# A Dynamic Load Balancing Technique for Parallel Execution of Structured Grid Models

Andrea Giordano[1], Alessio De Rango[2], Rocco Rongo[3], Donato D'Ambrosio[3], and William Spataro[3](✉)

[1] ICAR-CNR, Rende, Italy
giordano@icar.cnr.it
[2] DIATIC, University of Calabria, Rende, Italy
alessio.derango@unical.it
[3] DEMACS, University of Calabria, Rende, Italy
{rocco.rongo,donato.dambrosio,william.spataro}@unical.it

**Abstract.** Partitioning computational load over different processing elements is a crucial issue in parallel computing. This is particularly relevant in the case of parallel execution of structured grid computational models, such as Cellular Automata (CA), where the domain space is partitioned in regions assigned to the parallel computing nodes. In this work, we present a dynamic load balancing technique that provides for performance improvements in structured grid model execution on distributed memory architectures. First tests implemented using the MPI technology have shown the goodness of the proposed technique in sensibly reducing execution times with respect to not-balanced parallel versions.

**Keywords:** Parallel computing · Parallel software tools · Load balancing · Cellular Automata

## 1 Introduction

The computational demands of complex systems simulation, such as in Computational Fluid Dynamic (CFD), are in general very compute intensive and can be satisfied only thanks to the support of advanced parallel computer systems. In the field of Modeling and Simulation (M&S), approximate numerical solutions of differential equations which rule a physical system (e.g., Navier-Stokes equations) are obtained by using parallel computers [4]. Classical approaches based on calculus (e.g., Partial Differential Equations - PDEs) often fail to solve these kinds of equations analytically, making a numerical computer-based methodology mandatory in case of solutions for real situations. Discretization methods, such as the Finite Element Method (FEM) or Finite Difference Method (FDM) (c.f., [7,9,14,16]), which estimate values at points over the considered domain, are often adopted to obtain approximate numerical solutions of the partial differential equations describing the system. Among these discrete numerical methodologies, Cellular Automata (CA) have proven to be particularly adequate for

Y. D. Sergeyev and D. E. Kvasov (Eds.): NUMTA 2019, LNCS 11973, pp. 278–290, 2020.
https://doi.org/10.1007/978-3-030-39081-5_25

systems whose behavior can be described in terms of local interactions. Originally studied by John von Neumann for studying self-reproducing issues [27], CA models have been developed by numerous researchers and applied in both theoretical and scientific fields (c.f., [2,10,13,15,21,24–26,29,30,38]).

Thanks to their rule locality nature and independently from the adopted formal paradigm, complex systems simulation based on the above numerical methodologies can be straightforwardly implemented on parallel machines. Typically, parallelizations based on OpenMP, MPI and the more recent GPGPU approach (e.g., with CUDA or OpenCL) have proven to be valuable solutions for efficient implementations of computational models (e.g., [1,3,18,19,31,32,36,37]). Nevertheless, computational layers based on higher level abstractions (i.e, by using a Domain Specific Language - DSL) of the adopted paradigm have been applied with success for simulating complex systems (e.g., [9,22,34]).

Parallel Computing applications require the best distribution of computational load over processing elements, in order to better exploit resources [23]. For instance, this is the case of CA which model the dynamics of topologically connected spatial systems (e.g., lava or debris flows), in which the evolution initially develops within a confined sub-region of the cellular space, and further expanding on the basis of the topographic conditions and source location(s) [6]. Indeed, when a data-parallel spatial decomposition is utilized, most computation can take place where the so-called “active” cells are located, with the risk of overloading the involved processing element(s). In this case, an effective load-balancing technique can mitigate this issue.

Load Balancing (LB) consists in partitioning the computation between processing elements of a parallel computer, to obtain optimal resource utilization, with the aim of reducing the overall execution time. In general, the particular parallel programming paradigm which is adopted and the interactions that occur among the concurrent task determine the suitability of a *static* or *dynamic* LB technique. Static LB occurs when tasks are distributed to the processing elements before execution, while dynamic LB refers to the case when the workload is dynamically distributed during the execution of the algorithm. More specifically, if the computation requirements of a task are known *a priori* and do not change during computation, a static mapping could represent the most appropriate choice. Nevertheless, if these are unknown before execution, a static mapping can lead to a critical imbalance, resulting in dynamic load balancing being more efficient. However, if the amount of data to be exchanged by processors produces elevated communication times (e.g., due to overhead introduced system monitoring, process migration and/or extra communication), a static strategy may outperform the advantages of the dynamic mapping, providing for the necessity of a static load balancing.

