Article

Networks of Uniform Splicing Processors: Computational Power and Simulation

Sandra Gómez-Canaval ¹ , Victor Mitrana ^{1,2,*} , Mihaela Păun ³ ,
José Angel Sanchez Martín ¹ and José Ramón Sánchez Couso ¹

¹ Departamento de Sistemas Informáticos, Universidad Politécnica de Madrid, C/Alan Turing s/n, 28031 Madrid, Spain; sm.gomez@upm.es (S.G.-C.); joseangel.sanchez.martin@alumnos.upm.es (J.A.S.M.); joseramon.sanchez.couso@upm.es (J.R.S.C.)

² Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, 010014 Bucharest, Romania

³ National Institute for Research and Development of Biological Sciences, Independentei Bd. 296, 060031 Bucharest, Romania; mihaela.paun@gmail.com

* Correspondence: mitrana@fmi.unibuc.ro

Received: 30 June 2020; Accepted: 23 July 2020; Published: 24 July 2020

Abstract: We investigated the computational power of a new variant of network of splicing processors, which simplifies the general model such that filters remain associated with nodes but the input and output filters of every node coincide. This variant, called *network of uniform splicing processors*, might be implemented more easily. Although the communication in the new variant seems less powerful, the new variant is sufficiently powerful to be computationally complete. Thus, nondeterministic Turing machines were simulated by networks of uniform splicing processors whose size depends linearly on the alphabet of the Turing machine. Furthermore, the simulation was time efficient. We argue that the network size can be decreased to a constant, namely six nodes. We further show that networks with only two nodes are able to simulate 2-tag systems. After these theoretical results, we discuss a possible software implementation of this model by proposing a conceptual architecture and describe all its components.

Keywords: theory of computation; computational models; Turing machine; 2-tag system; splicing; splicing processor; network of uniform splicing processors; Apache Giraph; Apache Spark; Spark GraphX architecture; NSUP engine

1. Introduction

In the last two decades, computational models inspired by different phenomena that appear in nature have been vividly researched from a formal perspective. Several theoretical studies of these models have shown that they could solve intractable problems in an efficient way. Several classes of bio-inspired computational models have been surveyed in [1].

We place now our contribution into the general framework of bio-inspired and other parallel and distributed computing models. In this context, networks of bio-inspired processors constitute a group of parallel and distributed computational models having two characteristics: they are abstracted from biological phenomena and their parallelism is extremely high (theoretically unbounded). There is a huge degree of similarity between these networks and other models of computation with related or different origins: *tissue-like P systems* ([2]) in the membrane computing area ([3]), *evolutionary systems* abstracted from the evolution of various cell populations ([4]), *networks of parallel language processors*, which has been introduced as a parallel language-theoretic model ([5]), *flow-based programming*, which is a programming paradigm widely known ([6]), *connection machine*, which may be viewed as a network,

in the shape of a hypercube, consisting of microprocessors that process one bit per unit time ([7]), *distributed computing using mobile programs* ([8]), etc.

Roughly speaking, networks of bio-inspired processors can be depicted as graphs whose vertices host processors running operations on data organized in different ways: strings, pictures, graphs, multisets, etc. In the particular area of networks of bio-inspired processors, two main types of networks of bio-inspired processors, where the processed data is structured as strings, have been studied so far: evolutionary processors and splicing processors. A splicing processor ([9]) performs a formal operation, which is abstracted from the splicing of DNA molecules controlled by diverse types of enzymes ([10]). The name of the formal operation is *splicing*. DNA splicing makes it possible to genetically modify a biological cell for different aims like: plants with a higher resistance to infestation, organisms that are better adapted to environment changes, production of various hormones, etc. This type of DNA recombination is controlled by two types of enzymes: restriction enzymes and ligases. The role of restriction enzymes is to cut the DNA sequences at some recognition sites yielding two DNA sequences having the so-called “sticky ends”, while the role of ligase is to rejoin the sequences produced by the restriction enzymes. In [11], a computational model was introduced, called splicing system, that generates sets of strings by using an operation abstracted from the DNA splicing. Roughly speaking, two strings are used to symbolize the two DNA molecules while quadruples of strings, called splicing rules, symbolize the functions of restriction enzymes indicating where the two previous strings could be cut. The fragments obtained by cutting the strings can be rejoined provided that they were acquired by using the same splicing rule.

