Some Applications of Spiking Neural P Systems

Mihai Ionescu[1], Dragoş Sburlan[2]

[1] Research Group on Mathematical Linguistics
Universitat Rovira i Virgili
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
armandmihai.ionescu@urv.cat
[2] Ovidius University
Faculty of Mathematics and Informatics
Constantza, Romania
dsburlan@univ-ovidius.ro

Summary. In this paper we investigate some applications of spiking neural P systems regarding their capability to solve some classical computer science problems. In this respect it is studied the versatility of such systems to simulate a well known parallel computational model, namely the Boolean circuits. In addition, another notorious application - the sorting - is considered within this framework.

1 Introduction

Spiking neural P systems (shortly called SN P systems) are a class of computing models introduced in [9]. They are using ideas from neural computing, area currently under high investigation, with a focus on spiking neurons (see, e.g., [4], [12], [13]).

The new models are based on the tissue-like and neural-like P systems structure to which various features were added. Details can be found at the website of membrane computing ([21]). For an introduction in the area we refer to [16].

In short, an SN P system consists of a set of neurons placed in the nodes of a graph and sending signals (spikes) along synapses (edges of the graph), under the control of firing rules. One also uses forgetting rules, which remove spikes from neurons. Hence, the spikes are moved and created, destroyed, but never modified (there is only one type of objects in the system).

A generalization of the original model was considered in [15], [3] where rules of the form $E/a^c \to a^p; d$ where introduced. The meaning is that when using the rule, c spikes are consumed and p spikes are produced. Because p can be 0 or greater than 0, we obtain at the same time a generalization of both spiking and forgetting rules. Different from the original model of SN P systems, in [10], parallelism inside a neuron was introduced. By that we mean that when a rule $E/a^c \to a; d$ can be

applied (the contents of a neuron is described by the regular expression E), then we apply it as many times as possible in that neuron.

Based on the above features, we investigate their power to simulate Boolean gates and circuits. We also introduce here a modality to sort natural numbers (given as number of spikes) with SN P systems in the initial version.

2 Prerequisites

In this section we first introduce the definition of SN P system which we will use during our endeavor, altogether with some explanations on the exhaustive use of the rules. Then, we recall (some) basic notions on Boolean functions and circuits.

2.1 SN P systems

A *spiking neural P system* (in short, an SN P system), of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ are *neurons*, of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained by the neuron;
   b) R_i is a finite set of *rules* of the following two forms:
      (1) $E/a^c \to a; d$, where E is a regular expression over O, $c \geq 1$, and $d \geq 0$;
      (2) $a^s \to \lambda$, for some $s \geq 1$, with the restriction that $a^s \in L(E)$ for no rule $E/a^c \to a; d$ of type (1) from R_i;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$, for $1 \leq i \leq m$ (*synapses*);
4. $out \in \{1, 2, \ldots, m\}$ indicates the *output neuron*.

The rules of type (1) are *firing* (also called *spiking*) *rules*, and the rules of type (2) are called *forgetting rules*. The first ones are applied as follows: if the neuron contains k spikes, $a^k \in L(E)$ and $k \geq c$, then the rule $E/a^c \to a; d$ can be applied, and this means that c spikes are consumed, only $k - c$ remain in the neuron, the neuron is fired, and it produces one spike after d time units (a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized). If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, and so on. In the case $d \geq 1$, if the rule is used in step t, then in steps $t, t + 1, t + 2, \ldots, t + d - 1$ the neuron is *closed*, and it cannot receive new spikes (if a neuron has a synapse to a closed neuron and sends a spike along it, then the spike is lost). In step $t + d$, the neuron spikes and becomes again open, hence can receive spikes (which can be used in step $t+d+1$). A spike emitted by a neuron σ_i is replicated and goes to all neurons σ_j such that $(i, j) \in syn$.

The forgetting rules, are applied as follows: if the neuron contains exactly s spikes, then the rule $a^s \to \lambda$ can be used, and this means that all s spikes are removed from the neuron.