The simulation of complex physical phenomena implies the handling of an important amount of data. Several examples of both static and dynamic LB approaches can be find in literature for structured grid based models parallel implementations. In CAMEL [11] authors adopt a static load balancing strategy based on the scattered decomposition technique [28] which effectively can reduce the number of non-active cells per processor. A dynamic load-balancing algorithm is adopted in P-CAM [35]. P-CAM is a simulation environment based on a computational framework of interconnected cells which are arranged in graphs defining the cell interconnections and interactions. When a simulation starts, a decomposition of these graphs on a parallel machine is generated and a migration of cells occur thanks to a load balancing strategy. Another notable example based on a dynamic LB technique is presented in [39], and applied to an open-source parallel library for implementing Cellular Automata models, and provides for meaningful performance improvements for the simulation of topologically connected phenomena. Other recent examples can be found in [12] referred to multi-physics simulations centered around the lattice Boltzmann method, and in [17] where authors study LB issues in decentralized multi-agent systems by adopting a sandpile cellular automaton approach.

Accordingly, we here present an automatic domain detection feature that dynamically (and optimally) balances computational load among processing elements of a distributed memory parallel computer during a simulation. The paper is organized as follows. In Sect. 2 parallel decomposition techniques for CA models in distributed memory environments is discussed; subsequently, in Sect. 3, the proposed dynamic load balancing technique for CA is presented while preliminary experimental results are reported in Sect. 4. Eventually, a brief discussion and future developments section concludes the paper.

## 2 Parallel Execution of Cellular Automata in Distributed Memory Architectures

As anticipated, the Cellular Automata (CA) computational paradigm is particularly suitable for describing some complex systems, whose dynamics may be expressed in terms of local laws of evolution. In particular, CA can be fruitfully adopted to model and simulate complex systems that are characterized by an elevated number of interacting elementary components. Nevertheless, thanks to their implicit parallel nature, CA can be fruitfully parallelized on different parallel machines to scale and speed up their execution. A CA can be considered as a $d$-dimensional space (i.e., the cellular space), partitioned into elementary uniform units called cells, representing a finite automaton ($f_a$). Input for each $f_a$ is given by the cell's state and the state of a geometrical pattern, invariant in time and space over the cellular space, called the cell's neighbourhood. Two well-known examples of two-dimensional neighbourhoods are shown Fig. 1. At time $t = 0$, the CA initial configuration is defined by the $f_a$'s states and the CA subsequently evolves by applying the $f_a$'s *transition function*, which is simultaneously applied to each $f_a$ at each computational step. Despite their simplicity, CA can produce complex global behavior and are equivalent to Turing Machines from a computational viewpoint.

The CA execution on both sequential or parallel computers consists evaluating the transition functions over the cellular space. In this case, at step $t$, the

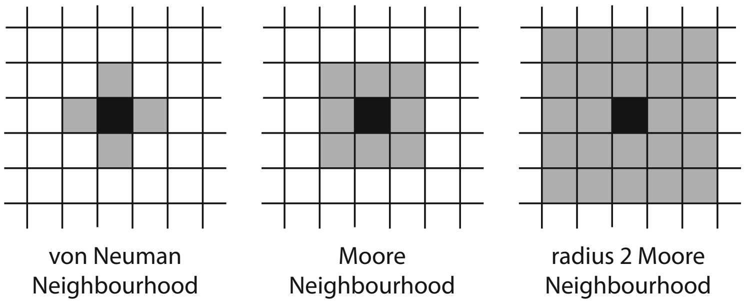

**Fig. 1.** CA neighbourhoods: von Neumann and Moore neighbourhoods are the left ones, respectively, both with visibility radius equal to 1, while a Moore neighbourhood with visibility radius equal to 2 is shown on the right.

evaluation of a cell transition function takes as input the state of neighbouring cells at step $t - 1$. This requires that the cell states at step $t - 1$ have to be stored in memory when executing step $t$, obtaining the so-called “maximum parallelism” behaviour. This issue can be straightforwardly implemented by using two matrices for the space state: the *read* and the *write* matrix. When executing of a generic step, the evaluation of the transition function is obtained by reading states from the read matrix and by then writing results to the write matrix. When all of the transition functions of cells have been evaluated, the matrices are swapped and the following CA step can take place.