Networks of splicing processors were proposed in [9]. This model resembles some aspects of the test tube distributed splicing systems introduced in [12] and investigated in [13] later on. The discrepancies between the models contemplated in [9,12] are accurately detailed in [9]. In [9] one also mentions the disparities between the networks of splicing processors and another distributed system based on splicing, namely the time-varying distributed H systems introduced in [14].

The computation in a network of splicing processors is a sequence of alternative steps: splicing and communication ones. This computation halts when a predefined condition is met. In every splicing step, all processors run simultaneously all splicing rules that could be applied on different copies of the strings that exist in the nodes hosting the respective processor. Therefore, the data in each node is actually a multiset of strings, which means that each string may have sufficiently many copies such that when several rules can be applied to the same string, they are all actually applied in a maximal parallel way to copies of that string. In each communication step, two actions are done according to different strategies:

- (i) simultaneously, in all the nodes of the networks, the data that are able to pass the filter of its node leaves that node and tries to enter all adjacent nodes;
- (ii) simultaneously, all the nodes handle the arriving data.

The communication strategies examined so far are based on filters that allow or forbid strings to enter nodes or go out from them. The filters considered so far are defined by two types of conditions: semantic and syntactical ones. In the case of filters based on syntactical conditions in networks of splicing processors, there are two variants: (i) each node has two filters that could be different, namely an input and an output filter ([9]) and (ii) filters are moved from the nodes to the edges ([15]). This variant may be understood as of the four filters of every pair of adjacent nodes collapse into just two filters that are placed on the edge between them.

We discuss here the main two contributions of this work. We consider here a new variant, somehow “in between” the two aforementioned variants. More precisely, the two filters associated with nodes collapse to only one such that the input and the output filters coincide. This variant, which has been considered for networks of evolutionary processors [16], has never been considered for networks of splicing processors.

We consider here the computational power of this variant. Although the new variants seems less powerful than the other two aforementioned ones, we prove that they have the same computational

power. To this aim, we compared the computational power of this variant with that of other two abstract computational models: Turing machine and 2-tag system. The results we obtained are as follows. We first performed a simulation of nondeterministic Turing machines that preserves the time complexity of the machine but the number of nodes depends linearly on the number of symbols in the alphabet of the machine. Further on, we argue that the construction might be used to reduce the number of nodes to a constant, namely 6. We then investigated the possibility to simulate another computationally complete model, namely the 2-tag system with these networks. We performed such a simulation having two important properties: each application of a tag operation is simulated in 8 computational steps (4 splicing steps and 4 communication steps) by the network and the size of the simulating network is optimal, that has 2 nodes only. The methods and techniques used in these simulations were the standard ones in such types of investigations: we explain how each computational step of a Turing machine or a 2-tag system can be simulated by a number of steps in a network of uniform splicing processors. It is worth mentioning that similar results have not been reported for the variant introduced in [16].

After these theoretical results, we discuss a possible software implementation of this model. This is the second main contribution of the paper. To this aim, we propose a conceptual architecture, which means the development of an NSUP engine using the GraphX component within the Spark framework. All the components of this architecture together with the way in which they are interconnected are amply described.

The paper is structured as follows. In the next section we recall the main definitions and concepts necessary to understand the paper. In the same section, we introduce the main concept, namely the network of uniform splicing processors. The third section discusses an efficient simulation of the Turing machine, which can eventually be accomplished by networks with only six uniform splicing processors. The fourth section presents an efficient simulation of another computationally complete model, that of a 2-tag system. In this case, the simulation can be accomplished by a network with only two uniform splicing processors, hence the simulation is an optimal one.

2. Basic Definitions

In the sequel, we define the basic concepts and notations that are to be used in this work; for all unexplained notions the reader is referred to [17].