In each time unit, in each neuron which can use a rule we have to use a rule, either a firing or a forgetting one. Because two firing rules $E_1/a^{c_1} \to a; d_1$ and $E_2/a^{c_2} \to a; d_2$ can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron, and then one of them is chosen non-deterministically. Note however that we cannot interchange a firing rule with a forgetting rule, as all pairs of rules $E/a^c \to a; d$ and $a^s \to \lambda$ have disjoint domains, in the sense that $a^s \notin L(E)$.

The initial configuration of the system is described by the numbers $n_1, n_2, \ldots, n_m$ of spikes present in each neuron. Starting from the initial configuration and applying the rules, we can define transitions among configurations. A transition between two configurations C_1, C_2 is denoted by $C_1 \Longrightarrow C_2$. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where all neurons are open and no rule can be used.

With any computation, halting or not, we associate a *spike train*, a sequence of digits 0 and 1, with 1 appearing in positions $1 \leq t_1 < t_2 < \ldots$, indicating the steps when the output neuron sends a spike out of the system (we also say that the system itself spikes at that time). With any spike train containing at least two spikes we associate a result, in the form of the number $t_2 - t_1$; we say that this number is computed by Π. By definition, if the spike train contains only one occurrence of 1, then we say that we have computed the number zero. The set of all numbers computed in this way by Π is denoted by $N_2(\Pi)$ (the subscript indicates that we only consider the distance between the first two spikes of any computation). Then, by $Spik_2P_m(rule_k, cons_q, forg_r)$ we denote the family of all sets $N_2(\Pi)$ computed as above by spiking neural P systems with at most $m \geq 1$ neurons, using at most $k \geq 1$ rules in each neuron, with all spiking rules $E/a^c \to a; t$ having $c \leq q$, and all forgetting rules $a^s \to \lambda$ having $s \leq r$. When one of the parameters m, k, q, r is not bounded, it is replaced with $*$.

A rule of the type $E/a^c \to a^p$ is called an *extended rule*, and is applied as follows: if neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule can fire, and its application means consuming (removing) c spikes (thus only $k - c$ remain in σ_i) and producing p spikes, which will exit immediately the neuron.

In this paper, we use SN P systems of the form introduced above, but using the rules in the exhaustive way. Namely if a rule $E/a^c \to a^p; d$ is associated with a neuron σ_i which contains k spikes, then the rule is enabled (we also say *fired*) if and only if $a^k \in L(E)$. Using the rule means the following. Assume that $k = sc+r$, for some $s \geq 1$ (this means that we must have $k \geq c$) and $0 \leq r < c$ (the remainder of dividing k by c). Then sc spikes are consumed, r spikes remain in the neuron σ_i, and sp spikes are produced and sent to the neurons σ_j such that $(i, j) \in syn$ (as usual, this means that the sp spikes are replicated and exactly sp spikes are sent to each of the neurons σ_j). In the case of the output neuron, sp spikes are

also sent to the environment. Of course, if neuron σ_i has no synapse leaving from it, then the produced spikes are lost.

We stress two important features of this model. First, it is important to note that only one rule is chosen and applied, the remaining spikes cannot evolve by another rule. For instance, even if a rule $a(aa)^*/a \to a; 0$ exists, it cannot be used for the spike remaining unused after applying the rule $a(aa)^*/a^2 \to a; 0$. Second, is that the covering of the neuron is checked only for enabling the rule, not step by step during its application. For instance, the rule $a^5/a^2 \to a; 0$ has the same effect as $a(aa)^*/a^2 \to a; 0$ in the case of a neuron containing exactly 5 spikes: the rule is enabled, 4 spikes are consumed, 2 are produced; both applications of the rule are concomitant, not one after the other, hence all of them have the same enabling circumstances.