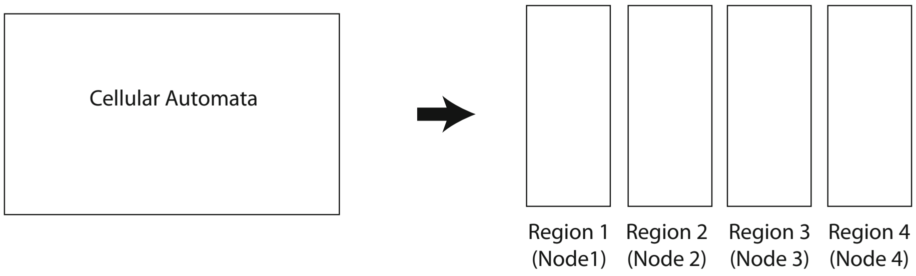

**Fig. 2.** Cellular space partitioning in regions that are assigned to different parallel computing nodes.

Being parallel computational models, CA execution can be efficiently parallelized by adopting a typical data-parallel approach, consisting in partitioning the cellular space in regions (or territories) and assigning each of then to a specific computing element [8,20] as shown in Fig. 2. Each region is assigned to a different computing element (node), which is in responsible of executing the transition function of all the cells belonging to that region. Since the computation of

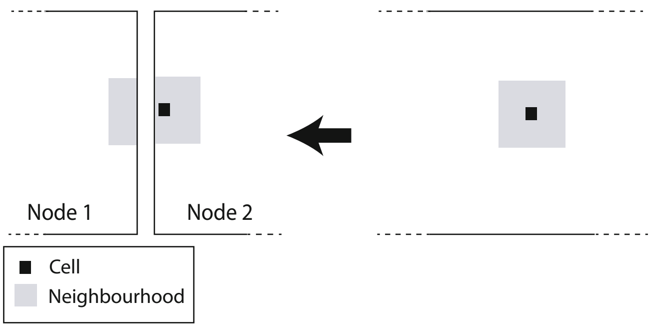

**Fig. 3.** The neighbourhood of a edge cell overlapping more regions

a transition function of a cell is based on the states of the cell's neighbourhood, this can overlap two regions for cells located at the edge part, as seen in Fig. 3.

Therefore, the transition function execution of cells in this area needs information that belongs to a adjacent computing node. As a consequence, in order to keep the parallel execution consistent, the states of these border cells (or *halo* cells) need to be exchanged among neighboring nodes at each computing step. In addition, the border area of a region (the *halo* cells) is divided in two different sub-areas: the *local border* and the *mirror border* (see Fig. 4). The local border is handled by the local node and its content replicated in the mirror border of the adjacent node.

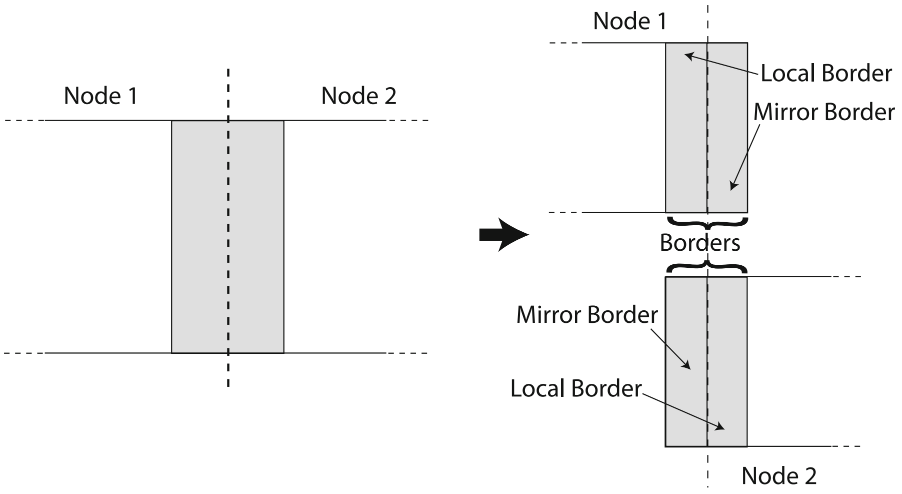

**Fig. 4.** Border areas of two adjacent nodes

The parallel execution of a CA in a distributed memory architecture consists in each node executing the loop illustrated in Algorithm 1. At each iteration, the node sends its halo borders to neighbour nodes, receiving in turn the corresponding halo cells (lines 3–4). Afterwards, the transition function is applied by reading input values of cells from the read matrix and updating the new result to the write matrix (i.e., recall the "maximum parallelism" behaviour described above). After all cells have been have updated, a swap between the read and write matrices occurs and the next CA step can be taken into account (line 7).

