# University of Calabria


**Department of Mathematics and Computer Science**
**Bachelor Degree in Computer Science**


# Application of Distributed Discrete-Event Simulation Techniques in Parallel Execution of Cellular Automata


***ADVISOR***
*Prof. William Spataro*
*Prof. Donato D'Ambrosio*
***CO-ADVISOR***
*Eng. Andrea Giordano*

**EXAMINEE**
*Alessio Portaro*
*176231*



**Academic Year 2018/2019**

# Contents

# Chapter 1

# Introduction

Nowadays, many entities operating in several fields, require some software or hardware utility to improve the results of their work. Among these utilities, some of the most interesting are the so-called *Simulation Systems*, namely software frameworks, more or less complex, useful to handle and process great amount of data, concerning the same simulated model. Between the various applications, the followings are the most relevant ones :

1. **Military simulators**, used to improve the effectiveness of the recruit's training. For instance there are many fly simulators to supplement the drills of pilots, or simulators to train soldier's strategy talent, which submit them battle scenarios and handle the evolution of the simulation starting from the soldier's answers.

2. **Infrastructures simulators**, developed in order to monitor the behavior of some new technology or infrastructure's component. For example one could simulate the usage of a new communication protocol, or the impact, over a road network's traffic, of building a new transport highway.

3. **Scientific simulations** are systems developed ad hoc to collect data in favour of some new theory or monitor the possible effects of a physical phenomenon, by the simulation of a model approximating its properties. I.e. in the next chapters of this treatise, it will be depicted a model which simulates the evolution of a landslide, by the simulation paradigm of a
**Cellular Automaton** : *SciddicaT*.

4. **Entertainment Industry** . In this field there are several applications and many others are to come, because of the great interest raised and all the heavy investments. One example that stands out among the others are the *Augmented reality* platforms.

The main categorization of the *Simulation Systems* is in two opposite types, differing in the way the time flowing is modeled :

- **Discrete-event simulation**, is a paradigm where the state of the simulation instantly evolves at certain points in time, while maintains its internal state between each couple of transition instants.

- **Continuous time simulation**, on the contrary is a paradigm where the simulation's state is supposed to change continuously during the time. The association between the state of the system and a point in the simulation time is kept via a mathematical function.

Both types of simulation are strongly related to the concept of time and the way the abstraction of the this last evolves during the system's evolution. In particular, in the case of the *Discrete-event simulation*, which is the one we discuss in this thesis, there can be differentiated two main execution types, in accordance with the different ways the abstraction of the time evolves :

- **Event-Driven execution** : When this paradigm of execution is adopted, the state of the system only changes when an event occurs, that's to say when an action would occur in the real system. Each event has to bear an unique time reference and usually affects the internal state of the system, generating new events itself. Whenever possible, this paradigm can be convenient because the simulator maybe doesn't perform useless operations, but could lead to a performance reduction in **as soon as possible** simulations.

- **Time-stepped execution** : This paradigm is characterized by the contemporary evolution of the system state and the time abstraction. The system makes the state evolve following a set of rules and increments the simulated time of a fixed quantity. As soon as the state change has been computed, the system starts the computation of a new step. A *Time-stepped* execution could be convenient in a simulation to be executed in the shortest possible time.

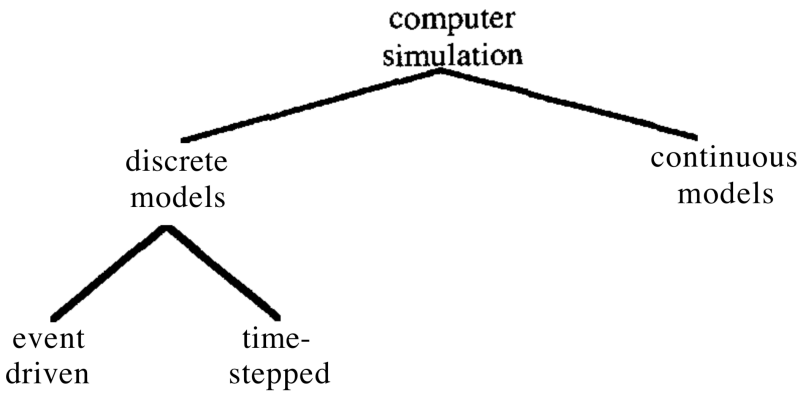


Figure 1.1: Simulation systems classification [1]

One example of *Time-stepped* execution is the **Cellular Automata** model. This simulation paradigm, often used to model natural processes, provides for a data structure representing the physical space, composed of a huge amount of *Cells* which evolve following the same fixed rules. Because in this kind of simulation, the main aim is to make the execution as fast as possible and because of how a *Cellular Automaton* evolution works, the most intuitive way to increment the performance is to split the huge amount of elementary computations concerning all the *Cells* into the space, between multiple CPUs, so that the execution is performed in parallel and in minor time. However the parallelization of the execution entails some overhead,

coming from the resolution of the problems resulting from the distribution process, one of whom is the information exchange between the parallel CPUs, needed to make evolve the *Cells* on the *Borders* of their space portions. The just mentioned communication is often performed using the **MPI** technology, which is an *API*, standard de facto for parallel applications which are designed to follow a message passing model. In the literature of the *Parallel and distributed discrete-event* simulation systems, there are many general valid solutions that could be applied to the specific case of *CA*. One of the most interesting concepts in this literature is the **LookAhead** applied to the **Synchronization Problem**. This problem occurs when two parallel processes, running portions of the same simulation, can invalidate their executions each other, affecting their respective simulated past, that's to say creating events with a timestamp minor then the actual value of simulated time. The *LookAhead* is a concept whose understanding is necessary to solve the problem, because it is a value which ensures to each parallel process, a lower-bound, in terms of simulated time, to the timestamp of the next event that could affect its simulation.
As suggested from the title, this thesis' main aim is to examine in depth some aspects of *Cellular Automata* that may result in overhead reduction and performances improvements, when this kind of simulation is executed in a Parallel and Distributed Environment. Most specifically our focus will be on the notion of **Idle Cell**, well known in CA's literature and the just mentioned *LookAhead*. Using these two concepts, it has been tried to come out with an algorithm to reduce the parallelization overhead.

