A cloud computing based framework for general 2D and 3D cellular automata simulation

Rodrigo Marques[1], Bruno Feijo[1], Karin Breitman[1], Thieberson Gomes[2], Laercio Ferracioli[2], and Hélio Lopes[1]

[1]*Department of Informatics, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil*
[2]*Department of Physics, Federal University of Espírito Santo, Vitória, ES, Brazil*

Discussion Paper - 2013

ABSTRACT

Cellular automata can be applied to solve several problems in a variety of areas, such as biology, chemistry, medicine, physics, astronomy, economics, and urban planning. The automata are defined by simple rules that give rise to behavior of great complexity running on very large matrices. 2D applications may require more than 10^6 x 10^6 matrix cells, which are usually beyond the computational capacity of local clusters of computers.

This paper presents a solution for traditional cellular automata simulations. We propose a scalable software framework, based on cloud computing technology, which is capable of dealing with very large matrices. The use of the framework facilitates the instrumentation of simulation experiments by non-computer experts, as it removes the burden related to the configuration of MapReduce jobs, so that researchers need only be concerned with their simulation algorithms.

KEY WORDS: Cloud Computing; Cellular Automata; Simulation; Sparse matrices; Game of life; Web services

1. INTRODUCTION

Cellular automata can be applied to solve problems in a variety of areas, such as biology, chemistry [1], medicine [2], physics [3, 4], astronomy [5], economics [6], and urban planning [7]. These automata are defined by simple rules that give rise to behavior of great complexity running on very large matrices. 2D applications may require more than 10^6 x 10^6 matrix cells, which are usually beyond the computational capacity of local clusters of computers.

Moreover, it is a huge amount of work to create a system to subdivide simulations, and process and combine the results. Cellular automata researchers put a lot of effort into networks and parallel distributed computation issues to solve their cellular automata problems.

1.1.Contributions.

This paper shows a solution for traditional cellular automata simulation based on cloud computing technology. The objective is to create a framework capable of dealing with very large matrices, highly scalable, and simple to use by non-experts so they need only be concerned with their algorithm, leaving the subdivision and parallelization processes to the framework.

Cloud computing technology is an increasingly popular trend to create dynamically scalable solutions. Rajkumar *et al*. [8] present a set of benefits of cloud computing, but, for the purposes of the present framework, we put our focus on scalability.

1.2 Paper outline.

The rest of this paper is divided as follows. Section 2 describes the foundations of cellular automata theory upon which our results are based. Section 3 gives a brief introduction of cloud computing concepts. Section 4 describes the proposed framework. Section 5 presents a case study and simulation results. Section 6 concludes this work.

2. CELLULAR AUTOMATA

2.1. Proposed Type of Cellular Automaton

A generic framework capable of running any type of cellular automaton is not possible, given the number of possible automata variations. Nevertheless, we propose a quite flexible framework that accommodates any cellular automaton that complies with the following definition:

Definition

A d-dimensional grid cellular automaton is represented by a 4-tuple

$$CA = (Z^d, N_r, S, g)$$

where

(i) Z^d is the *geometry* of the discrete cellular space defined as a finite grid of d-tuples of integer numbers that represent cells z. In this cellular space, $C(z)$ maps the cell z to its coordinates and $\|\ldots\|$ denotes the standard Euclidean norm. Thus $\|C(w)-C(z)\|=1$ if the cells w and z are adjacent to each other. Z^d is a finite grid of size $n_1 \times n_2 \cdots \times n_d$.

(ii) N_r is a subset of Z^d that defines the *neighborhood* of a cell z, such that

$$z \in N_r$$

$$N_r = \{q : \|C(q)-C(z)\| \le h(r)\}$$

where r is the neighborhood *radius*, and $h(r)$ determines the topology and size of the neighborhood, which do not change during the entire simulation time. The cardinality $|N_r|$ is the *size* of the Neighborhood. Figure 1 illustrates the most usual neighborhood types: von Neumann (a diamond-shaped neighborhood) and Moore (a square neighborhood).

(iii) S is a non-empty set of possible *cell states*, where a cell state is defined by a tuple of p state variables $s = (v_1, v_2, \cdots, v_p)$, which can be continuous or discrete-valued variables. In most of the cases there is only one state variable v_1, and this variable has discrete values, such as v_1 as a Boolean variable (0 or 1). Another common case is v_1 as a 3-value variable (–1, 0, 1).

(iv) g is a *local state transition function* to be applied synchronously to every cell z taking into account the states of all cells in the neighborhood of z, at a discrete instant of time t, and producing a new state for z at time t+1 (Figure 2), that is,

$$g : S^{|N_r|} \to S$$

This representation supports the evolution of the cellular automaton as a sequence of configurations $\langle {}^t c, {}^{t+1}c, {}^{t+2}c, \cdots, {}^{t+k}c \rangle$, where the *configuration* ${}^t c$ is the set of cell states s at a discrete instant of time t.

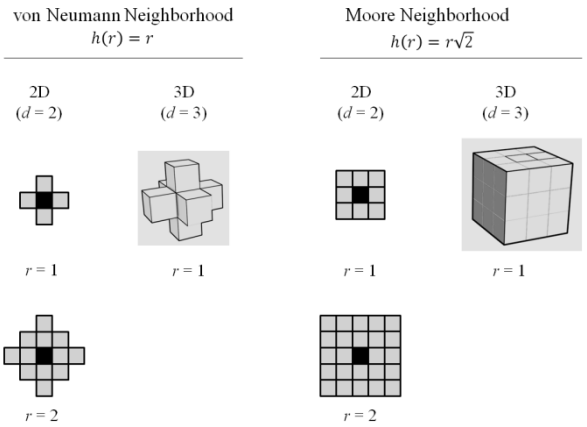


Figure 1. Common types of CA neighborhoods

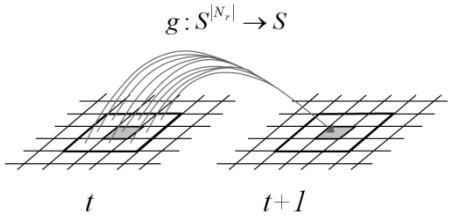


Figure 2. Local state transition function g updating the state of the 2D cell z with von Neumann neighborhood N_1

According to the definition above, a cellular automaton is a discrete dynamical system, whose behavior is governed by the transition function *g*. Groups of functions that lead to similar dynamics determine a class of cellular automata (CA). There are different proposals for CA classification [9,10], but we focus our attention on the following three cases, which can occur after a finite number of steps *k*: (a) evolution leads to a stable configuration, (b) configuration evolves to a periodic configuration (*e.g.* ${}^{t+k}c = {}^{t+k+2}c$ and ${}^{t+k+1}c = {}^{t+k+3}c$, which corresponds to a cyclic repetition of configurations with period = 1), and (c) evolution leads to a chaotic pattern.

The following cellular automata attributes are implied from the above definition:

- finite regular grid
  Z^d is represented by a matrix $n_1 \times n_2 \cdots \times n_d$, and we use the indices as coordinates (easily calculating Euclidean distances). Our framework can be used to deal with pixels of an image, voxels from a 3D scan or sites on a chessboard. Our definition of CA does not consider geometry as being arbitrary graphs. The adoption of a finite grid requires that boundary conditions be established (see Section 2.3).
- constant neighborhood
  The type and size of the neighborhood do not change amongst cells and during the CA evolution.
- deterministic
  The CA starts from an initial configuration and proceeds deterministically by successive applications of a deterministic transition function.
- uniform
  All cell states are updated by the same transition function. Some problems can only be solved by non-uniform CA in which each cell may have different transition functions [11].
- history independent
  The state transition does not depend on the history of the state transitions at the nodes and/or at its neighbors. However, the present framework allows that the current time *t* be considered by the transition function *g*.
- synchronous
  All cell states are updated simultaneously, that is, all cells are synchronized with a global clock and executed at the same time. This is a serious restriction, because many real system models require asynchronous agents. However, the theoretical foundations of asynchronous CA are not yet understood satisfactorily, and few works are available in the literature [9, 12].

Sometimes we identify the existence of a particular state $\bar{s} \in S$, called the *quiescent state*, such that if all neighbors of a cell *z* are quiescent, then the cell will also become quiescent. That is,

$$g(\bar{s}, \cdots, \bar{s}, s_z, \bar{s}, \cdots, \bar{s}) = \bar{s},$$

where s_z is the state of the cell *z* before the application of *g*. The configuration in which every cell is in the quiescent state is called the *quiescent configuration*.