**Algorithm 1.** CA execution

```
1 while !StopCriterion()          // Loop until CA stop criterion met
2  do
3    SendBorderToNeighbours() // Send halo borders to left and right
        neighbour nodes
4    ReceiveBorderFromNeighbours() // Receive halo borders from left
        and right neighbour nodes
5    ComputeTransitionFunction() // Read from read matrix and write
        to write matrix
6    SwapReadWriteMatrices() // Swap read and write matrices
7    $step \leftarrow step + 1$                              // Next CA step
```

## 3 A Dynamic Load Balancing Technique for Partitioned Cellular Automata

The proposed LB algorithm consists in dynamically computing the optimal workload that each processing element has to take into account for achieving uniform execution times over a simulation. The approach presented in this paper relies on exchanging computation load among computing nodes at given steps on the basis of a specific criterion. For example, the balancing of the load can occur at a predefined rate of CA time steps, or when the execution times among nodes is particularly unbalanced, and so on. The load balancing is actually achieved by means of CA columns exchange among nodes. In particular, each node exchanges columns with its neighbour nodes in a parallel fashion by exploiting the same communication channel already in use for the halo exchange[1]. During a simulation, the execution times experienced by the nodes for executing the current step are retrieved and stored in each node. When the LB phase has to be executed, each node sends its step time to a specific "master" node, which is in charge of establishing a suitable columns exchanges that nodes must perform in order to achieve a balanced workload.

[1] For the sake of simplicity, in this first implementation of the LB procedure the exchange of columns is *not toroidal*, i.e. it is not possible to exchange columns between the rightmost node and the leftmost node.

**Algorithm 2.** CA execution with Dynamic Load Balancing

```
1  while !StopCriterion()          // Loop until CA stop criterion met
2  do
3      if LoadBalancingCriterion()         // Load Balancing step reached
4      then
5          SendMyExecInfo()        // send to Master my size and timings
6          SequenceOfFLows = ReceiveSeqFlows()     // Receive sequence of
             flows from Master
7          if IamLBMaster then
8              LBInfo=ReceiveExecInfo()      // Receive sizes and timings
                 from all nodes
9              newRegionSizes = LoadBalance(LBInfo)   // Determine new node
                 sizes
10             allSequenceFlows = ComputeFlows(NewRegionSizes)
                 // Determine sequence of flows for all nodes
11             SendFlows(allSequenceFlows)    // Send sequence of flows to
                 nodes
12         forall $flows \in SequenceOfFLows$ do
13             ExchangeLBFlows(flows)
       // Back to normal CA execution
14     SendBorderToNeighbours()
15     ReceiveBorderFromNeighbours()
16     ComputeTransitionFunction()
17     SwapReadWriteMatrices()
18     $step \leftarrow step + 1$                          // Next CA step
```

The new CA loop containing the code implementing the load balancing is described in Algorithm 2. Please note that the normal CA execution takes place at every step as reported at lines 14–18. As mentioned before, the LB is executed when some conditions are met, i.e., when `LoadBalancingCriterion()` is `true` (line 3). At line 5, each node sends its CA space size (i.e., the number of columns), along with the elapsed time required for computing the last CA step, to the master node (`SendMyExecInfo()`), and waits for receiving information from this latter on columns to be exchanged to achieve load balancing (line 6). In particular, this information (`SequenceOfFlows`) consists in a *sequence* of columns to be exchanged with the left and the right neighbours. The reason behind why a sequence of columns exchange is necessary, rather than just an simple exchange of a given number of columns to left and right nodes, will be clarified in the following. At line 12–13 the actual columns exchange takes place. In particular, each `flows` of the sequence, i.e, the columns to be exchanged with the left and right neighbour, is in practice applied through `ExchangeLBFlows()`.