# Chapter 2

# Cellular Automata

## 2.1 *CA*'s definition

Introduced by mathematician John von Neumann in 1940s, a *Cellular Automata(CA)* is a discrete computational paradigm, studied in computer science, mainly used to simulate complex phenomenon whom behavior derives from the basic interactions of spacial units named *Cells*. A *CA*, as said before, is composed of a data structure which represents the space where the simulated phenomenon takes place. This data structure, often a matrix, is composed of many elementary units named A *Cells*, therefore this structure is called cellular space. The evolution of the entire simulation is strictly related to the evolution of all its elementary spacial units. A *Cell*'s evolution, for its part, is related to the application of certain rules at every step of the simulation. Even if the rules of a *CA* can be applied to an infinitely extended space, in a real simulation most of the time the number of *Cells* is finite. Every *Cell* interacts with a relatively narrow number of other *Cells* whom state affect its evolution. The set of *Cells* which affect the evolution of every spacial unit depends on the *Neighborhood* relation. Latter is bound both to the simulated model and the existing types known in literature, from which a modeler can choose.

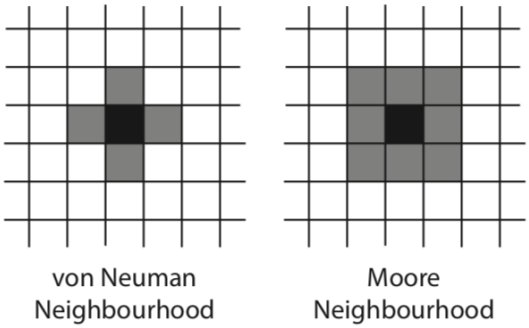


Figure 2.1: From the left to the right, Von Neumann and Moore Neighborhood with radius equal to one [2]

The *Neighborhood* relation is a geometrical pattern, which relates a set of *Cells*, fixed for every *Cell* and every step of the simulation's evolution. The two most

used types in literature, because of their simplicity and however effectiveness, are the following:

- *Von Neumann Neighborhood* : Fixed a number said radius, each *Cell* in any direction whom distance from the one considered, for instance called C1, is minor or equal to the radius, is said to be in the *neighborhood*, according to this definition.

- *Moore Neighborhood* : Fixed the just mentioned radius, for instance R, is said to be in the *neighborhood* every *Cell* whom all coordinates are distant from C1 at most R.

As said before, the evolution of every *Cell* at a certain point of the simulation is uniquely given from the internal state and the state of the *Neighborhood*. For each particular model there are a certain number of simple rules which have the aforementioned states as input and the next state of the *Cell* as output. From this point of view, each *Cell* evolves as a *FSA (Finite State Automaton)*. Despite this simulation paradigm may seem trivial, it is still very relevant for getting to see the chaotic and complex evolution of the entire model at a macroscopic level.
Let's think about *Conway's Game of Life*, a famous cellular automaton very studied in the 1970s. This is a two state and bi-dimensional automaton which elementary rules are the following :

- A *Cell* in the state **1** with a number of neighbors in the state **1** fewer than two goes to the state **0**

- A *Cell* in the state **1** with a number of neighbors in the state **1** greater than three goes to the state **0**

- A *Cell* in the state **1** with a number of neighbors in the state **1** between two and three remains in the same state

- A *Cell* in the state **0** with a number of neighbors in the state **1** equal to three goes to the state **1**

One of the most interesting aspects of this *CA* is that from some randomized initial configuration of the *Cells* in the simulated region, through its evolution this automaton shows the constant presence of some self-replicating structures, known as *Gliders*, generating from some other structure type known as *Gliders-Gun*. It can be proved that manipulating the initial configuration of the *CA* to generate *Gliders-Gun* and therefore *Gliders*, in certain *Cells* of the space, there will be certain interactions similar to computations. This aspect allows us to say that the aforementioned *CA* can emulate a *Turing machine*.
This example gives us an hint about *CA* simulation paradigm's potentiality as well as it's capability of modeling complex systems behavior, starting from very elementary rules.

## 2.2 Parallel execution of *CA*

Since a *CA*'s evolution consists of applying the same elementary rules (*Transition Function*) to an huge number of *Cells* in the simulation space, distributing the

computation on many process in order to speed up the overall time of the simulation, may seem reasonable and easily workable. The most intuitive way to distribute *CA*s is to split the cellular space in regular and contiguous portions and delegate the application of the *Transition function* on each portion, to a process into the parallel environment. The subject's literature shows several ways of making such a split, depending on the topology of the *CA*, its *cells*'s shape and the dimensions of the cellular space. In fact the split can be made along one or more dimensions. The latter case can make the split more complex.

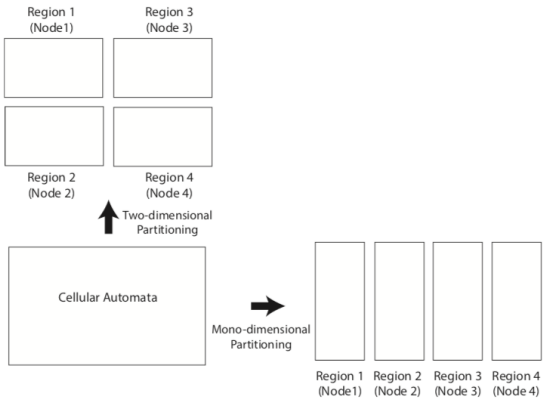


Figure 2.2: Some split examples : Mono-dimensional and Two-dimensional for a *CA* with two dimensions and squared *Cells* [2]

Let's now think about the evolution of the *CA* after being split and distributed to various processes, and let's focus on what happens for a *Cell* close to the cut of the split that as been made. The informations about the *Neighborhood* of such *Cells* are located partly in the process which makes it evolve, but the remaining part is missing. In fact some of the *Cells*'s informations are located into that processes which have been delegate of apply them the *Transition Function*. This trait is effective for every *Cell* along the aforementioned cut. It is therefore reasonable to address the entire set of *Cells* with this property and whom *Neighborhood*s reside into the same process with a proper name : *Border*. The process containing the *Border* and the one containing the *Neighborhood* of it's *Cells* are said to be adjacent processes.

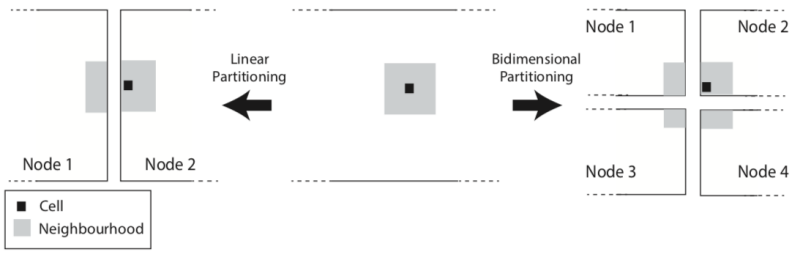


Figure 2.3: Missing *Neighborhood* of a *Cell* along the cut [2]

Because of the lack of information, before every discrete step of the simulation an interchange of information is required between every couple of adjacent processes,

each sending his *Border* and receiving another one from the respective coupled process.