2.2 Characteristics of the Transition Function

The user of the present CA framework is totally responsible for the definition of the transition function. He/she should remember that the function *g* determines the behavior of the CA, especially those related to convergence and invertibility. In our framework, we are mainly concerned with convergent solutions (stable or periodic). The CA is considered reversible if we can recover a configuration at any past instant of time. If there is an inverse rule g^{-1} that drives the cellular automaton back to previous configurations, the user can inform the framework of this and proceed backwards (please note that reverting the direction requires starting a new simulation in the framework). Reversibility is a natural problem that arises in physical systems. However, for dimensions $d \geq 2$, there is no general way of deciding whether a given transition function is reversible or not. A discussion on the invertibility of *d*-dimensional cellular automata can be found in [13].

2.3 CA Boundary Conditions

Boundary conditions are necessary to either simulate infinite grids or to represent natural boundaries of a real system model. These conditions are incorporated into our framework by including extra layers of cells, with a width equal to

the neighborhood radius *r*. In the present framework, boundary conditions can be fixed, periodic, or reflective. We detail each one below.

In the *fixed boundary condition*, the boundary cells have neighbors in the extra layers. These cells have a pre-specified state value that does not change during the entire simulation (value *w*, depicted in Figure 3a). This type of boundary condition corresponds to a Dirichlet boundary condition in partial differential equations. The fixed boundary condition is called a *null boundary condition* if the fixed state value is the quiescent state, *i.e.*

$$w = \overline{s}$$

In the *periodic boundary condition*, extreme opposite cells become neighbors, like a fabric being folded in one dimension. In practice, we can do this by copying the extremity cells to the opposite boundary layer at each time step (Figure 3b). In a 2D cellular automaton, if we do this in only one of the dimensions, we shall have the topology of a cylinder (as in the left grid in Figure 3d). If the 2D grid is folded in two dimensions, we shall have the topology of a torus (as depicted in the right grid in Figure 3d). Periodic boundaries are used as an approximation of infinite grids.

In the *reflective boundary condition*, the extreme cells are mirrored at the boundary. We can do this by copying the extremity cells to their adjacent boundary layers (as shown in Figure 3c). This type of boundary corresponds to a zero-flux boundary condition that constrains the solution of partial differential equations used in continuous diffusion models.

Boundary conditions can be combined, such as the long channel depicted in Figure 3d, with a periodic condition in one dimension and a reflective condition in the other.

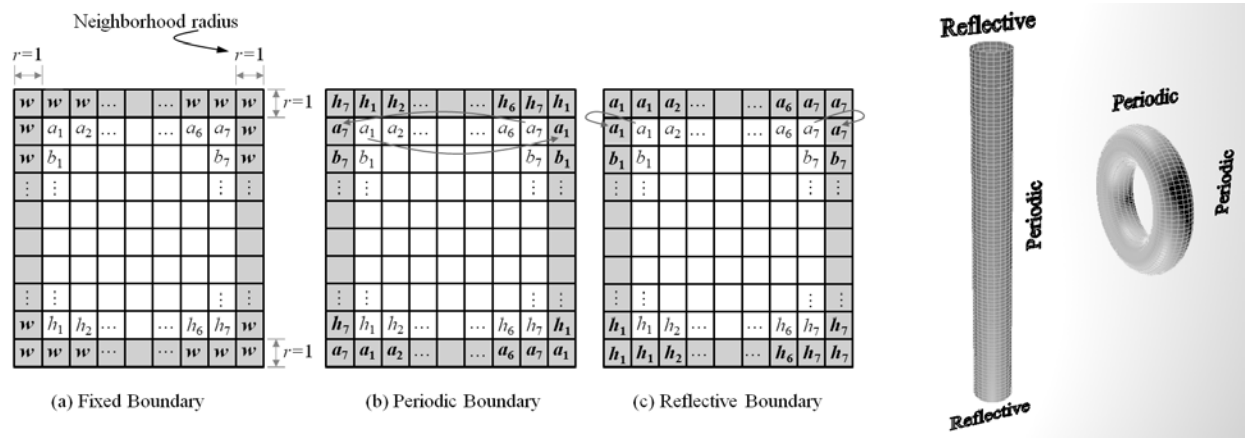


Figure 3. Examples of boundary conditions for 2D cellular automata

3. CLOUD COMPUTING

3.1 General Concepts

Cloud computing is a common expression to designate the delivery of computing as a utility [14, 15,16,17,18,19]. It refers both to the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide such services. Its dominant business model is the pay-as-you-go, in which customers contract and pay for a plethora of stratified computing services provided by commercial cloud computing providers [20]. Examples of the last are Amazon Web Services (from EC2 hardware resources to high-level services) [21, 22], Google App Engine [23], Relational Networks LongJump [24], Salesforce Force.com [25], and Microsoft Windows Azure [26].

As pointed out by Armbrust *et al.* [14], Microsoft Azure is an intermediate point on the spectrum of flexibility versus programmer convenience which ranges from the straightforward use of hardware virtual machines (as EC2) to the application of fully functional frameworks such as Google App Engine. On-demand software services are also located on the extreme point of this spectrum, which characterizes the business model of Software-as-a-Service (SaaS). Azure supports general-purpose computing, rather than a single category of applications. These characteristics make Azure an adequate platform for the cellular automata framework proposed by this paper. However, this does not mean that Azure is the best platform for everything, some types of problems are more conveniently served by other cloud computing services, or even by a combination of them (such as App Engine hosted on top of EC2).

3.2 Microsoft Azure

Microsoft Azure belongs to a class of cloud computing frameworks commonly referred to as Platform-as-a-Service (PaaS), as opposed to Software-as-a-Service (SaaS). In Azure, it is possible to opt for specific programming languages as well as development environments (IDEs); however, users cannot control the underlying operating system. Azure uses the notion of "roles" as a model of computation and networking, and that of "blobs, queues, and tables" as a model of storage.

A typical Azure application consists of one or more roles, each responsible for running one or more CPUs. There are only two types of built-in roles: Web Role or Worker Role. A Web Role is comparable to a Web application, whereas a Worker Role is a background service that runs some periodic, cyclical task. When creating a new role, users need to specify the number of CPU cores (which are defined by the size of the chosen Virtual Machine, *e.g.* size = medium equal to 2 cores at the time this article was written) and the number of CPUs (referred to as instances). Each instance of a worker role typically takes care of a combination of services and makes use of a large number of workers (*i.e.* processing units) to execute them. The proposed framework makes use of one Web Role, one Worker Role, and several of their instances.

The Azure storage system, a combination of blobs, queues, and tables components, may be used to implement synchronization and communication between workers. Blob (binary large object) storage is a direct way of storing large binary data, such as the cellular automata matrix of our framework. Blobs can be mapped as a device and work as a folder.

Queues store small job messages and are used as a communication mechanism between applications and services. Workers regularly ping the queues to acquire a message and run a job. Queues are designed to be reliable and persistent (*e.g.* an acquired message is sent back to the queue if a worker fails).

Tables are used to store data that needs additional structure, but that do not require being kept in relational databases.

4. PROPOSED FRAMEWORK

4.1 System Constraints and Underlying Strategy

Although we have considered a general formulation for cellular automata, the proposed framework is constrained by the following simplifications:

- only 2D and 3D grids are allowed (*i.e.* 1D cellular automata are not implemented);
- parts of the code and storage schema are optimized for two-state bidimensional cellular automata, *i.e.* the cases in which $s = (v_1)$, and v_1 is a Boolean variable (0 or 1).

The present CA framework is inspired on the Map and Reduce model presented by Dean and Ghemawat [27], in which the input matrix is partitioned into separate blocks that are subsequently distributed and processed in parallel by different units (mappers), and the results are then combined, or reduced, by a separate unit. The local nature of the CA transition rules lends itself well to the use of the Map and Reduce strategy.