Let us now summarize the master behaviour (lines 8–11). At line 8, the master receives information about the nodes state (i.e., space size and elapsed times) and determines, at line 9 (`LoadBalance()`), the new region sizes for the nodes, which minimize the unbalancing of the workload. On the basis of the new region sizes, the determination of the flows of columns that must be exchanged among nodes can be straightforwardly computed. However, it can be possible that some of the determined flows may exceed the columns availability of a given node. For instance, let us assume there are 3 nodes N1, N2 and N3, each having a 100 column CA space size right before the LB phase evaluation. Let us also assume that the `LoadBalance()` function computes new region sizes as 70, 20 and 210. In this case, in order to achieve 210 columns for N3, a flow of 110 columns should be sent from N2 to N3 (recall that it is not possible to exchange columns between N3 and N1, as reported in Sect. 2), though N2 hosts only 100 columns. In this example, this issue can be addressed by simply considering 2 exchange phases. In the first step, all the 100 columns between N2 to N3 are exchanged, while the remaining 10 columns are exchanged in a second phase. Note that in the second phase N2 hosts 30 columns, having received them from N1 in the first phase, and so is now able to send the 10 columns to N3. The aforementioned sequence of flows are computed by the master in the `ComputeFlows()` function (line 10) and thus sent to all the nodes by the `SendFlows()` function (line 11).

It is worth to note that Algorithm 2 represents a general framework for achieving a dynamic load balancing during a simulation. Most of the functions seen in the above pseudo-code do not require further specifications, except for the two methods: `LoadBalancingCriterion()` and `LoadBalance()`. The implementation of these two functions determines *when* the LB should take place and *how* to resize regions so as to achieve the "optimal" load balancing. In our preliminary implementation, the load balancing occurs at a predefined rate of CA steps while the `LoadBalance()` function follows an heuristic based on resizing the regions taking into account the region time differences normalized with respect to their old sizes. However, other strategies can be considered and *linked* to Algorithm 2 by implementing specific versions of the two methods just described.

## 4 Experimental Results

Preliminary experiments were carried out for testing the performance of the proposed LB algorithm on the SciddicaT CA debris flow model [5]. The testbed is composed by a grid of 296 columns × 420 rows, representing the DEM (Digital Elevation Model) of the Tessina landslide, occurred in Northern Italy in 1992. In order to create an initial unbalanced condition among processing nodes, landslide sources were located in the lower rows of the morphology, corresponding to higher topographic elevations (see Fig. 5). As the simulation develops, the landslide expands to lower topographic altitudes, thus progressively interesting other processing elements. The simulation was run for 4000 computational steps, corresponding to the full termination of the landslide event. Other parallelizations (e.g., multi-node and multi-GPGPU implementations) performed on SciddicaT on the same data set can be found in [9] and [33].

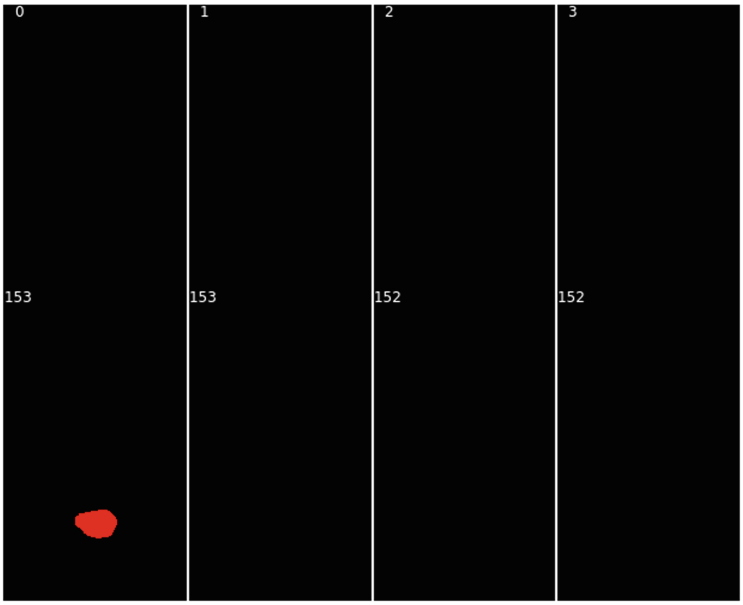

**Fig. 5.** Initial node partitioning for the Tessina landslide simulation. The initial landslide source in indicated in red. The upper numbering indicates the core id, the middle indicates the node partitioning as number of columns. (Color figure online)