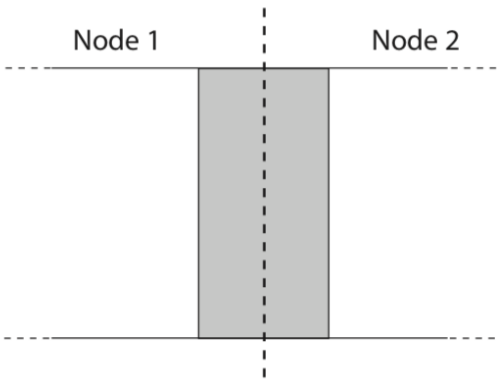


Figure 2.4: Example of a couple of processes and their related borders [2]

This preparatory operation, before the application of the *Transition Function*, introduces a performance's overhead strictly bonded to the parallel distribution of the computation. If such overhead could be avoid or just reduced, the performance gain resulting from the parallelization would be higher. The main aim of the study that is going to be shown in this thesis, is to find out some algorithm to exactly temper the mentioned overhead and obtain the related performance gain.

## 2.3 Technologies and Frameworks

As we have seen the interchange of *Borders* between processes, is suitable for a communication paradigm that uses messages. To implement this communication, has been chosen to use the well known *MPI(Message Passing Interface)* technology. It was designed as a language independent API (Application Programming Interface), in order to increase the portability of the applications using this technology, but however it has various implementations in many programming languages, among which the *C++* version was used in the implementation of the conceived mechanism. *MPI* is a standard for the communication of nodes in a cluster and is often used for high performance computing. The nodes in the cluster are often addressed as "processes" or "logical processes" and are grouped in so-called *Communicators* within which they are arranged along various topologies. The processes in this topology mainly talk in point-to-point (*p2p*) mode, and both blocking and non-blocking two-side communications are used. The main two couples of blocking and non-blocking methods are respectively **MPI_Send - MPI_Recv** and **MPI_Isend - MPI_Irecv**. When a non-blocking approach is chosen, at a stage of reception of the messages, a process has to be able to test or be notified when the operations are actually completed. One of the possible ways to perform such test of reception, is to use the method **MPI_Iprobe**, which ,without blocking the execution flow, samples the message receiving. In order to perform a *p2p* communication, each of the interlocutors has to know the ID of the other one within the *communicator*. Moreover the data type of the informations to be transmitted have to be agreed. In addition to all the native types, corresponding to the native types of the majority

of the programming languages, the standard MPI-1.2 provides for the definition of new data types (**MPI_Datatype**).
Thanks to this and other technologies, over time have been developed many frameworks for the parallel execution of *CA*. Some examples of frameworks, developed at the *"University of Calabria"*, are the following :

- *OpenCAL* : Modeling system for *CA*
- *libAuToti*: Library for *CA* implementation using MPI
- *CAMELot*: Parallel development environment for *CA*
- *VinoAC*: Framework for transparent execution of *CA* in parallel and distributed environment, using *MPI*

Among these frameworks, *VinoAC* is the one that has been expanded with a module that, as said before, tries to temper the overhead caused by the interchange of *Borders* and obtain a gain in performance terms.

## 2.4 *Cellular Automata* applications

Like it has been pointed out before, even in the *CA* case there are many useful applications, some of whom are particularly fitting for this model, despite another simulation paradigm could also be used. The introduction of this report illustrates several examples of fields where the usage of a simulation system could be advantageous. However, again, not each of that examples are befitting for the *CA* paradigm, so here are the most appropriate ones:

- *Cryptography* : Since, to days, a great percentage of the activities in the world of work makes use of the internet to exchange sensitive and personal informations, the *cryptography* field is gaining more and more relevance in order to come out with mechanisms that preserve the confidentiality of these informations. In this context, it could turn to be useful the usage of *CA* to create a *cryptography* key. It's in fact very important in this field to create a strong key, thanks to which the informations can be encrypted, whom prediction is statistically unworkable.
  As said before the evolution of an entire *CA* is often really complex as a whole, in spite of the elementary rules from which descends. Therefore, one could apply to a random initial configuration of a *CA* a *Transition Function* for a random number of times, in order to obtain an unpredictable state to fully or in part use as a key. The reliability of this method is discussed thus far, but it can be proved that the time complexity required to perform an attack to such a key is exponential.

- *Artificial Life* : Is known for some time now, in the field of Biology, that many composite organisms, which seem to have a sentient behavior at a certain degree, is actually driven by the basic interactions of elementary insentient components. This components are often static in the space and therefore can only interact with the other components which are near to them. In view of this, many biologists come out with the idea of applying the *CA* simulation

paradigm to this complex systems, since it is clear how the model is very fitting to this context, in order to simulate the just mentioned interactions and study more efficiently these phenomena. A well known example is the aforementioned *Conway's Game of Life*, which is a simple but very didactic model to understand the potential of such simulations for fields like the *Populations Ecology and Biology*.

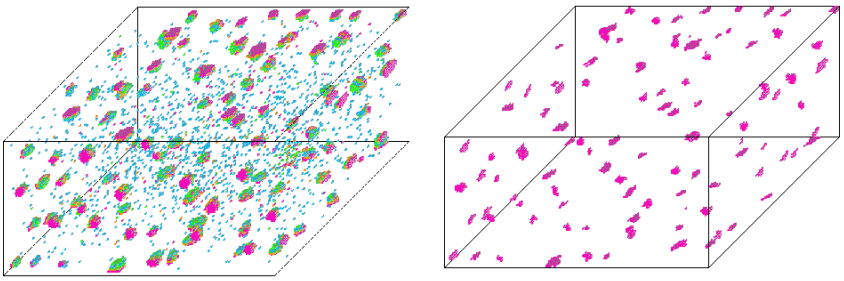

Figure 2.5: Bacterial Hyperstructures simulated with a *Cellular Automaton* [3]

One last example which could sound more relevant and even of tendency for the field of computer science nowadays, is the case of the *Cellular Neural Networks (CNN)*. Even in this case in fact, as known, a *Neural network* is a computational model inspired by the functioning of the human brain, that's to say how its complex behavior depends on the interactions between the neurons. The applications of a *Neural Network* are well known in literature and won't be debated here, however it's important to underline that a *CNN* model differs from a classic *Neural Network* because in the first case the information exchange is allowed only between neighboring units, like in a *CA*.