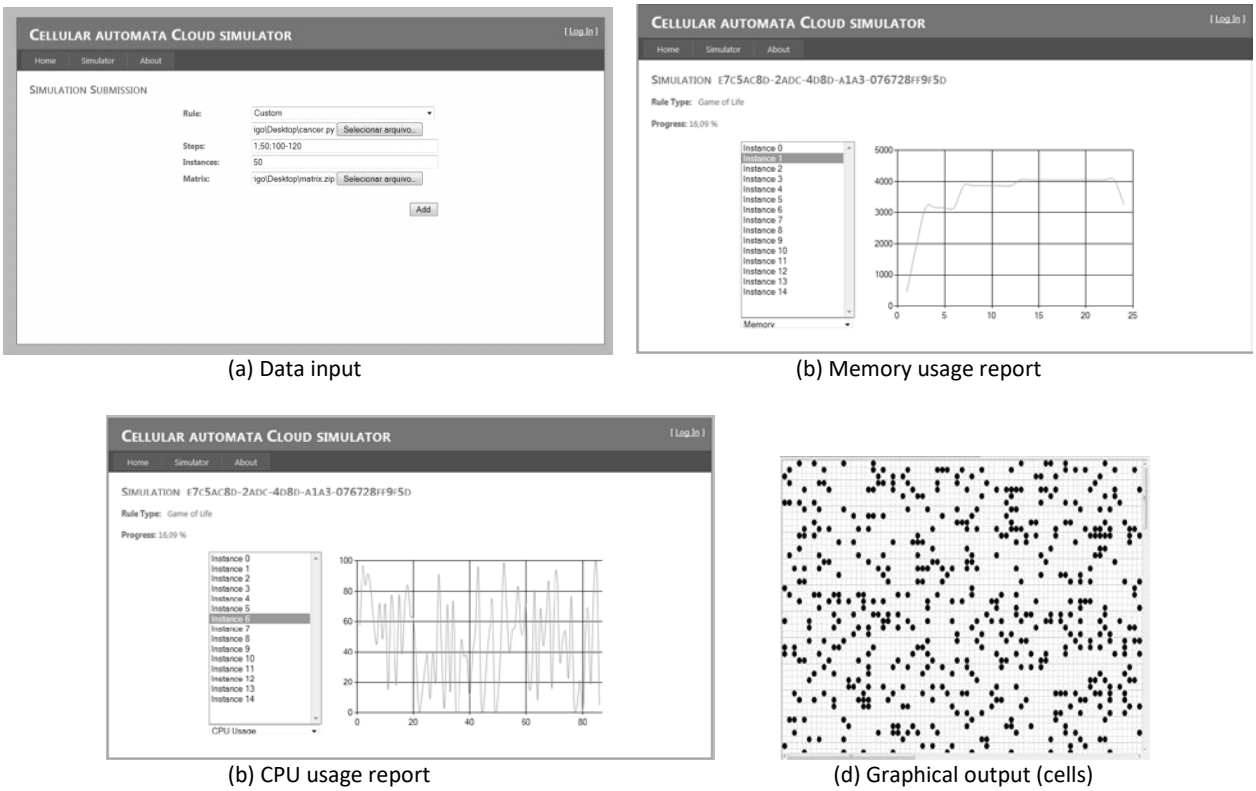
Our architecture, in particular, makes use of two types of units, also called Azure roles:

- MapReduce Worker Role
- Web Role

The MapReduce Worker Role provides three services:

- Map Service
- Reduce Service
- Cleaning Service

The Web Role (1) implements the user interface, (2) provides a means of submitting simulation data, (3) tracks results, and (4) sets up the simulation in the cloud. Figure 4 illustrates the Web Role user interface for a 2D cellular automaton with 10^{12} cells. This Azure role also provides the necessary tooling to prepare the simulation matrix for transmission over the Internet. The MapReduce Worker Role implements the Map and Reduce strategy, as envisioned in [27]. The remainder of Section 4 details the main activities of the proposed framework and discusses its overall architecture.



(a) Data input (b) Memory usage report

(b) CPU usage report (d) Graphical output (cells)

Figure 4. Example of web role UI for a 2D cellular automaton

4.2 Data Preparation

Figure 5 illustrates the data stream both in the local computer and the connection with the cloud. If the simulation matrix A is sparse, then A is transformed into a compressed sparse matrix to improve the transmission process. The process for compressing sparse matrices is described in Section 4.3. The final structure of A is serialized into a file

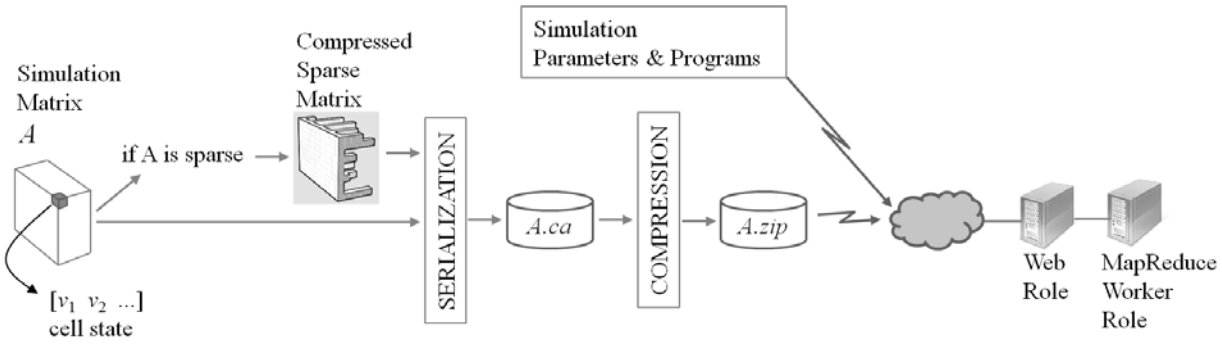


Figure 5. Data stream in the local computer and the connection to the cloud

called *A.ca*, which is submitted to a zip compression process and then transmitted to the cloud as the file *A.zip*.

The user should also provide the following simulation parameters and program:

- grid dimension d: 2D or 3D
- size of the simulation matrix: (n_1, n_2) or (n_1, n_2, n_3)
- file containing the simulation matrix A
- number of instances (CPUs) n_{cpu}
- number of cores n_c: 1, 2, 4, 6, or 8
- time steps to be stored during simulation in the format $range_1;range_2;...$, where $range_i$ is a single instant of time or a range $t_{initial} - t_{final}$ (*e.g.* 1;50;100-120)
- number of state variables p
- type of neighborhood: von Neumann or Moore

- neighborhood radius r
- quiescent state $\bar{s}$ (default is 0)
- boundary condition: fixed, periodic, reflective
- local state transition function g in the form of a text file containing a program written in a script language. The language is automatically detected (currently, Python and Lua only). The signature of g is

  ```
  [v1 v2 ... vp] g(p,i1,i2,i3,Nr,t)
  ```

  where `Nr` is the matrix representing the neighborhood of the cell that has integer coordinates (`i1`,`i2`,`i3`), `p` is the number of state variables, `[v1`,... `vp]` is a list of state variables, and `t` is the current time.

4.3 Sparse Matrices

There are several static and dynamic storage schemes for sparse matrices [28,29], but only dynamic ones are suitable for cellular automata, because CA matrices do not comply with any specific, standard storage optimization scheme. Most of the dynamic storage schemes are variants of the orthogonal linked-list scheme proposed by Knuth in 1968 [30]. In this paper, instead of using arrays of integers to describe the locations of the non-zero elements in the original matrix, we use circular linked lists for both rows and columns. This is far from the efficiency of the most common storage schemes found in well-known math libraries (*e.g.* Intel Math Kernel Library [31]), but it is an adequate scheme for the proposed system, which deals with frequent modifications of cells and requires fast node to node access. Moreover, we are not concerned with the high time complexity to compress the original matrix (which can be $O(n_1^3)$ for an $n_1 \times n_1$ matrix). Indeed, we focus on space reduction rather than rewarding processing time, because the size of the simulation matrix is a serious limitation for communicating with the cloud.

Figure 6 illustrates our storage scheme for a 2D cellular automaton, that is, a matrix with dimension $d = 2$ (Figure 6a). In this scheme, each dimension of the matrix is represented by a circular linked list of head control nodes (shaded nodes of Figure 6b). Each non-null element in the matrix is a node with d pointers (Figure 6c) to link with the next non-zero element (if there is no non-null element next to it, it points to itself).

A sparse matrix is a matrix with a large number of zero elements that allows us to save time and/or memory by exploiting the zeros. Therefore, this definition depends not only on the matrix itself, but on the algorithm used on the

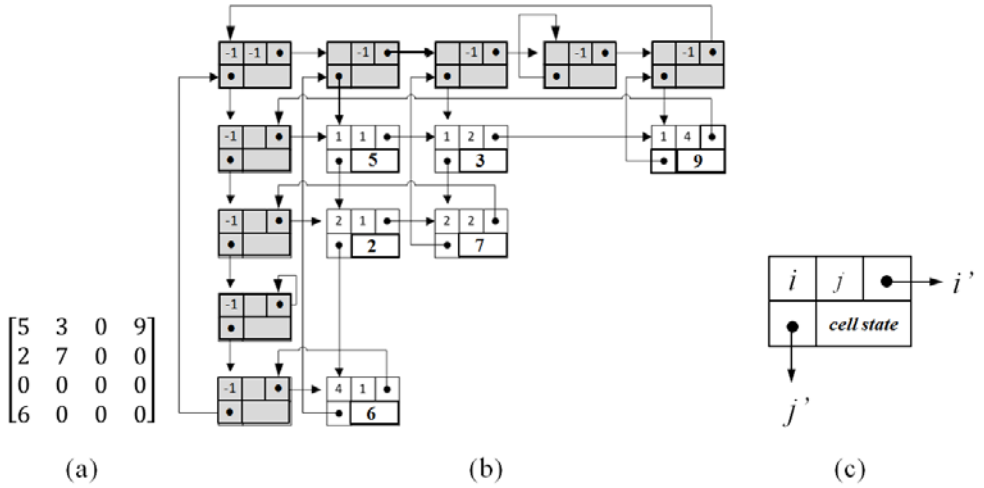


Figure 6. 2D representation of a compressed sparse matrix.