If several rules of a neuron are enabled at the same time, one of them is nondeterministically chosen and applied. The computations proceed as in the SN P systems with usual rules, and a spike train is associated with each computation by writing 0 for a step when no spike exits the system and 1 within a step when one or more spikes exit the system. Then, a number is associated – and said to be generated/computed by the respective computation – with a spike train containing at least two occurrences of the digit 1, in the form of the steps elapsed between the first two occurrences of 1 in the spike train. Number 0 is computed by computations whose spike trains contain only one occurrence of 1.

2.2 Boolean Functions and Circuits

An n-ary *Boolean function* is a function $f\{true, false\}^n \mapsto \{true, false\}$. $\neg$ (negation) is a unary Boolean function (the other unary functions are: constant functions and identity function). We say that Boolean expression φ with variables $x_1, \ldots, x_n$ *expresses* the n-ary Boolean function f if, for any n-tuple of truth values $t = (t_1, \cdots, t_n)$, $f(t)$ is *true* if $T \vDash \varphi$, and $f(t)$ is false if $T \nvDash \varphi$, where $T(x) = t_i$ for $i = 1, \ldots, n$.

There are three primary Boolean functions that are widely used: The NOT function - this is a just a negation; the output is the opposite of the input. The NOT function takes only one input, so it is called a unary function or operator. The output is true when the input is false, and vice-versa. The AND function - AND function returns true only if all inputs are true; if there is an input which is false the function returns false. The OR function - the output of an OR function is true if the first input is true or the second input is true or the third input is true, etc. (hence, to return true is enough for one input to be true). Both AND and OR can have any number of inputs, with a minimum of two.

Any n-ary Boolean function f can be expressed as a Boolean expression φ_f involving variables $x_1, \ldots, x_n$.

There is a potentially more economical way than expressions for representing Boolean functions, namely *Boolean circuits*. A Boolean circuit is a graph $C = (V, E)$, where the nodes in $V = \{1, \ldots, n\}$ are called the *gates* of C. Graph C has

a rather special structure. First, there are no cycles in the graph, so we can assume that all edges are of the form (i, j), where $i < j$. All nodes in the graph have the "in-degree" (number of incoming edges) equal to 0, 1, or 2. Also, each gate $i \in V$ has a *sort* $s(i)$ associated with it, where $s(i) \in \{true, false, \vee, \wedge, \neg\} \cup \{x_1, x_2, \dots\}$. If $s(i) \in \{true, false\} \cup \{x_1, x_2, \dots\}$, then the in-degree of i is 0, that is, i must have no incoming edges. Gates with no incoming edges are called the *inputs* of C. If $s(i) = \neg$, then i has "in-degree" one. If $s(i) \in \{\vee, \wedge\}$, then the in-degree of i must be two. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges) is called the *output gate* of the circuit.

This concludes our definition of the *syntax* of circuits. The *semantics* of circuits specifies a truth value for each appropriate truth assignment. We let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X \mid s(i) = x$ for some gate i of $C\}$). We say that a truth assignment T is appropriate for C if it is defined for all variables in $X(C)$. Given such a T, *the truth value of gate* $i \in V$, $T(i)$, is defined, by induction on i, as follows: If $s(i) = true$ then $T(i) = true$, and similarly if $s(i) = false$. If $s(i) \in X$, then $T(i) = T(s(i))$. If now $s(i) = \neg$, there is a unique gate $j < i$ such that $(j, i) \in E$. By induction, we know $T(j)$, and then $T(i)$ is *true* if $T(j) = false$, and vice-versa. If $s(i) = \vee$, then there are two edges (j, i) and (j', i) entering i. $T(i)$ is then *true* if only if at least one of $T(j)$, $T(j')$ is *true*. If $s(i) = \wedge$, then $T(i)$ is *true* if only if both $T(j)$ and $T(j')$ are *true*, where (j, i) and (j', i) are the incoming edges. Finally, *the value of the circuit*, $T(C)$, is $T(n)$, where n is the output gate.