- *Scientific Modeling* : The final, but most relevant, application discussed here is for the field of the *Computational science*. The main aim in this sphere is to come out with computable models, that can solve mathematical or physical problems, or simulate a natural process and therefore be able to predict its effects and avoid the potential damages. Among the several possibilities discussed already, a *CA* is the one more appropriate for those phenomena which, as said before, are regulated by simple local rules.

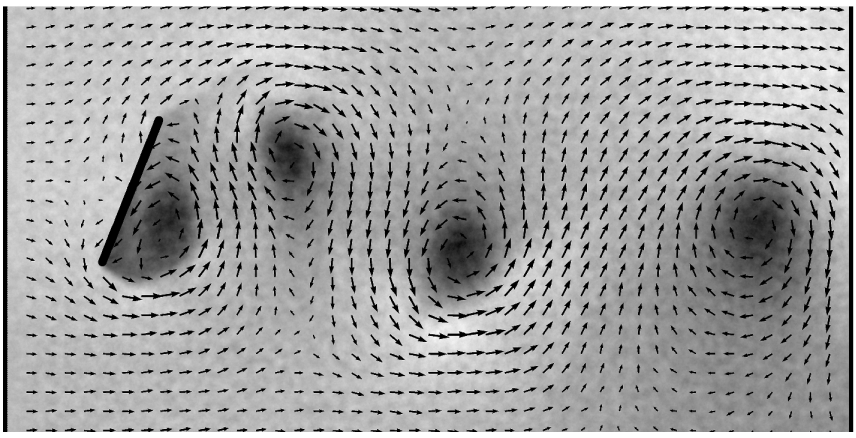

Figure 2.6: Lattice Gas *cellular Automaton (LGA)* [3]

Such paradigm is in fact often applied to model natural processes comparable to the flux of fluids. A famous example is the one of the *Lattice Gas cellular Automaton (LGA)*, well studied in the 1990s, which models, in various versions, exactly fluid flows and has led to several mathematical results, even in a continuous form. Another *CA* model, that will be discussed in the next chapters as a study case for this research work, is the so-called *SciddicaT*. This model simulates, even here, a flow of rubbles rolling down along a ridge which then may active a landslide.

This discussion has only mentioned a small number of fields of application for a *CA* and only the most relevant ones, for our purposes, have been chosen. Between these, the field of the *Computational Science* was the one inspiring this thesis and the starting point for the results that came out from the research work, even if these latter can be applied in other spheres.

# Chapter 3

# *D*istributed discrete-*E*vent *S*imulation

*DES*

## 3.1 Parallel Environments and Execution paradigms

Even if the main aim of this research work was to think up to an algorithm, ensuring the consistency of the simulation's results and a good speed-up, without hardware or software constraints, so that it can be as much generally strong as possible, sometimes certain assumption on the parallel environment's characteristics have turned necessary. It can be useful for this reason to get a quick overview of the primary traits which distinguish a parallel environment, in order to later understand the development choice that have been made.
First of all let us talk about the several possible hardware platforms. The main categorization differentiates the following machines, in the way all the informations are stored :

- Shared memory Multiprocessors : Multiple processors share the same physical memory, hence every change to the data is visible to the other processors immediately
- Distributed memory Computers : Multiple computers often heterogeneous have their own memory. If one of the computer modifies some data in its own local memory, and some other one needs exactly that data, some kind of message exchange is need. For this kind of architecture is very common the employment of general purpose connection networks rather than some customized one. This results in much more latency for the communication

More targeted architectures exist and have better performance than those aforementioned, however it doesn't make any sense to talk about an algorithm for such architectures here.

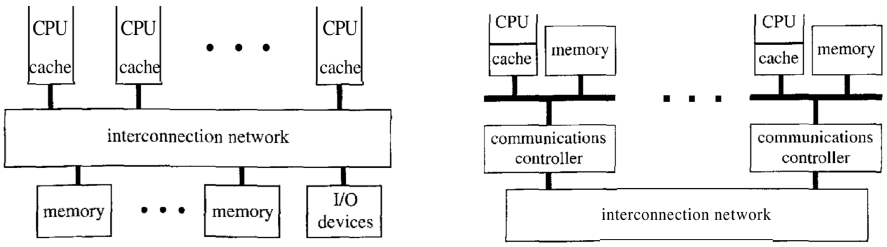


Figure 3.1: From the left to the right, shared memory multiprocessors architecture and distributed memory computers architecture [1]

In second place, we can categorize the various paradigms for the parallel execution of a program, which can be summarized in these types :

- *SPMD(Single Program Multiple Data)* : The same executable is launched multiple times and different data in input (often one of these is some process ID)
- *MPSD(Multiple Program Single Data)* : For every kind of task, a targeted program is created
- *MPMD(Multiple Program Multiple Data)* : Less used paradigm, arrangement of the first two in this list

The parallel environment whom this text will refer from now on, belongs to the category of the Distributed memory architectures and uses the SPMD paradigm execution, even more because the latter is used in many *C++* implementation of the *MPI* technology, considered in the previous chapter.
Besides the one made before, a further observation about the communication network concerns the reliability and the keeping of the sending order. In the conception of the algorithm, purpose of this work, it has been assumed a 100% reliability of the connection network between the processes in the parallel environment, as well as it has been assumed that, traversing the network, the sending order of the messages doesn't change. This trait will result clearly important in the next chapter.