matrix [32]. In the proposed framework, a matrix is considered sparse if the number of zeros effectively permits the reduction of storage space (considered later). Therefore, we compute the stored size of the matrix to decide whether it is worthwhile to compress the matrix. The stored size of a $n_1 \times n_2 \times n_3$ matrix is $n_1 \times n_2 \times n_3 \times$ *sizeof*(*data*) and a sparse matrix with m non-null element needs, at most, $6 \times (n_1+n_2+n_3+m) \times$ *sizeof*(*data*) elements (because each *data* needs three pointers and three dimension indices). Thus

$$6 \cdot (n_1 + n_2 + n_3 + m) << n_1 \cdot n_2 \cdot n_3$$

$$m << \frac{n_1 \cdot n_2 \cdot n_3}{6} - (n_1 + n_2 + n_3)$$

In our system, if $m < [((n_1 \cdot n_2 \cdot n_3)/6) - (n_1 + n_2 + n_3)]$ we shall compress the matrix, otherwise we shall store it in serial format. It is important to emphasize that sparse matrix compression is the final memory footprint to access matrix data, because Huffman's techniques only reduce the size for storage and transmission; we still need to decompress the matrix to access the sparse matrix information.

4.3.1 Sparse Matrices Serialization

The serialization of a sparse matrix structure into a file needs to comply with the following requirements:

- access to a specific cell without loading the whole structure.
- data coherence for cells in the same row.

The data structure of a cell (defined in Section 4.4) uses three pointers to represent a 3D matrix, so, when we write it to a file, we need to change the pointer offset value to the file offset value. We create a table with the missing cell offsets; thus, when we write a cell and we cannot put the value for the row neighbor, we create a key with the neighbor cell index *(x,y,z)* and put the current cell offset as the value. If the key already exists, we append the current cell offset to the value. When we write a cell, we check if it is in the table key list, and if so, we iterate over the cells offset stored on key value, changing the missing cell offset value to the correct offset value, and clear the key value data. The data structure in the memory (in the compression process) has a field to mark if a cell is written and another field to store the offset value.

This process produces a fast way to find cells, even on disk, so we do not need to load the matrix to produce a submatrix. We also implement the matrix operations for sparse matrices described in [33].

4.4 Partition of the Simulation Matrix

The system subdivides the simulation matrix into N submatrices according to the number of instances (n_{cpu}). Each submatrix *K* should be properly expanded (decompression) to allow for parallel processing of the N submatrices.

Figure 7 illustrates the matrix subdivision process, in which we split the matrix into cubic cells

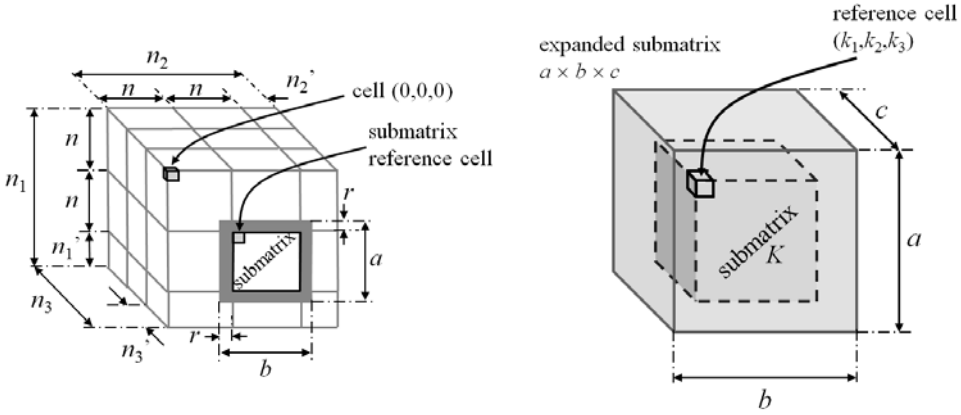


Figure 7. Partition of the simulation matrix $n_1 \times n_2 \times n_3$ and expansion of the submatrices.

based on the following equations:

$$n_{cpu} = trunc\left(\frac{n_1}{n} \cdot \frac{n_2}{n} \cdot \frac{n_3}{n}\right)$$

$$n = truncPow2\left(\sqrt[3]{\frac{n_1 \cdot n_2 \cdot n_3}{n_{cpu}}}\right)$$

where *trunc* truncates its argument to an integer value, and *truncPow2* truncates its argument to a value that is a power of 2.

Using this approach we can explore the process locality, because each submatrix must be generated and sent to the worker, and this consumes a lot of time. So, we send the maximum possible size for each worker. The final submatrix needs to contain its neighborhood, so we expand it based on the neighborhood radius *r*; thus the submatrix size is $a \times b \times c$ (Figure 7), and normally $a=b=c=(n+r)$. Another consideration is the boundary condition, which needs to be respected in the expansion.

Due to the truncation process in calculating *n*, some small submatrices (with dimension not equal to *n*) appear (n_1', n_2', and n_3' in Figure 7), which we call *leaks*. Each *leak* is processed when a worker completes its work with a regular submatrix. We are aware that cube submatrices are more suitable for parallel execution, mainly on GPUs.

After we split and expand a submatrix, it receives a registration string [*k1,k2,k3,a,b,c*] that describes the submatrix (Figure 7). The first three numbers of this string are the *reference cell* indices of the submatrix *K* (this cell is not the *(0,0,0)* cell in the submatrix due the neighborhood radius), and the last three numbers are the size of the expanded submatrix.

The split process mentioned above does not require the whole matrix to be loaded; instead, we load the submatrices in blocks with common neighborhoods.

4.5 Proposed Architecture

The communication architecture of the proposed CA framework in the cloud is presented in Figure 8. Furthermore, in Figure 8 we propose a special graphical notation, together with specially designed components, to represent roles, services, workers, storage models, and communication actions.

The proposed graphical notation has the following characteristics:

- A role is represented by a rectangle with an outer shadow, a worker by a rectangle, a queue by an arrow with message compartments (and the first message on the arrowhead), a table by a small empty table, and a blob by a cylinder.
- A service is indicated by a closed dashed line forming a rectangular shape.
- Communication actions are indicated by a dashed arrow for control messages, a double arrow for data streams, and a single arrow for creation/access/update processes.

The Web Role performs the following tasks:

- creates the Azure table **`CaJobTable`** and inserts the first line of information in it.
- creates the Azure queue **`CaJobQueue`** and continues inserting job messages in it, like the first job to be done or a cancelation order.
- creates a blob containing the file of the compressed simulation matrix *A* (Initial Configuration Blob) and names it with the MD5 [34] hash value of the file (<*md5*>Initial).

The **`CaJobTable`** has the following information:

- PK: “partition key” generated by a MD5 hash function of the current simulation file. PK determines the physical location (server) of the table line.
- RK: “row key” randomly generated as a GUID (globally unique identifier), that is, a unique reference string (a 32-character hexadecimal number) that identifies the line. Lines with the same “partition key” are stored in the same physical location.
- Steps: string with the time steps to be stored during simulation (see 4.2).
- StepsDone: string with progress information about each step. We use a concatenation of `"[StepID|Progress];"`, where `StepID` is the iterative step index, and `Progress` is a float point value of the step progress in %: [0,100].
- InstanceCount: number of instances used.
- Rule: string with the script code of the local state transition function *g* (see 4.2).

- Status: information about job status (Not Started, Started, Processing, Done). A job is in Started state when it starts the matrix subdivision step, and it is in Processing state when it is processing a cellular automata rule.