3 Simulating Logical Gates and Circuits

In this section we show how SNP systems can simulate logical gates. We consider that input is given in one neuron while the output will be collected from the output neuron of the system. Boolean value 1 is encoded in the spiking system by two spikes, hence a^2, while 0 is encoded as one spike.

We collect the result as follows. If the output neuron fires two spikes in the second step of the computation, then the Boolean value computed by the system is 1 (hence true). If it fires only one spike, then the result is 0 (false).

3.1 Simulating Logical Gates

Lemma 1. *Boolean AND gate can be simulated by SN P systems using two neurons and no delay on the rules, in two steps.*

Proof. We construct the SNP system (formed by only one neuron):

$$\Pi_{AND} = (\{a\}, \sigma_1 = (0, \{a^2 \to a; 0, a^3 \to a; 0, a^4/a^2 \to a; 0\}), \emptyset, 1).$$

The functioning of the system is rather simple (remember that the rules are used in an exhaustive way). Suppose in neuron 1 we introduce three spikes. This

means we compute the logical AND between 1 and 0 (or 0 and 1). The only rule the system can use is $a^3 \to a; 0$ and one spike (hence the correct result - 0 in this case) is sent to the environment.

If 4 spikes are introduced in neuron 1 (the case 11), the output neuron will fire using the rule $a^4/a^2 \to a; 0$, and will send two spikes in the environment. The system with the input 00 behaves similarly to the 01 or 10 cases. We have shown how the system we have constructed gives the right answer in one computational step and gets back to its initial configuration for a further use, if necessary.

We want to emphasize here that no "extended" rule was used. Of course, a rule $a^4 \to a^2$ can substitute, with the same effect, the rule we have preferred above (namely $a^4/a^2 \to a; 0$) but, in simulating Boolean gates, we have tried to minimize the use of such rules. An extended rule is used only once in simulating Boolean gates, more precisely in the simulation of OR gate.

If in the system above, in the output neuron, we change only the rule $a^3 \to a; 0$ (with the rule $a^3 \to a^2; 0$) we obtain the OR gate.

Lemma 2. *Boolean OR gate can be simulated by SN P systems using two neurons and no delay on the rules, in two steps.*

We now pass to the simulation of logical gate NOT.

Lemma 3. *Boolean NOT gate can be simulated by SNP systems using two neurons, no delay on the rules, in two steps.*

Proof. We first want to stress that in simulating this gate we did not use any extended rules. The case when such rules are used is left to the reader.

Let us construct the following SN P system:

$$\Pi_{NOT} = (\{a\}, \sigma_1, \sigma_2, \{(1,2),(2,1)\}, 1),$$

and:

- $\sigma_1 = (a, \{a^2/a \to a; 0, a^3 \to a; 0\})$,
- $\sigma_2 = (0, \{a/a \to a; 0, a^2/a^2 \to a; 0\})$.

Let us emphasize that in order to simulate Boolean gate NOT, in the initial configuration, neuron 1 contains 1 spike, which, once used to correctly simulate the gate, has to be present again in the neuron such that the system returns to its initial configuration. This is done with the help of neuron 2 which in step 2 of the computation refills neuron 1 with one spike.

The system is given in its initial configuration in Figure 1. This gives us the opportunity to introduce the way we graphically represent a SN P system: as a directed graph, with the neurons as nodes and the synapses indicated by arrows. Each neuron has inside its specific rules and the spikes present in the initial configuration.

If the input in the Boolean gate is 1, then two spikes are placed in neuron 1. Having three spikes inside (two from the input, and one initially present inside)

neuron 1 can use only rule $a^3 \to a; 0$, thus sending one spike to the environment (hence Boolean 0 – the correct result – is obtained), and one spike to neuron 2. The latter one will send the spike back, in the second step of the computation by using rule $a/a \to a; 0$, and the system regains its initial configuration.