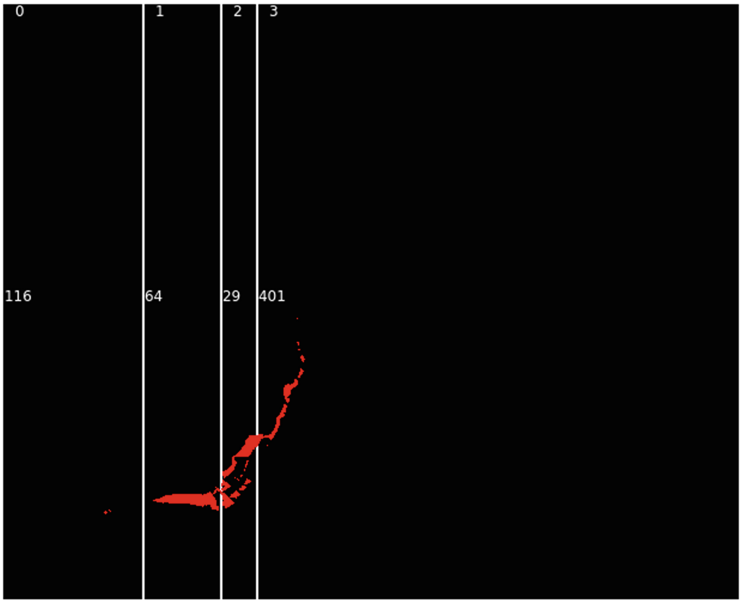

**Fig. 6.** Final load balanced configuration referred to the last step of the simulation, with the landslide that has evolved towards the upper-right side of the CA space. Please note the new node partitioning corresponding to a balanced configuration.

### 4.1 Performances

The preliminary tests were performed on a 4-core i7 Linux-based PC with 8 GB RAM. The opensource C++ OpenMPI 2.0 version of MPI was used for message passing among processes. Besides a normal not-balanced execution, three different load balancing tests were considered by considering different LB steps applications, i.e., the `LoadBalancingCriterion()` is `TRUE` each 250, 400 and 800 steps, respectively, for each of the three LB experiments. Table 1 summarizes the obtained results.

**Table 1.** Execution times of preliminary tests executed for assessing the performance of the Load Balancing algorithm on the SCIDDICA CA debris-flow model. Four different tests are reported, referred to Normal (i.e., not load balanced) execution, LB executed each 250 steps, LB each 400 steps and LB each 800 steps, respectively.

| Test | Times (s) | Improvement (%) |
|---|---|---|
| No LB | 675 | - |
| LB each 250 steps | 503 | 25% |
| LB each 400 steps | 501 | 26% |
| LB each 800 steps | 484 | 28% |

As seen, the application of the LB algorithm permitted an improvement up to 28% of the overall execution time (about 484 s versus 675 s of the not-balanced algorithm). Furthermore, for these preliminary tests, improvements are noted gradually as the number of the LB application step increases, proving that in these experiments the LB algorithm indeed provides benefits in reducing execution times, though introducing, as expected, some degree of overhead which is however counterbalanced by the aforementioned absolute time decrease. Eventually, Fig. 6 shows the last step of the simulation referred to the 800-step based LB experiments. As noted, starting from an initial uniform node partitioning, the simulation ends with a new node partitioning corresponding to a balanced execution time node configuration.

## 5 Conclusions

We here present a dynamic load balancing feature that exploits computational resources to reduce overall execution times in parallel executions of CA models on distributed memory architectures. Specifically, the algorithm executes load balancing among processors to reduce processor timings at regular intervals, based to an algorithm which computes the optimal distribution load exchange among adjacent nodes. Preliminary experiments, considering the SciddicaT CA landslide model and executed for assessing the advantage of the dynamically load balanced version with respect the non-balanced one, resulted in good improvements on a standard 4-core i7 based PC. In particular, improvements up to 28%

were obtained when the LB algorithm is applied for the simulation of the 1992 Tessina landslide (Italy).