The first job to be done is inserted in the **`CaJobQueue`** as the following string: `PK+";"+RK`

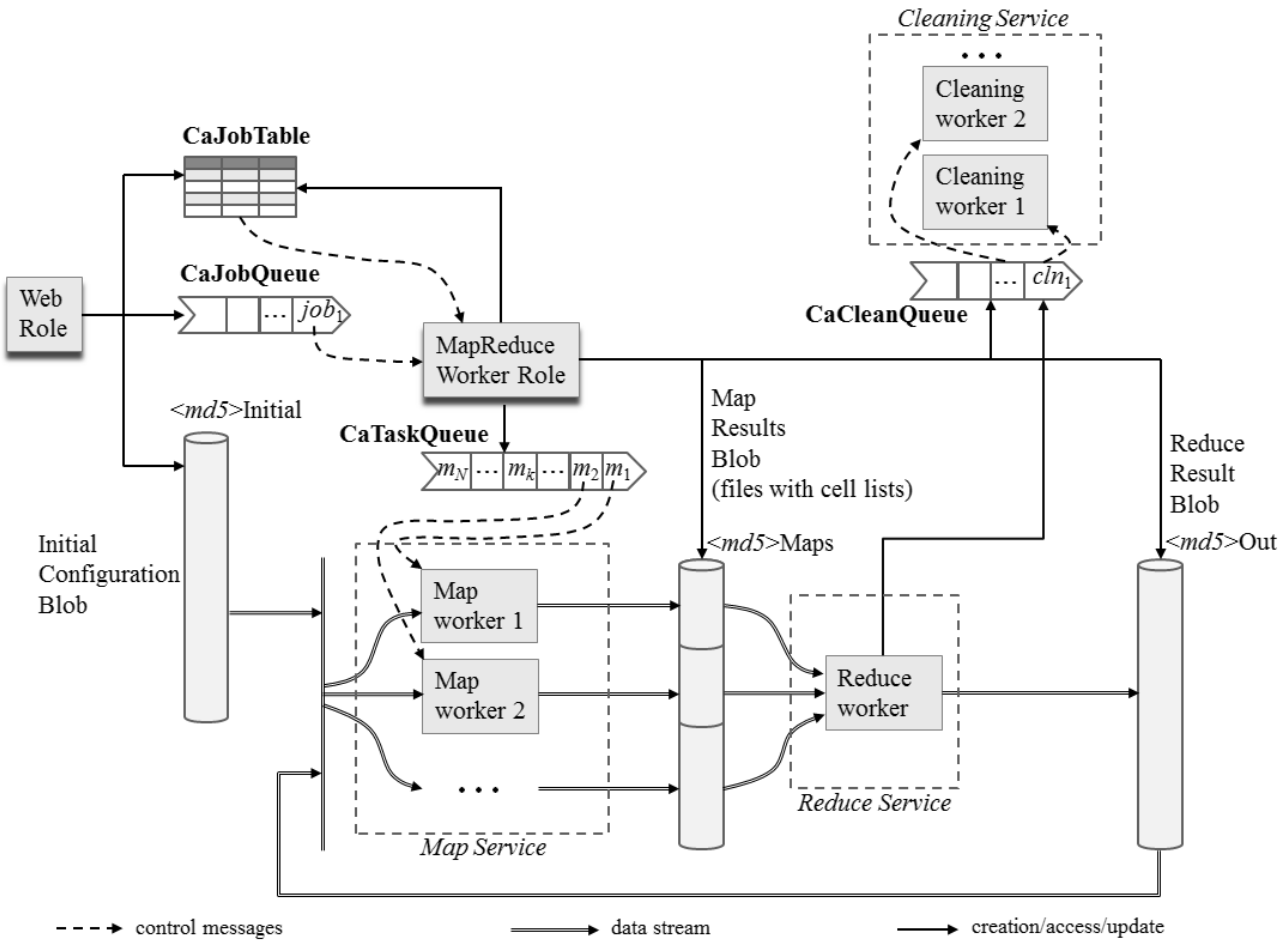


Figure 8. The framework communication architecture

The Base64 encoding of this string is sent to the user as a key for tracking the simulation process.

The MapReduce Worker Role performs the following tasks (Figure 8):

- creates the Azure queue **`CaTaskQueue`** after communicating with the Web Role.
- creates two blobs: a blob for the map results (called <*md5*>Maps), which contains files with transformed submatrices, and a blob for the reduce result (called <*md5*>Out). These blobs are mapped as a device in the workers to simplify the implementation and accelerate I/O tasks.
- creates the Azure queue **`CaCleanQueue`**, which is used to clean the Map Results Blob.
- updates the Azure table **`CaJobTable`**.
- continues communicating with the Web Role through **`CaJobQueue`** and **`CaJobTable`**.

The MapReduce Worker Role acquires the first job message and creates the **`CaTaskQueue`** with a message for each split (*i.e.* each submatrix) of the simulation matrix *A*. The partition process that generates each submatrix is described in Section 4.3. As a Map Service, workers acquire the task messages, perform a state transition for each submatrix, and store the results in <*md5*>Maps.

The Reduce Worker combines all submatrices and creates a final file with the compressed sparse matrix on <*md5*>Out folder, when all submatrices have been processed and stored. After this, it puts a message in **`CaCleanQueue`**. The workers of the Cleaning Service acquire messages from **`CaCleanQueue`** and clean simulation temporary data, such as old submatrices files and temporary files.

4.6 Simulation Process

The MapReduce Worker Role puts a message in **`CaTaskQueue`** for each generated submatrix with the following string:

```
PK+";"+RK+";"STEP+";"+I+";"+J+";"+K+";"+W+";"+H+";"+L
```

where *I, J, K, W, H,* and *L* are the elements of the submatrix registration string (described in Section 4.4), and *STEP* is the simulation step id. Map Workers wait for messages in the **`CaTaskQueue`** and, when they pop a valid message, they start the state transition for each cell in the submatrix.

The process starts decoding the message to load the submatrix. The sparse matrix file is available for all workers and can only be loaded as read-only, so we can have lot of workers accessing it without any problem. Each worker allocates enough memory to store the submatrix and starts to populate it loading each cell from the file in the compressed format. The load request process is an asynchronous call, which helps the loader on the server and increases the cache performance (what is done by the operation system). Each worker has more than one core, so even the load process is parallelized at the CPU level. Therefore each core loads a block in the submatrix. It is important to emphasize that the inter-block linking is thread-safe, and it is done after all cells are loaded.

After loading the submatrix, the worker performs iterations over every cell, discarding neighborhood cells. The resulting matrix is temporarily stored as a linked list, with non-null cells, after simulation. That greatly simplifies the process of recreating a compressed sparse matrix. The simulation algorithm is presented in Figure 9.

```
1   r ← neighborhood_radius
2   mo ← submatrix
3   t ←time_index
4   cells ←create list -- creates an empty linked list
5   parallel for  id ← [0, (#row-2*r) / ncpu) do
6      for  i1 ← [r + row_0 + id * (#row-2*r)/ ncpu,  row_0 + (1 + id) * (#row-2*r) / ncpu) do
7           for i2 ← [r + col_0, col_0 + #col-2*r) do
8               for i3 ← [r + dep_0, dep_0 + #depth-2*r) do
9                   Nr = getMatrix(mo, i1, i2,i3, r, r, r)
10                  val ←g(p,i1,i2,i3,Nr,t)
11                  if val ≠ null then
12                    cells.add(val, i1, i2,i3)
13                  end if
14              end for
15          end for
16     end for
17  end parallel for
18  result ← compress_list(cells)
```

Figure 9. Simulation algorithm

In the code of Figure 9, the variables *row_0*, *col_0* and *dep_0* are the indices of the first submatrix cell (considering the radius *r*), so we need to translate it to the first cell that needs to be simulated. Moreover, *#row*, *#col*, *#depth* are the simulation matrix size. Similarly, we decrease the size by *r* to get to the simulation matrix size. After we iterate over cells, we split the simulation amongst n_{cpu} processing units (CPUs).

Each cell creates a neighborhood matrix *Nr*. So, given a cell to be simulated, *getMatrix* creates a matrix centered in (*i1*,*i2*,*i3*) with radius *r*, based on the compressed submatrix *mo*. Finally, we call the transition function *g*, which is the code given by the user, and test if the transition produces a non-null cell. If it does produce a valid cell, we store it in the *cells list*. The *cells list* is a thread-safe list.

After processing the entire matrix, we compress the list using a zip compression algorithm and save it in the temporary blob <*md5*> Maps. We don't need to create a compressed sparse matrix, because it is a temporary result. The file name is the string <*md5*>-<*time*>-<*submatrix reference string*>, where <*time*> is the simulation step.

4.7 Reduction Process

The Reduction process starts after all submatrices have been processed and stored. This process uses asynchronous events for folder file changes on the Reduce Worker. When this worker starts to reduce, it combines the submatrices into a compressed sparse matrix structure. Each submatrix file is a zip compressed list of non-null cells. Thus, for each file, we sequentially insert the non-null cells into the compressed sparse matrix structure.

The system can operate in two ways. A one level reduction consists of a service that processes all non-null cells and generates the final matrix file. The other way is the multi-level reduction (Figure 10), in which we start *n* role services, which reduce a submatrix and create a temporary sparse matrix file. In this second way, we start *n/2* role services to combine the submatrices and generate the next step. We continuously repeat this process until we have only one thread that finishes the submatrices bindings. Each reduction role service has an *ID* when it finishes the

processing step. A reduction role service stays in an active state only if its *ID* belongs to a valid *ID* sequence, which is only if *ID* < *Step Number / 2*. This process needs $log_2(n)$ steps to complete, but if the matrix is really large we can take advantage of parallel processing.