## 3.2 Discrete Event Simulation

The *Discrete-Event Simulation* paradigm's attempt, is to model the evolution of some real system with a chain of consecutive states, which stay unvaried for a definite amount of time preceding an instant when the system is thought to evolve immediately and enter in a new unchanging state. In this context the temporal dimension is very important to uniquely identify the state of the simulation, but it turns out to be necessary to give a definition of all the different "kinds" of time that can be encountered in the treatise. In the literature [1] there are the following definition, which are really relevant for the considerations that are going to be made :

- *Physical Time* : Is the amount of time that would have elapsed, between two events or instants during to phenomenon being simulated, into the real world
- *Simulated Time* : Is an abstraction of the *Physical Time*, used inside the computer context to remark the state of the simulation's evolution
- *Wallclock Time* : Is the amount of time taken from a machine to perform all of the necessary computations to make the simulation evolve to a certain *Simulated Time*. Obviously the latter quantity depends on the hardware executing the aforementioned computations

Related to the disambiguation of the overlying terms, it may be useful to briefly examine how the *Simulated Time* can be handled, in terms of how fast but also what is the reason of its increment. The main partition, concerning the haste of the simulation's execution, is into :

- *Real time Simulation* : This kind of simulation needs the human interaction, maybe waiting for some input or simply a check. A military simulation, targeted for an airplane's pilot, may be an example of application where this approach is necessary, even if it slows down the execution time
- *As fast as possible Simulation* : When the simulation doesn't need any kind of external input, but the main purpose is to speed-up the execution, in those cases this approach is the most appropriate

Totally clear from the above partition, is the one concerning the reasons of every discrete state's change. In fact depending on the model being simulated, a simulation is said to be :

- *Event driven* if the events internal or external to the simulation, cause a change to the state and the *Simulated Time*
- *Time stepped*, more suitable for an execution *As fast as possible*, if the simulation doesn't need any input to evolve and therefore when a state is completely set, the next one can be computed

A *Time stepped* approach seems befitting for a *CA* model, because its evolution consists of reading the state a the step $t$ of the *Simulated Time* and then compute the state of the *CA* at the time $t+1$, without any other requirement.

## 3.3 The synchronization problem and lookahead

As previously occurred in this treatise, when a sequential algorithm is transposed into a parallel environment, then some problem to be solved comes out. That's exactly the case for a *Discrete Event Simulation*'s parallelization.
For instance in a Distributed environment containing heterogeneous machines, it is not that difficult to find out that at a given instant in the *Wallclock Time* two different processes have reached different points into the *Simulated Time*. Let's call this two processes **Lp1**(standing for *Logical process*) and **Lp2**, and let's say Lp1 has reached a point into the *Simulated Time* successive to the point reached from Lp2. If Lp1 and Lp2 can respective computations, the discrepancy between the *Simulated*

*Time* instants could be a problem, when Lp2 affects Lp1 in its past. Lp1 in fact would need to handle that event which potentially could have changed its state if it had arrived earlier in the *Wallclock time*. In this case Lp1 is said to violate the *Local Causality Constraint*, defined in literature as follows :

> *Local Causality Constraint* : A discrete-event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order [1]

To obviate to this issue one has to come up with some algorithm to prevent a situation like the one mentioned. The algorithms solving this problem can be categorized in two groups :

- *Conservative approach* : An Lp can wind on only if it has the assurance to not receiving any message or event that could affect its past states
- *Non conservative approach* : An Lp planning to not receive anything affecting its past can wind on, but has to perform a roll-back operation if receives such a message or event

When a *Conservative approach* is chosen an Lp at an instant $t$ can process an event with a timestamp $t+l1$ or make the simulation advance to the step $t+l2$, only if it is sure that nothing with a timestamp less then $t+l1$ or $t+l2$ will be received. To make such assumption an Lp must have enough information about which other Lps in the simulation can affect its execution an particularly the value of the so-called *Lookahead*. The latter is defined as follows :

> *Lookahead* : If a logical process at simulation time $\boldsymbol{T}$ can only schedule new events with time stamp of *at least* $\boldsymbol{T+L}$, then $\boldsymbol{L}$ is referred to as the lookahead for the logical process [1]

To underline the relevance of this quantity, let us thing for instance to a situation when it equals zero. Under this condition an Lp cannot process any event or wind on to the next step because then it could receive a message violating the *Local Causality Constraint*. In the next chapter the importance of this concept is going to result even more strong, when a parallelism with *CA* paradigm will be drawn.

# Chapter 4

# Application of *DES* techniques in Parallel execution of Cellular Automata

## 4.1 Synchronization problem for *CA*

As a first step of this research process, it has been tried to study the synchronization problem applied to the *CA* paradigm in a parallel and distributed environment, in order to came out with some kind of *Conservative* algorithm which could prevent the infringement of the *Local Causality Constraint*, in a situation where every Lp handles a different automaton with its own time step, in terms of *Simulated Time*. In that situation the different *CA* would have been modeling different phenomena into the same complex system, and therefore would have been related to each other.

Since also in this case an algorithm with the most general validity was wanted, the problem has been modeled around two main data structures. The first one is a dependency graph, dynamically configurable, which denotes for each Lp in the simulation, those which it can affect and those which can affect its own simulation. Let's call it **dependency_graph**. Let this structure be composed by two lists, one containing the IDs of the Lps on which one depends, and the other containing the IDs of the Lps depending from it. Let's call the lists respectively **incoming_arcs** and **outgoing_arcs**. The second data structure is a vector containing the time step information. In particular, each Lp has two vectors, one containing the information about the Lps which depend from it, and a second one containing the information about the Lps on which it depends. It is important to say that just because of its connotation of general validity, any focus was given on the kind of information that a generic couple of processes has to exchange each other. The only constraint which is supposed to bound the evolution of every Lp, is that it has to receive from every Lp on which depends, its latest state informations as well as it has to send its own state information to let wind on the Lps which depend by it. Let's call the two vectors respectively **outgoing_steps** and **incoming_steps**.
After the element appearing in the algorithm being discussed, it's time to discuss its functioning as well. It will be used an example here, in order make the presentation of the algorithm easier. For instance immagine two processes Lp1 and Lp2 designated to make evolve in parallel environment, a couple of interdependent CA. Lp1 has a

time step of 2 units meanwhile Lp2 has a time step of 9 units and depends on the first one. After a preparatory informations exchange at the *Simulated Time* 0, every process is ready to evolve at the next step, therefore Lp1's *Simulated Time* goes to 2 while instead Lp2 goes to 9. Because of its dependency from Lp1, Lp2 now can't evolve to the next step until Lp1 reaches the greatest instant until it moves into the future of Lp2. Lp1 instead can safely advance to time step 8 with 3 transitions. After performing every transition both Lp1 and Lp2 update not only their own time reference but also a reference for every of their related Lps. In our case Lp1 updates a reference to the time of Lp2 and Lp2 updates a reference for Lp1's time. It is exactly for this reason that when Lp1 hits the time step at the *Simulated Time* 8, knowing the time step of Lp2 read from the vector **outgoing_steps** can say it needs to send its state's informations to Lp2 which is dependent from him. Symmetrically, for the same reason Lp2 can say that he needs to receive some information from Lp1 to be able to evolve. Subsequently to this informations exchange both Lp1 and Lp2 are able to procede with their own operations.