A finite and non-empty set whose elements are called symbols is said to be an *alphabet*. Given a finite set A we denote its cardinality by $\text{card}(A)$. A *string* over an alphabet V is defined as a finite sequence of symbols from an alphabet V . We denote by V^* the set of all strings over V , the empty string is denoted by λ , and the length of the string x is denoted by $|x|$. Furthermore, $\text{alph}(x)$ is the minimal alphabet U such that $x \in U^*$ holds.

We now formally define the splicing operation following [10]. A *splicing rule* over the alphabet V is a quadruple of strings of the form $[(v_1, v_2); (v_3, v_4)]$, where v_1, v_2, v_3 , and v_4 are strings in V^* . Given a splicing rule $r = [(v_1, v_2); (v_3, v_4)]$ as above and the strings $x, y, z \in V^*$, we say that z is the result applying r to x and y (and denote this by $(x, y) \vdash_r z$) if x and y can be written as $x = x_1 v_1 v_2 x_2$, and $y = y_1 v_3 v_4 y_2$, for some $x_1, x_2, y_1, y_2 \in V^*$, and $z = x_1 v_1 v_4 y_2$. If L is a language over V and R is a set of splicing rules, we define

$$\sigma_R(L) = \{z \in V^* \mid \exists x, y \in L, \exists r \in R \text{ such that } (x, y) \vdash_r z\}.$$

We first note that a string of L belongs to $\sigma_R(L)$ if and only if it can be obtained by applying a splicing rule to two (possibly the same) strings in L . Second, we note that the definition of a splicing rule from above is actually a 1-splicing rule as defined in [10]. In what follows, we actually use 2-splicing rules as we do not distinguish between the two strings that can arise from the application of a splicing rule.

Let V be an alphabet; we now define two predicates for a string $z \in V^+$ and two disjoint subsets P, F of V as follows:

$$\begin{aligned}\varphi^{(s)}(z; P, F) &\equiv P \subseteq \text{alph}(z) \wedge F \cap \text{alph}(z) = \emptyset \\ \varphi^{(w)}(z; P, F) &\equiv \text{alph}(z) \cap P \neq \emptyset \wedge F \cap \text{alph}(z) = \emptyset.\end{aligned}$$

In the definition of these predicates, the set P is a set of *permitting symbols* while the set F is a set of *forbidding symbols*. Informally, both conditions require that no forbidding symbol occurs in z . As one can see, the former condition is stronger than the second one since it requires that all permitting symbols are present in z , while the latter requires that at least one permitting symbol appears in z .

These predicates are extended to a language $L \subseteq V^*$ by

$$\varphi^\beta(L, P, F) = \{w \in L \mid \varphi^\beta(w; P, F)\},$$

with $\beta \in \{(s), (w)\}$.

A *splicing processor* over an alphabet V is a 6-tuple (S, A, PI, FI, PO, FO) , where:

- S is a finite set of splicing rules over V .
- A is a finite set of *auxiliary strings* over V . These auxiliary strings are to be used, together with the existing strings, in the splicing steps of the processors. Auxiliary strings are available at any moment.
- $PI, FI \subseteq V$ are the sets of permitting and forbidding symbols, respectively, which form the *input filter* of the processor.
- $PO, FO \subseteq V$ are the sets of permitting and forbidding symbols, respectively, which form the *output filter* of the processor.

A splicing processor as above is said to be *uniform* if $PI = PO = P$ and $FI = FO = F$. For the rest of this note we deal with uniform splicing processors only. The set of uniform splicing processors over V is denoted by USP_V .

A *network of uniform splicing processors* (NUSP) is a 9-tuple $\Gamma = (V, U, \langle, \rangle, G, \mathcal{N}, \alpha, \underline{In}, \underline{Halt})$, where:

- V and U are the input and network alphabet, respectively, $V \subseteq U$, and $\langle, \rangle \in U \setminus V$ are two special symbols.
- $G = (X_G, E_G)$ is an undirected graph without loops with the set of nodes X_G and the set of edges E_G . Each edge is given in the form of a binary set. G is called the *underlying graph* of the network. In almost all works on networks of splicing processors (see, e.g., the survey [18]), the underlying graph is a complete graph.
- $\mathcal{N} : X_G \rightarrow USP_U$ is a mapping, which associates with each node $x \in X_G$ the splicing processor $\mathcal{N}(x) = (S_x, A_x, P_x, F_x)$.
- $\alpha : X_G \rightarrow \{(s), (w)\}$ defines the type of the filters of a node.
- $\underline{In}, \underline{Halt} \in X_G$ are the *input* and the *halting* node of Γ , respectively.