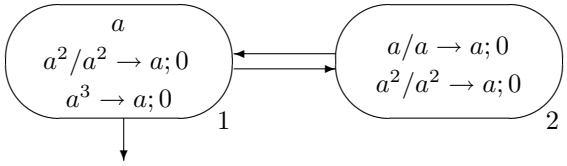

Figure 1. SN P systems simulating **NOT** gate

If the input in the Boolean gate is 0, hence one spike is introduced in neuron 1, it uses the rule $a^2/a \to a; 0$, two spikes are sent to the environment (and the result of the computation is 1), and to neuron 2 in the same time. In the second step of the computation neuron 2 uses the rule $a^2/a^2 \to a; 0$, consumes the two spikes present inside, and sends one back to neuron 1. The system recovers its initial configuration.

After showing how SN P systems can simulate logical gates, we pass to the simulation of circuits.

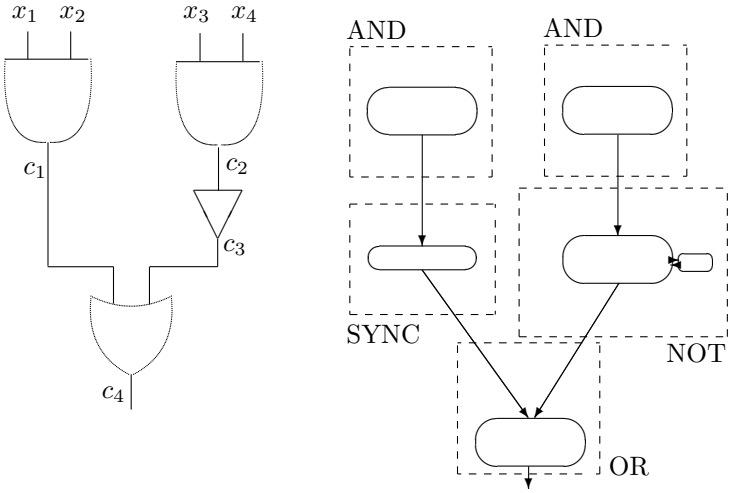

Figure 2. Boolean Circuit and the Spiking System

3.2 Simulating Circuits

Next, we are presenting an example of how to construct a SN P system to simulate a Boolean circuit designed to evaluate a Boolean function. Of course, in our goal

we are using the systems Π_{AND}, Π_{OR}, and Π_{NOT} constructed before, to which we add extra neurons to synchronize the system for a correct output.

We start with the same **example** considered in [1] and [11], namely the function $f : \{0,1\}^4 \to \{0,1\}$ given by the formula

$$f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4).$$

The circuit corresponding to the above formula as well as the spiking system assign to it are depicted in Figure 2.

In order for the system that simulates the circuit to output the correct result it is necessary for each sub-system (that simulates the gates AND, OR, and NOT) to receive the input from the above gate(s) at the same time. To this aim, we have to add synchronization neurons, initially empty with a single rule inside $(a \to a; 0)$. Note that in Figure 2. we have added such a neuron in order for the output of the first AND gate to enter gate OR at the same time with the output of NOT gate (at the end of the second step of the computation).

Having the overall image of the functioning of the system, let us give some more details on the simulation of the above formula. For that we construct the SN P system

$$\Pi_C = (\Pi^{(1)}_{AND}, \Pi^{(2)}_{AND}, \Pi^{(3)}_{NOT}, \Pi^{(4)}_{OR})$$

formed by the sub-SN P systems for each gate, and we obtain the unique result as follows:

1. for every gate of the circuit with inputs from the input gates we have a SN P system to simulate it. The input is given in neuron labeled 1 of each gate;
2. for each gate which has at least one input coming as an output of a previous gate we construct a SN P system to simulate it by "constructing" a synapse between the output neuron of the gate from which the signal (spike) comes and the input neuron of the system that simulates the new gate.
   Note that if synchronization is needed the new synapse is constructed from the output neuron of the output gate to the synchronization neuron and from here another synapse is constructed to the input of the new gate in the circuit.