Further experiments will be carried out in order to compute most favorable LB parameters (e.g., other LB steps), besides testing the algorithm with other LB strategies as, for instance, considering a LB criterion only when elapsed times between nodes are significant, and other heuristics to compute the new node workload. For instance, automated optimization techniques such as evolutionary algorithms or other heuristics, will be considered for calibrating LB parameters referred to the particular parallel system and the adopted simulation model.

Future developments will regard the application of the LB algorithm on other CA models, besides the extension on two-dimensional node partitioning, thus permitting the application of the LB technique also on more complex network topologies (i.e., meshes, hypercubes, etc.).

**Acknowledgments.** Authors thank BS student Rodolfo Calabrò from University of Calabria for helping in code implementation and testing phases.

## References

1. Abraham, M.J., et al.: GROMACS: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX **1**, 19–25 (2015)
2. Aidun, C., Clausen, J.: Lattice-Boltzmann method for complex flows. Annu. Rev. Fluid Mech. **42**, 439–472 (2010)
3. Amritkar, A., Deb, S., Tafti, D.: Efficient parallel CFD-DEM simulations using OpenMP. J. Comput. Phys. **256**, 501–519 (2014)
4. Andersson, B., Andersson, R., Håkansson, L., Mortensen, M., Sudiyo, R., van Wachem, B.: Computational Fluid Dynamics for Engineers. Cambridge University Press (2011). https://doi.org/10.1017/CBO9781139093590
5. Avolio, M., et al.: Simulation of the 1992 Tessina landslide by a cellular automata model and future hazard scenarios. Int. J. Appl. Earth Obs. Geoinf. **2**(1), 41–50 (2000)
6. Cannataro, M., Di Gregorio, S., Rongo, R., Spataro, W., Spezzano, G., Talia, D.: A parallel cellular automata environment on multicomputers for computational science. Parallel Comput. **21**(5), 803–823 (1995)
7. Cervarolo, G., Mendicino, G., Senatore, A.: A coupled ecohydrological-three-dimensional unsaturated flow model describing energy, $H_2O$ and $CO_2$ fluxes. Ecohydrology **3**(2), 205–225 (2010)
8. Cicirelli, F., Forestiero, A., Giordano, A., Mastroianni, C.: Parallelization of space-aware applications: modeling and performance analysis. J. Netw. Comput. Appl. **122**, 115–127 (2018)
9. D'Ambrosio, D., et al.: The open computing abstraction layer for parallel complex systems modeling on many-core systems. J. Parallel Distrib. Comput. **121**, 53–70 (2018). https://doi.org/10.1016/j.jpdc.2018.07.005
10. Di Gregorio, S., Filippone, G., Spataro, W., Trunfio, G.: Accelerating wildfire susceptibility mapping through GPGPU. J. Parallel Distrib. Comput. **73**(8), 1183–1194 (2013)

