heterogeneous systems," 2018, Conference paper, p. 588 – 595, cited by: 9.
[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] A. De Rango, L. Furnari, A. Giordano, A. Senatore, D. D'Ambrosio, S. Straface, and G. Mendicino, "Preliminary model of saturated flow using cellular automata," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11973 LNCS, p. 256 – 268, 2020, cited by: 7.
[10] 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.
[11] 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.

[12] 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.

[13] J. O. Coplien, "Curiously recurring template patterns," in *C++ gems*, 1996, pp. 135–144.

[14] 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.

[15] 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.