For the above formula and the circuit depicted in Figure 2 we will have:

- $\Pi^{(1)}_{AND}$ computes the first AND_1 gate $(x_1 \wedge x_2)$ with inputs x_1 and x_2.
- $\Pi^{(2)}_{AND}$ computes the second AND_2 gate $(x_3 \wedge x_4)$ with inputs x_3 and x_4; these two P systems, $\Pi^{(1)}_{AND}$ and $\Pi^{(2)}_{AND}$, act in parallel.
- $\Pi^{(3)}_{NOT}$ computes NOT gate $\neg(x_3 \wedge x_4)$ with input $(x_3 \wedge x_4)$. While $\Pi^{(3)}_{NOT}$ is working, the output value of the first AND_1 gate passes through the synchronization neuron.
- The input enters in the first neuron of OR gate, and SN P system $\Pi^{(4)}_{OR}$ completes its task. The result of the computation for OR gate (which is the result of the global P system), is sent into the environment of the whole system.

Generalizing the previous observations the following result holds:

Theorem 1. *Every Boolean circuit α, whose underlying graph structure is a rooted tree, can be simulated by a SN P system, Π_α, in linear time. Π_α is constructed from SN P systems of type Π_{AND}, Π_{OR} and Π_{NOT}, by reproducing in the architecture of the neural structure, the structure of the tree associated to the circuit.*

4 A Sorting Algorithm

We pass now to a different problem SN P systems can solve, namely to sort n natural numbers, this time not using the rules in the exhaustive way, but as in the original definition of such systems.

We first exemplify our sorting procedure through an example. Let us presume we want to sort the natural numbers 1, 3, and 2, given in this order. For that we construct the following system given only in its pictorial format below:

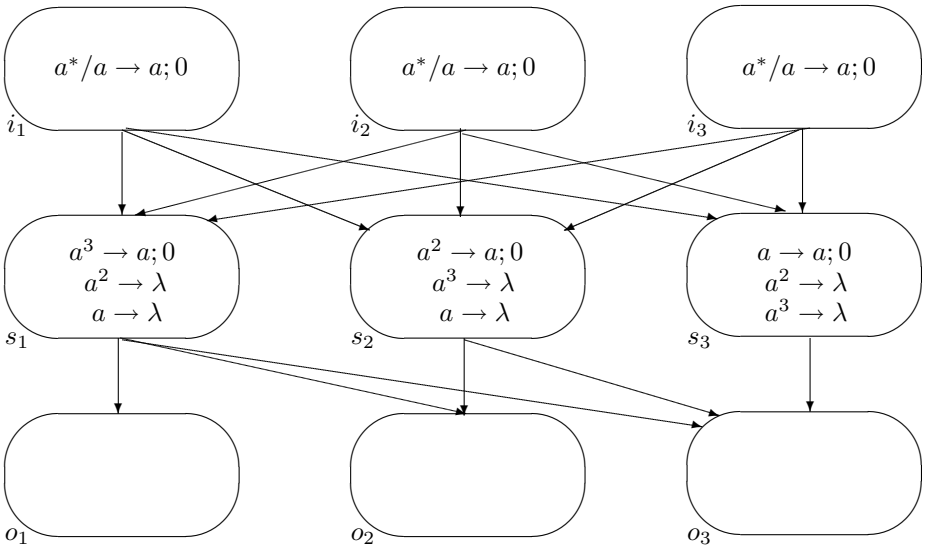

Figure 3. Sorting three natural numbers

We encode natural numbers in the number of spikes (1 – one spike, 3 – three spikes, 2 – two spikes) which we input in the first line of the system (hence in the neurons labeled i_1, i_2, an i_3). It can be noticed that the neurons in the first layer of the structure are having the same rule inside ($a^*/a \to a; 0$) and outgoing synapses to all the neurons in the second layer of the structure (the ones denoted s_1, s_2, and s_3). Neuron labeled s_1 has outgoing synapses with all neurons in the third layer of the system, only one spiking rule inside ($a^3 \to a; 0$, where 3 is the number of numbers that have to be sorted), and two deletion rules ($a^2 \to \lambda$, and $a \to \lambda$). For the other neurons in the second layer, the exponent of the firing rule

decreases one by one as well as the synapses with the neurons from the third layer of the system.