**Algorithm 1** run_simulation() *Pseudo-code of the algorithm's main loop*

```
/* Vector containing the step reached from
 *   every dependent Lp
 */
Simulated_time_outgoing[N]
/* Vector containing the step reached from
 *  every Lp on which this one depends
 */
Simulated_time_incoming[N]
while not SIMULATION_END do
  Can_do_step = true
  for all Lp in incoming_processes do
    if Simulated_time_incoming + incoming_steps <MY_CURRENT_TIME
    then
      Can_do_step = false
  if Can_do_step then
    CA_Transition()
    MY_CURRENT_TIME += step
    for all Lp in outgoing_processes do
      MPI_Isend(MY_CURRENT_TIME)
  for all Lp in incoming_processes do
    incoming_message = MPI_IProbe(Lp)
    if incoming_message then
      MPI_Recv(Lp)
      Simulated_time_incoming += incoming_steps
```

Like one can see from Algorithm 4.1, in a real situation with a more complex dependency graph, thanks to the *MPI*'s non-blocking functions **MPI_Isend** and **MPI_IProbe** any deadlock situation can be prevent. Furthermore, again thanks to *MPI*, the algorithm doesn't need to implement any acknowledgment mechanism to make sure that every dependent Lp knows about the advancement of the simulation in another process because, as said before, the *MPI* communication network can

thought with a perfect reliability.
Even if this algorithm can be useful in certain situations, the effort to think up such a mechanism was minimal as well as it's minimal its scientific significance. In fact it doesn't profit by any of the aforementioned *DES* techniques nor brings to any performance gain in the execution of such kind of simulation. The reason is that every *CA* simulation can be categorized as a time-stepped simulation and moreover in usually it's evolution depends on its own internal state. A general mechanism as the one just described is meaningless if any assumption, about how the execution of a *Cellular Automaton* can be affected, can be done.

## 4.2 *Lookahead*'s study for border swap avoidance

Things are different if we talk about the execution of a single *Cellular Automaton* distributed through a parallel and distributed architecture. In this case in fact, every Lp which is assigned to make evolve a portion of the cellular space is inherently bound to the other Lps in the simulation. In this context, as seen in the section dedicated to the Parallel execution of *CA*, every couple of adjacent Lps, has to permanently communicate, asking and being asked about their respective *Borders*. Because of this, in the second phase of the study work, the main focus has been on finding a way to moderate the overhead due to that communication.
In particular an Lp containing a *Border* that has not changed, in any of its cells, after applying the *Transition Function*, in principle doesn't need to transmit that *Border* again. Unfortunately there is an issue here regarding the possibility of run into a *deadlock* situation. So, if for example an Lp skips a transmission, its adjacent interlocutor, which expects this information after every step of the evolution, will freeze its execution. Thus, a mechanism to let every the Lps know when to receive or send a *Border* is needed. In other words each Lp needs a *Lookahead* information about every of its adjacent interlocutors. However, to find a value for such information one needs to find some middle ground between a general valid concept and something dependent on the model. The middle ground was found into the *CA*'s concept of *Idle Cell*. A cell whom state doesn't change after applying the *Transition Function* is said to be and *Idle Cell*. This concept is particularly relevant if one thinks of the possible assumptions that can be done about the evolution of a *Cell* with another *Idle* one in its *Neighborhood*. For instance it's not unreasonable to think that under certain conditions such *Cell* could take two steps to change its state. Applying the same reasoning recursively to *Cells* more and more distant from an active one, likewise the number of steps to change their state could grow up. In this way an Lp can find a value of *lookahead* for its borders if considering the *Idle Cells* in its own Cellular space and in its adjacent Lp's ones. It seems clear, from the previous arguments, that a *lookahead* calculated with this assumptions, has to be directly proportional to the distance from the nearest active *Cell* of each border.

## 4.3 The starting point for the implementation

Since, unlike in the first algorithm's event, in this case there are many frameworks already implemented which are delegated to run a *CA* in a parallel and distributed environment, it was decided to use one of these as a starting point. Between several

of these, it was decided to use the "University of Calabria" 's framework named *VinoAC*. In particular, this framework has been extended with a module whom functioning will be shortly explained. As said in the past chapters, *VinoAC* is a framework for the transparent execution of a *CA*. The modeler has only to specify its model's elementary rules and then to start the simulation, without caring about the internal mechanism which permit the execution. This framework was developed in *C++*, thus it has an Object-oriented structure. Its architecture was designed to permit the usage of different parallel communication paradigms, but the only implementation to date uses the aforementioned *MPI* paradigm. The three primary classes in the framework architecture
are the following :

- ***Model2D*** : This is an object containing all of the model's information. The framework provides a basic interface to be extended for every specific CA. After one overrides the class's method which represents the transition function the simulator can make the CA evolve.

```
1  #ifndef MODEL2D_H
2  #define MODEL2D_H
3
4  #include <Space2D.h>
5
6  template <class T>
7  class Model2D
8  {
9    protected:
10     Space2D<T> *space;
11   public:
12     void setSpace(Space2D<T> *space);
13     virtual void init() = 0;
14     // Transition function of the (x,y) cell
15     virtual void transitionFunction(int x, int y) = 0;
16     virtual void finalize() = 0;
17     virtual ~Model2D();
18 };
19 #endif
```

Listing 4.1: Model2D.h, Interface to be implemented by the Modeler

- ***Space2D*** : This is an object whose job is to handle the cellular space. The modeler doesn't need to know if the cellular space is distributed in multiple processes or is computed in sequential. As we said before, the basic interface is designed to be implemented using different communication paradigms, but the one used from the conceived algorithm uses the *MPI* paradigm. This object contains two matrix representing the cellular space. From the matrix **regionCurr** the simulator reads the current state of the *CA* and applying the *Transition Function* it writes the new state into the matrix **regionNext** and then calling the method **swap()**, the latter matrix is copied into the first one.

```
1  //Class for a two-dimensional CA space when the MPI technology is adopted
2  #ifndef SPACE2DMPI_H
3  #define SPACE2DMPI_H
4
5  #include <Space2D.h>
6  #include <Element.h>
7
8  template <class T>
9  class Space2DMpi : public Space2D<T>
10 {
11 protected:
12   enum Direction { N, NE, E, SE, S, SW, W, NW};
13
14   int currentStep = 0;
15   // Read matrix
16   T *regionCurr;
17   // Write matrix
18   T *regionNext;
19
20   void updateBorders();
21 public:
22
23   void swap();
24 };
25
26 #endif
```