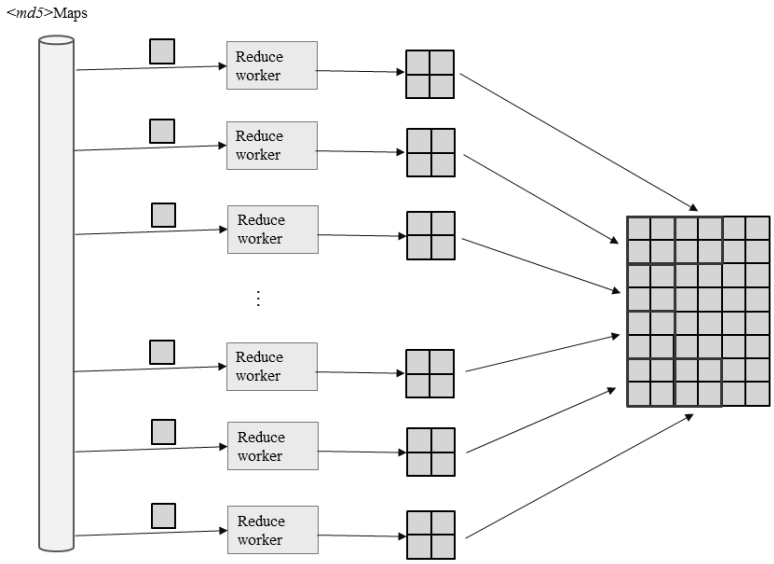


Figure 10. Multi-level reduction process

Finally, we save the matrix on <*md5*>Out folder and put a message in **`CaCleanQueue`** with the string <md5>-<*time*>. The MapReduce Worker starts to split the next simulation step when an asynchronous event detects the new file on <*md5*>Out folder.

4.8 Cleaning Process

Each file stored in a cloud server requires space and, therefore, costs money. It is desirable, therefore, to reduce the overall number of files, especially when they are used for storing intermediate results. To this end, we have devised a specific role service to clean temporary files. The cleaning process gets messages from **`CaCleanQueue`**, decodes them, and starts removing temporary files.

Even files that contain simulation results may be removed if they are not in the Steps list, but again, they can only be removed if the step result is not necessary for the final reduce step.

4.9 Cancelation Process

Sometimes we notice that something is wrong with the simulation and we want to cancel it. However, a cancelation process requires that all involved workers be informed. In the proposed architecture, the user can cancel a job using the Web Role interface, forcing the Web Role to send a message to **`CaJobQueue`**, with a high priority flag assigned to the following message string: `(PK+";"+RK+";CANCEL").` Due to its high priority, this message will be the next one processed.

When the MapReduce Worker decodes the message, it puts the same message in **`CaTaskQueue`**, with the high priority flag set. This will force the workers to pop the cancelation message next time they try to get a message from the queue. This procedure works for the first worker only, because it is the only one that knows about the cancelation of the Job. Thus, the Map Workers must be capable of communicating amongst themselves.

The Map Workers can communicate with others workers using MPI (Message Passing Interface) [35] or WCF (Windows Communication Foundation) [36] contracts. In this way, when a worker decodes a cancelation message, it sends a broadcast message to the other workers, the Job is stopped, and the Cleaning Service starts to clean the Jobs Files. Azure allows the creation of Internal Endpoints to inter-role communication, and we can select the protocol and the port number. We only use asynchronous events to listen on the port.

5. CASE STUDY

5.1 Game of Life

As mentioned above, we tested the 2D case extensively. Moreover, we focused on cases that demanded the manipulation of very large matrices (10^6 x 10^6 matrix cells), which require both robust and efficient strategies for storage and processing. Our goal is to investigate the limits of performance when doing massive simulations. The case we describe here is an analysis of the Game of Life, a two-state bidimensional cellular automata that can simulate several phenomena. It is a very popular cellular automaton, and it is very simple to implement and understand.

The Game of Life is based on the following four simple rules:

- Any live cell with less than two live neighbors dies due to loneliness.
- Any live cell with more than three live neighbors dies due to overpopulation.
- Any cell with exactly three live neighbors comes alive.
- Any cell with two live neighbors stays in the same state for the next generation.

5.2 Initial Configuration

We create an initial 10^6 x 10^6 matrix (10^{12} cells) by writing a script that randomly generates a matrix of zeros and ones. The script creates a text file with each cell represented by a 0 or 1 (for instance, a 3x3 identity matrix is the string 100010001). Also, we can define the live probability of the matrix (that is, the probability that a cell is alive). Using the conversion tool integrated to the web role, we converted the matrix and uploaded it into the cloud.

5.3 Tests

The tests were designed with the following objectives in mind: to determine how fast the speedup of instance numbers is, to measure the influence of the number of cores per instance, and to evaluate the overall effectiveness of the proposed solution. In particular, we would like to know the value of instance numbers from which speedup does not significantly increase due to the Map and Reduce process, and node intercommunication. Another point of interest was to estimate how effective the sparse matrix compression actually was. With this last goal in mind, we created a series of initial state matrices (100 for each pair size and life probability) with varying life probability values, and compressed each one of them to evaluate the effectiveness of the compression process.

We created a series of simulations, changing the number of instances and the number of cores, and we collected time information for the Map, Simulation, and Reduce processes. Also, we registered the amount of used memory and space. Furthermore, we measured the time to transfer files to the Initial Blob.

We used the following test sequence:

- sparse compression efficiency
- file transfer time over the Internet
- speed up analysis over instances number X cores number
    - internal processing time

5.4 Analysis of the Results

The objective of testing the compression scheme of the sparse matrix was to demonstrate the advantage of focusing on space reduction rather than on processing time. Another point of interest was the time to transfer the file over the Internet. Internally, cloud servers typically use a 10 GBits Ethernet protocol; so, even for internal communications, we needed to reduce transfer sizes. In the compression tests, the script runs a sequence of initial states by changing the matrix life probability and the matrix size, and submits the sequence to the compression algorithm. Each test runs ten times, and the mean value was registered. The results are summarized in Figure 11 and Table 1.

Size / p_{Life}	**100**	**1k**	**10k**	**100k**	**1M**
5	7.07%	5.49%	5.51%	5.05%	5.02%
10	12.27%	10.29%	10.49%	10.03%	10.04%
20	22.49%	20.62%	20.51%	20.34%	20.01%
40	42.26%	40.49%	40.11%	40.35%	40.05%
80	82.35%	80.43%	80.16%	80.47%	80.02%

Table 1. Sparse matrix compression test results — p_{Life} is the matrix life probability

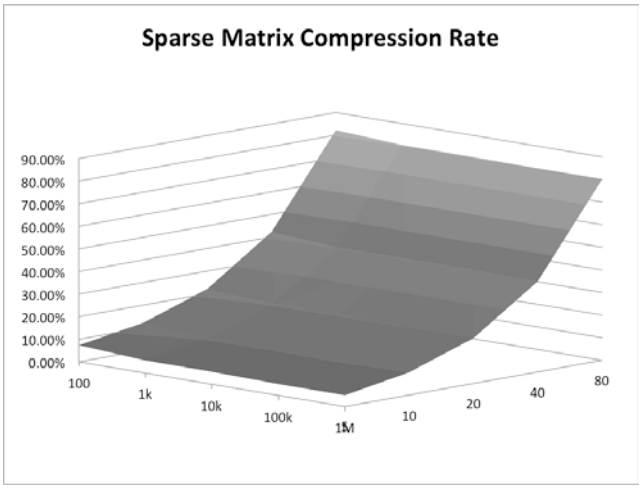


Figure 11. Graph of the sparse compression test results

Figure 11 shows that the compression algorithm works correctly, because the sequence converges to the compression ratio limit, which is given by $\lim_{size \to \infty} ratio = p_{Life}$. The ratio increases depending on the matrix size, but it is limited by p_{Life}.

We tested the file transfer time by sending zip files over five scenarios:

- Ethernet 1 Gbps
- Ethernet 10 Gbps
- Internet Link 100 Mbps
- Internet Link 10 Mbps
- Internet Link 1 Mbps

The first two options represent the internal transfer inside a cluster. Azure uses Ethernet 10 Gbps, but most local clusters use Ethernet 1 Gbps. The last three options are Internet links. Figure 12 shows the simulation results for each option. The horizontal header is the average size of the matrix, after all compressions, and for a 25% of life probability.