In the initial configuration of the system we have one spike in neuron i_1, three spikes in neuron i_2 and 2 spikes in neuron i_3. In the *first step* of the computation, one spike from each neuron is consumed and sent to neurons from the second layer of the system. Each of them receives the same number of spikes, namely 3.

In the *second step* of the computation, neuron labeled s_1 consumes all three spikes previously received and fires to neurons o_1, o_2 and o_3. Hence, each neuron from the output layer has one spike inside. The other neurons from the second layer delete the three spikes they have received. In the same time neurons i_2 and i_3 fire again sending 2 spikes (one each) to all neurons from the second layer.

In the *third step* of the computation, neuron s_2 fires only to neurons o_2 and o_3 (so, they will have one more spike inside, hence 2, while o_1 remains with only one spike), the other spikes from neurons s_1 and s_3 being deleted. In the same time neuron i_2 refills the neurons in the second layer of the system with one spike, which will be consumed in the *forth step* of the computation by neuron s_3 and sent to the output neuron o_3.

So, in the last step of the computation there are: 1 spike in the neuron o_1, 2 spikes in the neuron o_2, and 3 spikes in the neuron o_3.

We pass now to the general case, constructing the system in the pictorial form:

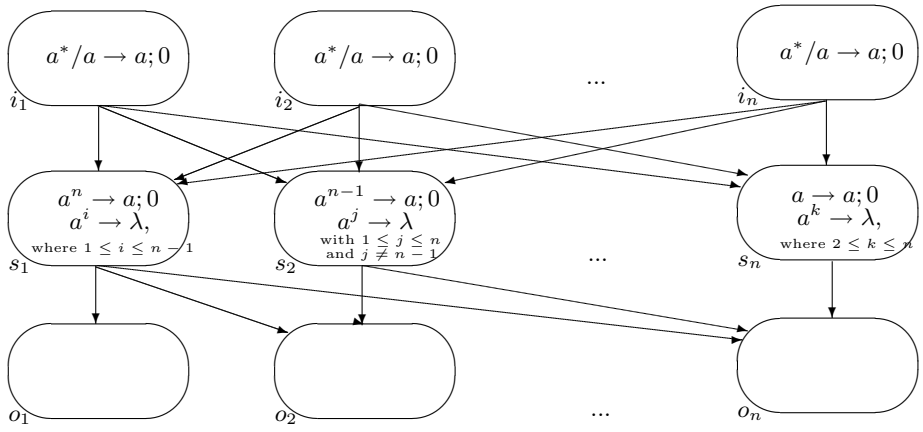

Figure 4. Sorting n natural numbers

The functioning of the system is similar to the one described in the example above. We introduce n natural numbers encoded as spikes, one in each neuron from the first layer of the structure (denoted by i_j, with $1 \leq j \leq n$). As long as they are not empty they consume at each step a spike, and send n spikes, one to each neuron from the second layer of the structure (denoted by s_i, with $1 \leq i \leq n$). The latter neurons have n different thresholds (decreasing one by one from n – neuron labeled s_1, to 1 – neuron labeled s_n), and have n different number of synapses

with the neurons from the third layer of the structure. The latter ones contain the result of the computation.

Theorem 2. *SN P systems can sort a vector of natural numbers where each number is given as number of spikes introduced in the neural structure.*

Based on the above construction, the time complexity (measured as usually as the number of configurations reached during the computation) is $O(T)$, where T is the magnitude of the numbers to be sorted. Although the time complexity is better than the "classical", sequential algorithm, in this case one can notice that the construction presented depends on the number of numbers to be sorted.