Listing 4.2: Space2DMpi.h, Space handler using the MPI paradigm

- ***Engine2D*** : This is an object whose job is to pull together the two previous objects. It reads the informations from the model and then it changes the state of the *CA*. After initializing the simulation, the method **start()** is called. This method contains the main loop of the simulation.

```
1  #ifndef ENGINE_H
2  #define ENGINE_H
3
4  #include <Model2D.h>
5  #include <Space2D.h>
6
7
8  //Base class for the engine running a 2D cellular automata
9
10 template <class T>
11 class Engine2D{
12     protected:
13     // The CA space
14     Space2D<T>* space;
15     // The application model
16     Model2D<T>* model;
17     // The number of steps
18     int nsteps;
19
20 public:
21
22     void setSpace(Space2D<T> *space);
23     void setModel(Model2D<T> *model);
```

```
24      //Sets the number of steps
25      void setNsteps(int nsteps);
26      void start(){
27          // Initialize the cellular space
28          model->init();
29
30          for (int step = 0; step < nsteps; ++step){
31              space->setCurrentStep(step);
32              // Initialize the step
33              space->startStep(step);
34              for (int x = minX; x <= maxX; ++x)
35                  for (int y = minY; y <= maxY; ++y)
36                      // Apply the transition function to every cell in the space
37                      model->transitionFunction(x,y);
38              // Write regionNext into regionCurr matrix
39              space->swap();
40          }
41      }
42  };
43
44  #endif
```

Listing 4.3: Engine2D.h, The engine running the simulation and the main loop method

This framework has been chosen as a starting point of the implementation phase of this work, due to its suitable architecture presenting a clear division between the *CA*'s model and its cellular space. But, even if *VinoAC* was a good starting point, some criticality were found into the **Space2D** class and in the way it handles the *Borders* exchange.

## 4.4 *VinoAC*'s *LookAhead* module

Since *VinoAC* implements the classic execution of *CA*, the class **Space2D**, after each step of the simulation, takes care of the *Borders* exchange between every couple of adjacent Lps. That's exactly why the first thing clearly to modify was this class and more specifically its method **startStep(int step)** which contains all the preliminary operations to each step of the simulation's evolution, including the *Borders* communication. Secondly a way to know if a *Cell* is *Idle* seems necessary, therefore the base class *Model2D* has been extended with the addition of a the method **bool isStanding(int x, int y)** by which it is possible to ask for such information to the specific *CA* model. With these elements being given, the only remaining thing to do is to explain the main algorithm's functioning.
The algorithm consists of two parts, one which dynamically updates the lookahead value for each adjacent Lp of the one considered, during the application of the *Transition Function*, and the other which handles the conditional exchange of the *Borders* or some other related information. The second part is the one that's going to be explained first.
Consider as usual, two adjacent processes Lp1 and Lp2. Let's consider only the Lp1's point of view as a simplification in order to better understand the functioning of the mechanism. With this assumption we mean to say that Lp1 is the sender of the *Border* whereas Lp2 is intended to only play as the receiver. Suppose also

that in an initial phase both Lp1 and Lp2 know about the *LookAhead* value related to the communication that they usually do at every step. From now on we'll refer to this value with the symbol *LAh*. Considering what the *LAh* value means, that's to say a promise that the *Border*, known from both the Lps in an initial phase, won't change for the next *LAh* steps, then an Lp can make evolve its portion of cellular space for that number of steps, saving itself the overhead caused by the usual communication operation. Obviously every both Lp1 and Lp2 are meant to decrement their local reference of the *LAh* value after each step of the evolution, in order to contemporary reach a situation where the reference equals 0, meaning that the *Border* well know until that moment could have changed and therefore the Lps need to perform some kind of communication to figure out what to do next. After reaching this point, Lp1 computes a new value of *LAh*, which as known is a non-negative integer. Accordingly to the resulting value, a choice takes place :

1. If the *LAh*'s value is strictly greater than zero, that means the *Border* won't in effect change. It's important in fact to underline that the *LookAhead* value in a context of general validity, is intended to represent only a lower bound for the real number of steps that a *Border* needs to change. The modeler in effect could be unable to make such strong assumptions on the simulated phenomena to come out with an exact value of *LAh*. Since it has this information, Lp1 notifies Lp2 that the *Border* won't change sending a message with tag **LookAhead_Update**, containing the new value. Lp2 after decrementing to zero its reference to *LAh*, puts itself in listening of some message. Reading the tag of the message sent from Lp1 updates its reference of *LookAhead* and resumes its execution.

2. If otherwise the *LAh*'s value equals zero, that means the *Border* will change after the *Transition Function* is applied. Therefore Lp1 sends a message with tag **LookAhead_Update_Following_Border**, meaning that it will send the new *Border* into the next step. Then, just after the *Transition Function* is applied and the *Border* has changed, Lp1 sends a message with tag **Border_Update**, containing the new boundary informations. Lp2 which was waiting for a message, receives the one with tag **LookAhead_Update_Following_Border**, applies the *Transition Function* and then receives the new *Border*.

Thanks to this first part of the algorithm, an Lp which can compute the value of *LookAhead* for each of its *Borders* can improve its performance in the execution of the *CA* moderating the overhead coming from the parallelization. It is clear that in a real situation each Lp has to communicate with several adjacent processes and it also has to play both the roles of *Borders* sender and receiver. In order to avoid any *deadlock* situation, even in this case the *MPI*'s non-blocking functions fit our case. In fact, performing all of the send operations in non-blocking mode and then, playing the role of receiver, performing all of the receive operations, every Lp can avoid any kind of *deadlock*.

```
#ifndef SPACE2DMPILAH_H_
#define SPACE2DMPILAH_H_

template <class T>
class Space2DMpiLAh: public Space2DMpi<T> {
public:
void Space2DMpiLAh<T>::startStep(int step){
    //Two FOR cycles to avoid deadlock. The first one contains only non-blocking send
    for(int i=0;i<Neighbors_Number;i++){
        if(last_LAh_sent == 0){
            //To advance this Lp's neighbor needs some info about this Lp's border
            int new_lookahead = computeBorderLookAhead(i);
            if(new_lookahead>0){
                /* When the new_lookahead value is still greater then zero,
                 * it means the border sent in the past still won't change for
                 * at least LAh number of steps
                 */
                MPI_Isend(LOOKAHEAD_UPDATE);
            }
            else{
                MPI_Isend(LOOKAHEAD_UPDATE_FOLLOWING_BORDER);
            }
        }else last_LAh_sent--;
    }
    // Receiver loop
    for(int i=0;i<Neighbors_Number;i++){
        if(last_LAh_received == 0){
            //To advance this Lp needs infos about neighbor's border -> SEND

            while(true){
                MPI_Iprobe(LOOKAHEAD_UPDATE);
                MPI_Iprobe(LOOKAHEAD_UPDATE_FOLLOWING_BORDER);
                if(RECEIVED_LOOKAHEAD){
                    MPI_recv(LOOKAHEAD_UPDATE);
                    break;
                }
                if(RECEIVED_LOOKAHEAD_N_BORDER){
                    MPI_recv(LOOKAHEAD_UPDATE_FOLLOWING_BORDER);

                    MPI_recv(border);
                    break;
                }
            }
        }else last_LAh_received--;
        }
    }
}

};

#include "Space2DMpiLAh.ipp"
#endif /* SPACE2DMPILAH_H_ */
```