Size / **Interface**	**1.46k**	**103.4k**	**5.7M**	**387.6M**	**8.2G**
Ethernet 10Gbps	12 s	12 s	22 s	64 s	2.7 min
Ethernet 1Gbps	12 s	12 s	29 s	1.6 min	7.7 min
Internet 100Mbps	12 s	12 s	35 s	4.7 min	51.4 min
Internet 10Mbps	17 s	23 s	57 s	28 min	623.2 min
Internet 1Mbps	17 s	28 s	4.7 min	3.9 h	> 1 day

Table 2. File transfer test results

The results in Table 2 show that it is very important to reduce the file size before sending it over the Internet. Giant matrices, e.g. 10^6 x 10^6, take at least one hour to upload. Due to the excessive time required to upload test files using 1Mbps Internet Link, we canceled the process after one day of transfer time. The internal transfer works well at high speed connections; therefore, the time used for internal transfer, in most cases, is not a serious problem, because we transfer only small submatrices blocks between nodes. Another observation is that the Azure storage system can work with a 10 Gbps transfer rate, which is important because most storage systems use SAS connections that are limited to 6 Gbps.

Once we have the initial state file on the server, we can start the simulation process. After uploading the 10^6 x 10^6 matrix, we can test the combination of number of instances and number of cores per instance.

The Map process is a very fast procedure, because it only needs to know the matrix size and the number of CPUs to split and create the messages to push on the task queue. During the Map processing tests, we measured the mean time to split and push messages. Figure 12 shows this mean time by varying the number of CPU instances.

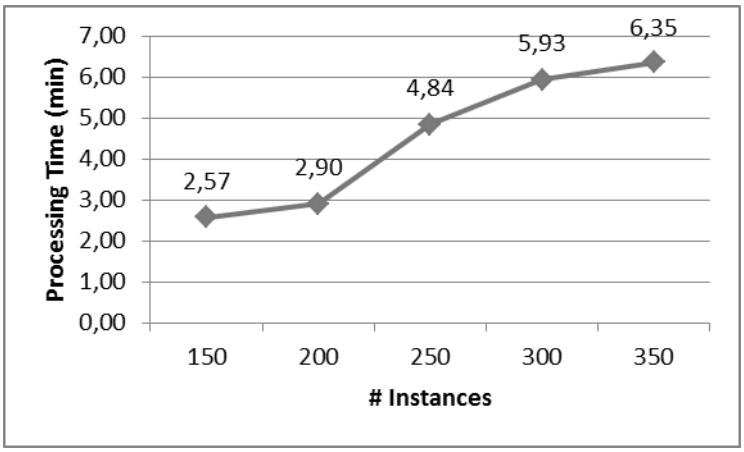


Figure 12. Map procedure split time chart

In these tests we are not changing the number of Map workers, but only varying the number of instances necessary to subdivide and create a message. After the submatrices were mapped for each test, we collected the processing times for the process state (Table 3).

Processing Time (days) – Processing State					
Instances / Cores	**150**	**200**	**250**	**300**	**350**
1	> 10	> 10	> 10	> 10	> 10
2	> 10	> 10	11.57	9.65	8.27
4	10.83	8.81	2.89	2.41	2.07
8	28.94	21.70	0.72	0.60	0.52

Table 3. Processing time

The process state needs to build the submatrix and process each individual cell. Thus, it starts reading the matrix file on storage and creating the sparse matrix on local memory. This procedure is asynchronous and parallel, so each thread builds a subblock in the submatrix and combines it with the full matrix structure. This procedure is thread-safe blocking. The reading operation is made in a mapped local disk, so the operational system combines and handles the requests. In a high-speed network, it is very important to create a request buffer to reduce the request latency.

The results show that the time taken to produce the results is a question of how many cores we select to run, and that they range from one to ten days to process. We spent, on average, ten minutes to start up the machines. The subdivision process was very efficient, and the file structure used to load the matrix was a very fast way of rebuilding the matrix without loading large bounds of memory. Also, the storage speed and network infrastructure behavior proved appropriate for this type of use.

Finally, we needed to reduce the matrix. This step can be done in two ways: one-level reduce or multi-level reduction.

The one-level reduction is a simple way to reduce something, in which we use a unique machine to load the cells list and build the full matrix. It is very efficient if the matrix is small, which was not the case. We tested the processing time to reduce the matrix using only one reduce worker and varying the number of cores. Figure 13 shows the results.

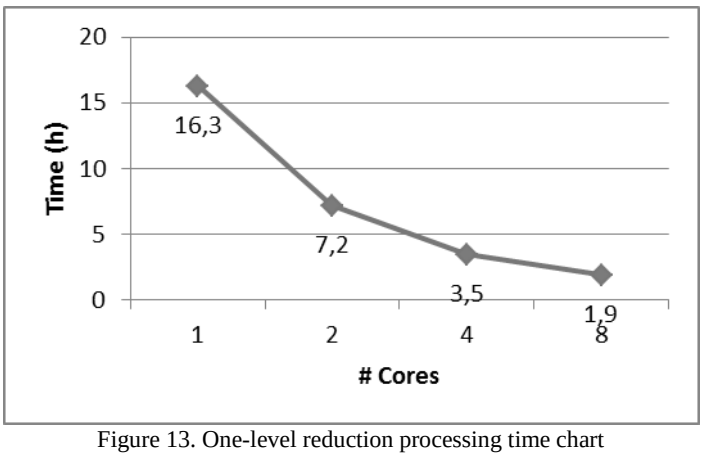


Figure 13. One-level reduction processing time chart

Obviously, the time taken with one worker and one core is very large, because it needs to load the files and process each non-null cell. Thus, this method can only be used in the case of small matrices. For large matrices we must use multi-level reduction.

We tested the multi-level reduction varying the number of workers from 16 to two, by sequences of power of two.

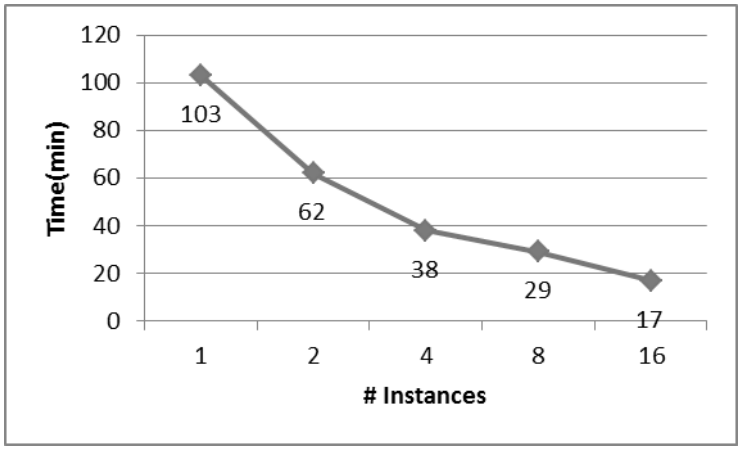


Figure 14. Multi-level reduction processing time chart

For big matrices, large numbers like 16 or eight workers reduce time more efficiently, but this is not true for small matrices because there are overheads related to the mapping and reducing stages of the proposed process. The results show that the split process is very fast, because it is a process that is constrained to a specific region in the map. Thus, a worker loads only the files it needs to process and then creates the next integration file to bind the sparse matrices.

6. CONCLUSION

This paper describes a framework for general 2D and 3D cellular automata simulation with very large matrices (10^6 x 10^6), which is based on cloud computing technology. Such massive simulations are possible because the proposed system uses an optimized sparse matrix compression method for subdivision and reconstruction that allows the server to reduce the transfer size over internal network and Internet.

The proposed method makes it possible for studies to use an on-demand application that simulates massive matrices very fast, due to highly parallel subdivision and reduction. The major problem is the time to transfer large files to the cloud, a direct consequence of the quality and speed of current commercial Internet transfer rates. In some instances, it is possible to use a procedural matrix builder to reduce the transfer, but this is not always the case.

The proposed framework can be easily used by non-computer experts, because they only have to provide simulation parameters and the transition function in the form of a text file containing a program written in a script language. The framework removes the burden related to the subdivision, mapping, and parallelization processes, so that researchers need only be concerned with their simulation algorithms.

In future work, we intend to integrate the proposed solution into local clusters using Microsoft HPC Pack SDK, which transforms a local cluster in an integrated private cloud system. Moreover, the scheduler can be improved to automatically increase the number of instances to accomplish deadlines. In this way, it could operate with deadlines instead of number of instances.

Another improvement is the use of GPU Roles that can potentially solve CA much faster, but, currently, there are very few services in the market that support them, and GPU servers remain very expensive.