5 Final Remarks

Spiking neural P systems are a versatile formal model of computation that can be used for designing efficient parallel algorithms for solving known computer science problems. Here we firstly studied the ability of SN P systems to simulate Boolean circuits since, apart for being a well known computational model, there exists many "fast" algorithms solving various problems. In addition, this simulation, enriched with some "memory modules" (given in the form of some SN P sub-systems), may constitute an alternative proof of the computational completeness of the model.

Another issue studied here regards the sorting of a vector of natural numbers using SN P systems. In this case, due to its parallel features, the obtained time complexity for the proposed algorithm overcome the classical sequential ones.

Several open problems arose during our research. For instance, in case of Boolean circuits the simulation is done for such circuits whose underlying graphs have rooted tree structures, therefore a constraint that need further investigations.

In what regards the sorting algorithm, the presented construction depends on the magnitude of the numbers to be sorted. We conjecture that this inconvenient might be eliminated. Also, we conjecture that further improvements concerning time complexity can be made.

Acknowledgements

The work of the authors was supported as follows. M. Ionescu: fellowship "Formación de Profesorado Universitario" from the Spanish Ministry of Education, Culture and Sport, and Dragoş Sburlan: CEEX grant (2-CEx06-11-97/19.09.06), Romanian Ministry of Education and Research.

References

1. R. Ceterchi, D. Sburlan: Simulating Boolean Circuits with P Systems, *LNCS*, 2933, 104–122, 2004.

2. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On String Languages Generated by Spiking Neural P Systems. In [5], Vol. I, 169–194.
3. H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Spiking Neural P Systems with Extended Rules. In [5], Vol. I, 241–265.
4. W. Gerstner, W Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge Univ. Press, 2002.
5. M.A. Gutiérrez-Naranjo et al., eds.: *Proceedings of Fourth Brainstorming Week on Membrane Computing*, Febr. 2006, Fenix Editora, Sevilla, 2006.
6. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosik, S. Woodworth: Normal Forms for Spiking Neural P Systems. In [5], Vol. II, 105–136, and *Theoretical Computer Sci.*, to appear.
7. O.H. Ibarra, S. Woodworth: Characterizations of Some Restricted Spiking Neural P Systems. In *Pre-proceedings of Seventh Workshop on Membrane Computing, WMC7*, Leiden, The Netherlands, July 2006, 387–396.
8. O.H. Ibarra, S. Woodworth, F. Yu, A. Păun: On Spiking Neural P Systems and Partially Blind Counter Machines. In *Proceedings of Fifth Unconventional Computation Conference, UC2006*, York, UK, September 2006, 123–135.
9. M. Ionescu, Gh. Păun, T. Yokomori: Spiking Neural P Systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
10. M. Ionescu, Gh. Păun, T. Yokomori: Spiking Neural P Systems with an Exhaustive Use of Rules, *International Journal of Unconventional Computing*, accepted.
11. M. Ionescu, T.-O. Ishdorj: Boolean Circuits and a DNA Algorithm in Membrane Computing. *LNCS*, 3850, 272–291.
12. W. Maass: Computing with Spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
13. W. Maass, C. Bishop, eds.: *Pulsed Neural Networks*, MIT Press, Cambridge, 1999.
14. M. Minsky: *Computation – Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.
15. A. Păun, Gh. Păun: Small Universal Spiking Neural P Systems. In [5], Vol. II, 213–234, and *BioSystems*, in press.
16. Gh. Păun: *Membrane Computing – An Introduction*. Springer, Berlin, 2002.
17. Gh. Păun: Languages in Membrane Computing. Some Details for Spiking Neural P systems. *LNCS* 4036, 2006, 20–35.
18. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Spike Trains in Spiking Neural P Systems. *Intern. J. Found. Computer Sci.*, 17, 4 (2006), 975–1002.
19. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite Spike Trains in Spiking Neural P Systems. Submitted 2005.
20. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*, 3 volumes. Springer-Verlag, Berlin, 1997.
21. The P Systems Web Page: `http://psystems.disco.unimib.it`.