Listing 4.4: Space2DMpiLAh.h::**startStep(int step)**, implementation of the first part of the algorithm

The assumption of the discussed mechanism however, remains the computation of the *LAh* value. As said before the *LookAhead* is a quantity supposed to be, in the *CA*'s case, proportional to the distance between those *Cells* which are not *Idle*, and the *Border* of the cellular space's portion assigned to an Lp.
After underlining that even the proportionality above is an implementation choice, that could be invalidated in certain cases, we can say that for simplicity and for the purpose of letting the algorithm apply in general, it has been chosen to make the *LookAhead* exactly coincide with that distance value. This chose allows to maintain the lower-bound connotation of the *LAh*, compared to its computation as a quantity proportional to the distance, meaning it would have been a greater value. Because of the data structure used to represent the cellular space, that's to say a two-dimensional matrix, the computation of a distance occurs with constant time complexity (***O(1)***), thanks to the indexing of every element in such data structure. Since the *LookAhead* is a lower-bound to the number of steps needed from a *Cell* on the *Border* to become active and therefore change, requiring another *Borders* exchange, then its value has to be computed as the distance between the nearest not *Idle Cell* and each border. The crucial point is then to detect the nearest active *Cell*. In order to do find this *Cell*, each Lp has to iterate all over its portion of cellular space. Such iteration is also performed during the application of the *Transition Function*, therefore one can take advantage of that and during this iteration check whether a *Cell* is *Idle* or not, and in the latter case compute its distance from each *Border*, maintaining this value as its *LookAhead*, if it's smaller than the minimum at that moment.
In the *VinoAC* framework, the iteration is delegated to the class **Space2D**, which has been extended in the *LookAhead* module by the class **Space2DMpiLAh**. This class referring to the method **isStanding(int x, int y)** of the class **Model2DLAh**, can know if a *Cell* is *Idle*. Starting from this information, an Lp can save into a vector a *LookAhead* value for each of its neighbors.

```
1  #ifndef SCIDDICAT_H
2  #define SCIDDICAT_H
3  class SciddicaT : public Model2DLAh<T>{
4      void transitionFunction(int x, int y){
5          /*[...]*/
6          /* Typical usage of the methods
7           * isStanding and updateBordersLookaheadConsidering */
8          if(!isStanding(x, y)){
9              Space2DMpiLAh<T> *space_lah = dynamic_cast<Space2DMpiLAh<T> *>(space);
10             space_lah->updateBordersLookaheadCosidering(x, y);
11         }
12
13     }
14     bool isStanding(int x,int y);
15
16 };
17
18 #endif /*SCIDDICAT_H*/
```

Listing 4.5: An implementation of Model2DLAh. The update of the lookahead's vector exploiting the iteration necessary to apply the *Transition Function*

As one can see from the Listing 4.5, the class **Model2DLAh** thanks to its reference to **Space2D**, calls the method **updateBordersLookaheadCosidering**, to

let the cellular space's handler update the vector containing all the *LookAheads*. This last method implementation simply computes the distance between the input *Cell* and each *Border* to later, with a conditional statement, update the just cited vector. If the cellular space would have been split with a two-dimensional cut, the resulting *Borders* would have a complex shape, which would make the computation of their distance from a *Cell* more difficult. Thus, in order to make these computations easier, it has been decided to apply to the cellular space a mono-dimensional cut. Thanks to this choice the *Borders* between each couple of adjacent space portions, result to have a regular shape and therefore the computations of the distances happen in constant time.

As a final analysis, a relevant aspect to examine is the behavior of the discussed mechanisms, in the context of a real implementation. As seen before, the real computation has required several simplifications and adjustments of the mechanism even if it seemed to fit into an ideal setting. For instance, let's consider two processes Lp1 and Lp2 which both found in their portion of cellular space, that we suppose to be adjacent, two *Cells* **C1** and **C2** respectively the nearest to the cut distinguishing the regions. Consistently with the distances of **C1** and **C2** from the *Border*, both Lp1 and Lp2 compute the value of *LAh* and in the majority of the cases the two values are different, because each Lp iterates only over its portion of cellular space, even if there could be a closer *Cell* to the *Border* but located in the adjacent portion of space. This inconsistency, could invalidate the entire simulation. The solution to this problem is merely to compute a minimum of the to values, respectively bound to the positions of **C1** and **C2**, on the occasion of the communication between Lp1 and Lp2, and consider the resulting value as the correct one.

Once this adjustments are done, the algorithm is ready to be effectively implemented without affecting the consistency of the simulation in any way. The last thing to examine thus is the performance improvement in terms of time, which, it must be remembered, is the main aim that prompted the ideation of the discussed mechanism.

# Chapter 5

# Experimental Results and Performance Evaluation

*Test case : SciddicaT*

# Chapter 6

# Possible future extensions

# Bibliography


[1] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley, 1999.

[2] A. Giordano, A. De Rango, D. D'Ambrosio, R. Rongo, and W. Spataro, "Strategies for parallel execution of cellular automata in distributed memory architectures", *27th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2019.

[3] L. Le Sceller, C. Ripoll, and V. Norris, "Modelling bacterial hyperstructures with cellular automata", *Springer, Berlin, Heidelberg*, 2006.

[4] M. Gorodnichev and Y. Medvedev, "A web-based platform for interactive parameter study of large-scale lattice gas automata", *Springer, Cham*, 2019.