11. Di Gregorio, S., Rongo, R., Spataro, W., Spezzano, G., Talia, D.: High performance scientific computing by a parallel cellular environment. Future Gener. Comput. Syst. **12**(5), 357–369 (1997)
12. Duchateau, J., Rousselle, F., Maquignon, N., Roussel, G., Renaud, C.: An out-of-core method for physical simulations on a multi-GPU architecture using lattice Boltzmann method. In: 2016 International IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress, pp. 581–588. IEEE (2016)
13. Filippone, G., D'ambrosio, D., Marocco, D., Spataro, W.: Morphological coevolution for fluid dynamical-related risk mitigation. ACM Trans. Model. Comput. Simul. (ToMACS) **26**(3), 18 (2016)
14. Folino, G., Mendicino, G., Senatore, A., Spezzano, G., Straface, S.: A model based on cellular automata for the parallel simulation of 3D unsaturated flow. Parallel Comput. **32**(5), 357–376 (2006)
15. Frish, U., Hasslacher, B., Pomeau, Y.: Lattice gas automata for the Navier-Stokes equation. Phys. Rev. Lett. **56**(14), 1505–1508 (1986)
16. Cervarolo, G., Mendicino, G., Senatore, A.: Coupled vegetation and soil moisture dynamics modeling in heterogeneous and sloping terrains. Vadose Zone J. **10**, 206–225 (2011)
17. Gasior, J., Seredynski, F.: A cellular automata-like scheduler and load balancer. In: El Yacoubi, S., Wąs, J., Bandini, S. (eds.) ACRI 2016. LNCS, vol. 9863, pp. 238–247. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44365-2_24
18. Gerakakis, I., Gavriilidis, P., Dourvas, N.I., Georgoudas, I.G., Trunfio, G.A., Sirakoulis, G.C.: Accelerating fuzzy cellular automata for modeling crowd dynamics. J. Comput. Sci. **32**, 125–140 (2019)
19. Giordano, A., De Rango, A., D'Ambrosio, D., Rongo, R., Spataro, W.: Strategies for parallel execution of cellular automata in distributed memory architectures, pp. 406–413, February 2019. https://doi.org/10.1109/EMPDP.2019.8671639
20. Giordano, A., et al.: Parallel execution of cellular automata through space partitioning: the landslide simulation Sciddicas3-Hex case study, pp. 505–510, February 2017
21. Higuera, F., Jimenez, J.: Boltzmann approach to lattice gas simulations. Europhys. Lett. **9**(7), 663–668 (1989)
22. Jammy, S., Mudalige, G., Reguly, I., Sandham, N., Giles, M.: Block-structured compressible Navier-Stokes solution using the ops high-level abstraction. Int. J. Comput. Fluid Dyn. **30**(6), 450–454 (2016)
23. Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
24. Langton, C.: Computation at the edge of chaos: phase transition and emergent computation. Physica D **42**, 12–37 (1990)
25. Lucà, F., D'Ambrosio, D., Robustelli, G., Rongo, R., Spataro, W.: Integrating geomorphology, statistic and numerical simulations for landslide invasion hazard scenarios mapping: an example in the Sorrento Peninsula (Italy). Comput. Geosci. **67**(1811), 163–172 (2014)
26. McNamara, G., Zanetti, G.: Use of the Boltzmann equation to simulate lattice-gas automata. Phys. Rev. Lett. **61**, 2332–2335 (1988)
27. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign (1966)
28. Nicol, D.M., Saltz, J.H.: An analysis of scatter decomposition. IEEE Trans. Comput. **39**(11), 1337–1345 (1990)

29. Ninagawa, S.: Dynamics of universal computation and 1/f noise in elementary cellular automata. Chaos, Solitons Fractals **70**(1), 42–48 (2015)
30. Ntinas, V., Moutafis, B., Trunfio, G., Sirakoulis, G.: Parallel fuzzy cellular automata for data-driven simulation of wildfire spreading. J. Comput. Sci. **21**, 469–485 (2016)
31. Oliverio, M., Spataro, W., D'Ambrosio, D., Rongo, R., Spingola, G., Trunfio, G.: OpenMP parallelization of the SCIARA Cellular Automata lava flow model: performance analysis on shared-memory computers. Procedia Comput. Sci. **4**, 271–280 (2011)
32. Procacci, P.: Hybrid MPI/OpenMP implementation of the ORAC molecular dynamics program for generalized ensemble and fast switching alchemical simulations (2016)
33. Rango, A.D., Spataro, D., Spataro, W., D'Ambrosio, D.: A first multi-GPU/multi-node implementation of the open computing abstraction layer. J. Comput. Sci. **32**, 115–124 (2019). https://doi.org/10.1016/j.jocs.2018.09.012, http://www.sciencedirect.com/science/article/pii/S1877750318303922
34. Reguly, I., et al.: Acceleration of a full-scale industrial CFD application with OP2. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1265–1278 (2016). https://doi.org/10.1109/TPDS.2015.2453972
35. Schoneveld, A., de Ronde, J.F.: P-CAM: a framework for parallel complex systems simulations. Future Gener. Comput. Syst. **16**(2–3), 217–234 (1999)
36. Spataro, D., D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W., Marocco, D.: The new SCIARA-fv3 numerical model and acceleration by GPGPU strategies. Int. J. High Perform. Comput. Appl. **31**(2), 163–176 (2017). https://doi.org/10.1177/1094342015584520
37. Was, J., Mróz, H., Topa, P.: Gpgpu computing for microscopic simulations of crowd dynamics. Comput. Inform. **34**(6), 1418–1434 (2016)
38. Wolfram, S.: A New Kind of Science. Wolfram Media Inc., Champaign (2002)
39. Zito, G., D'Ambrosio, D., Spataro, W., Spingola, G., Rongo, R., Avolio, M.V.: A dynamically load balanced cellular automata library for scientific computing. In: CSC, pp. 322–328 (2009)