We also plan to test 3D cases more extensively. Currently we are modeling the cellular automaton used by Kim *et al.* [37], for brain tumor segmentation, which is a complex 3D CA example. Finally, we foresee the framework being used to create a series of CA problem solver services, allowing research groups to integrate simulators and create solutions for huge problems.

The service is available at http://casim.azurewebsites.net, and the SDK can be found at http:// www.icad.puc-rio.br/casim/, together with some examples.

ACKNOWLEDGMENTS

This work was partially supported by CNPq (National Council for Scientific and Technological Development, linked to the Ministry of Science and Technology), CAPES (Coordination for the Improvement of Higher Education

Personnel, linked to the Ministry of Education), FINEP (Brazilian Innovation Agency), and ICAD/VisionLab (PUC-Rio).

REFERENCES

[1] Mendes , R. L., Santos, A. A.; Martins, M. L.; Vilela, M. J. Cluster size distribution of cell aggregates in culture, Physica A, vol. 298, 2001. doi: 10.1016/S0378-4371(01)00238-2

[2] White, S. H.; D. Rey, A. M.; Sanchez, G. R. Modeling epidemics using cellular automata, Applied Mathematics and Computation, vol. 186(1), pp. 193-202. 2007. doi: 10.1016/j.amc.2006.06.126

[3] Zhao, Y.; Billings, S.A.; Coca, D. Cellular automata modelling of dendritic crystal growth based on Moore and von Neumann neighbourhoods, Int. J. Modelling, Identification and Control, vol 6, 119-125. 2009. ISSN: 1746-6172, doi: 10.1504/IJMIC.2009.024328

[4] Gomes, T.; Silva, R. M. A. & Ferracioli, L. 2006. A Model of Velocity Distribution of a River Based on a Qualitative Computer Modelling Environment. In: Proceedings of The International Workshop on Applied Modelling & Simulation, Buzios, RJ, 133-137.

[5] Isliker, H., Anastasiadis, A., Vassiliadis, D. and Vlahos, L. Solar flare cellular automata interpreted as discretized MHD equations, Astronomy And Astrophysics, Vol. 335, Issue 3, pp. 1085-1092, July 1998.

[6] Qiu, G.; Kandhai, B.D.; Sloot, P.M.A. Understanding the complex dynamics of stock markets through cellular automata, Phys Rev E., vol. 75. 2007. doi: 10.1103/PhysRevE.75.046116

[7] Itami, R. Simulating spatial dynamics: cellular automata theory. Landscape and Urban Planning, 30, pp. 27-47. 1994.

[8] Rajkumar, B., Chee, S.Y., Srikumar, V.: Market oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. In: The 10th IEEE International Conference on High Performance Computing and Communications, Dalian, China, September 25-27 (2008). doi: 10.1016/0169-2046(94)90065-5

[9] Wolfram S. Universality and complexity in cellular automata. *Physica D* 1984; 10(1-2): 1–35. doi:10.1016/0167-2789(84)90245-8

[10] Wolfram, S.; Packard, N. H. Two-dimensional cellular automata. Journal of Statistical Physics, vol. 38, pp. 901–946. 1985.

[11] Sipper, M. and Ruppin, E. Co-evolving cellular architectures by cellular programming. In Proceedings of IEEE Third International Conference on Evolutionary Computation (ICEC'96), pp 306-311, May 1996. ISBN: 0-7803-2902-3, doi: 10.1109/ICEC.1996.542380

[12] Pighizzini G. Asynchronous automata versus asynchronous cellular automata. *Theoretical Computer Science* 1994; 132(1-2): 179–207. doi: 10.1016/0304-3975(94)90232-1

[13] Sarkar P, Barua R. Multidimensional σ–automata, π-polynomials and generalised S-matrices. *Theoretical Computer Science* 1998; 197 (1-2): 111-138. doi: 10.1016/S0304-3975(97)00160-6

[14] Armbrust M, Fox A, *et al*. Above the clouds: a Berkeley view of cloud computing. Technical Report no. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[15] Danielson, Krissi (2008-03-26). "Distinguishing Cloud Computing from Utility Computing". Ebizq.net. Retrieved 2010-08-22.

[16] Miller, Michael. Cloud Computing – Web-Based Applications That Change The Way You Work And Collaborate Online, Que Publishing, 2009

[17] Gruman, Galen (2008-04-07). "What cloud computing really means". InfoWorld. Retrieved 2009-06-02;

[18] "Gartner Say's Cloud Computing Will Be As Influential As E-business". Gartner.com. Retrieved 2010-08-22;

[19] Carr, N., Here comes HaaS, 2006 – http://www.roughtype.com/archives/2006/03/here_comes_haas.php

[20] Vogels, W., A Head in the Clouds – The Power of Infrastructure as a Service, In: First workshop on Cloud Computing in Applications (CCA'08), October, 2008.

[21] Kulkarnil, G., Sutar, R. and Gambhir, J. Cloud computing-infrastructure as service Amazon EC2. International Journal of Engineering Research and Applications (IJERA). Vol. 2, Issue 1, Jan-Feb 2012, pp.117-125, ISSN: 2248-9622.

[22] Garfinkel, Simson L. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical Report TR-08-07. Center for Research on Computation and Society, School for Engineering and Applied Sciences, Harvard University, Cambridge. August 2007.

[23] Malawski, M.; Kuźniar, M.; Wójcik, P.; Bubak, M., "How to Use Google App Engine for Free Computing," Internet Computing, IEEE , vol.17, no.1, pp.50,59, Jan.-Feb. 2013 doi: 10.1109/MIC.2011.143

[24] Cheng, D. PaaS-onomics: A CIO's Guide to using Platform-as-a-Service to Lower Costs of Application Initiatives While Improving the Business Value of IT. Tech. rep., LongJump, 2008. (also available at https://na.longjump.com/networking/RepositoryPublicDocDownload?id=892085391bcc87675092&docname=PaaSonomicsWhitepaper2.pdf&cid=892085391&encode=application/pdf).

[25] Timane, Rajesh, Analysis of Cloud Computing Market Players. International Journal of Research in IT & Management. Volume 1, Issue 5. September 2011. ISSN 2231-4334. Available at SSRN: http://ssrn.com/abstract=2188357

[26] Miller, Wes. What Windows Azure Means to Software ISVs Microsoft Windows Azure. In Directions on Microsoft Research Series. August 2012. (also available at http://www.microsoft.com/en-us/news/itanalyst/docs/08-2012DirectionsAzureISV.pdf) ISSN: 1077-4394

[27] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (Jan. 2008), 107-113. doi: 10.1145/1327452.1327492

[28] Duff I. S. A survey of sparse matrix research. In: Proc. of the IEEE, 65(4), pp. 500-535. April 1977. doi:10.1109/PROC.1977.10514

[29] Barret R, Berry M, *et al*. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Ed., SIAM, 1995. (also available at www.netlib.org/templates/templates.pdf). ISBN: 0-89871-328-5, doi: 10.1.1.109.2076

[30] Knuth, D.E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, 1968. ISBN-10: 0201485419
[31] Intel, Sparse Matrix Storage Formats. In: Math Kernel Library Reference Manual, available at http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm [accessed on March 18, 2012].
[32] Davis, T.A. and Hu, Y. The university of Florida sparse matrix collection, ACM Transactions on Mathematical Software (TOMS), Artcle No. 1, Vol. 38, Issue 1, November 2011. doi: 10.1.1.94.1925
[33] Randolph E. Bank, Craig C. Douglas. Sparse Matrix Multiplication Package (SMMP), Advances in Computational Mathematics - Adv. Comput. Math. , vol. 1, no. 1, pp. 127-137, 1993, doi: 10.1007/BF02070824
[34] Turner, S., Chen, L. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. RFC 6151. Internet Engineering Task Force (IETF). March 2011. ISSN: 2070-1721.
[35] Kuan-Ching Li; Hsun-Chang Chang; Chao-Tung Yang; Li-Jen Chang; Hsiang-Yao Cheng; Liang-Teh Lee, "Implementation of visual MPI parallel program performance analysis tool for cluster environments," Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on , vol.2, no., pp.215,218 vol.2, 28-30 March 2005 doi: 10.1109/AINA.2005.210
[36] Kozminski, A., "Windows Presentation Foundation (WPF) technology meets the challenges of operator interface design in automatic test systems," AUTOTESTCON, 2012 IEEE , vol., no., pp.80,83, 10-13 Sept. 2012 doi: 10.1109/AUTEST.2012.6334585
[37] Kim E, Shen T, Huang X. A Parallel Cellular Automata with Label Priors for Interactive Brain Tumor Segmentation, IEEE 23rd International Symposium on Computer-Based Medical Systems (CBMS), 232-237, 2010. ISBN: 978-1-4244-9167-4, doi: 10.1109/CBMS.2010.6042647