The *size* of an NUSP Γ is defined as the number of nodes of the graph, i.e., $\text{card}(X_G)$. A *configuration* of an NUSP Γ is a mapping $C : X_G \rightarrow 2^{U^*}$, which associates a set of strings with every node of the graph. Although a configuration is a multiset of strings, each one appearing in an arbitrary number of copies, for the sake of simplicity, we work with the support of this multiset. A configuration can be seen as the sets of strings, except the auxiliary ones, which are present in the nodes at some moment. For a string $w \in V^*$, we define the initial configuration of Γ on w by $C_0^{(w)}(\underline{In}) = \{\langle w \rangle\}$ and $C_0^{(w)}(x) = \emptyset$ for all other $x \in X_G$.

A configuration is followed by another configuration either by a splicing step or by a communication step. A configuration C' follows a configuration C by a splicing step if each component

$C'(x)$, for some node x , is the result of applying all the splicing rules in the set S_x that can be applied to the strings in the set in $C(x)$ together with those in A_x . Formally, configuration C' follows the configuration C by a splicing step, written as $C \Rightarrow C'$, iff for all $x \in X_G$, the following holds:

$$C'(x) = \sigma_{S_x}(C(x) \cup A_x).$$

In a communication step, the following actions take place simultaneously for every node x :

- (i) all the strings that can pass the output filter of a node are sent out of that node;
- (ii) all the strings that left their nodes enter all the nodes connected to their original ones, provided that they can pass the input filter of the receiving nodes.

Note that, according to this definition, those strings that are sent out of a node and cannot pass the input filter of any node are lost. Formally, a configuration C' follows a configuration C by a communication step (we write $C' \models C$) iff for all $x \in X_G$

$$C'(x) = (C(x) - \varphi^{\beta(x)}(C(x), P_x, F_x)) \cup \bigcup_{\{x,y\} \in E_G} (\varphi^{\beta(y)}(C(y), P_y, F_y) \cap \varphi^{\beta(x)}(C(y), P_x, F_x))$$

holds. For an NUSP Γ , a computation on an input string w is defined as a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \dots$, where $C_0^{(w)}$ is the initial configuration of Γ on w , $C_{2i}^{(w)} \Rightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \models C_{2i+2}^{(w)}$, for all $i \geq 0$. A computation on an input string w *halts* if there exists $k \geq 1$ such that $C_k^{(w)}(\underline{Halt})$ is non-empty. Such a computation is called an *accepting computation*.

The language accepted by Γ is defined as

$$L(\Gamma) = \{z \in V^* \mid \text{the computation of } \Gamma \text{ on } z \text{ is an accepting computation}\}.$$

Given an NUSP Γ with the input alphabet V , we define the following computational complexity measure. The *time complexity* of the finite computation $C_0^{(x)}, C_1^{(x)}, C_2^{(x)}, \dots, C_m^{(x)}$ of Γ on $x \in V^*$ is denoted by $Time_\Gamma(x)$ and equals m . The time complexity of Γ is the partial function from $\mathbf{N}$ to $\mathbf{N}$,

$$Time_\Gamma(n) = \max\{Time_\Gamma(x) \mid x \in L(\Gamma), |x| = n\}.$$

3. Efficient Simulations of Turing Machines

The main result of this note is a time efficient simulation of Turing machines by NUSP. As there are many variants of Turing machines, we start by giving the definition of the Turing machine we are working with. Formally, a Turing machine is a 7-tuple $M = (Q, V, U, \delta, q_0, B, F)$, where Q is a finite set of states, V is the input alphabet, U is the tape alphabet, $V \subset U$, q_0 is the initial state, $B \in U \setminus V$ is the “blank” symbol, $F \subseteq Q$ is the set of final states, and δ is the transition function, $\delta : (Q \setminus F) \times U \rightarrow 2^{Q \times (U \setminus \{B\}) \times \{R,L\}}$.

Our variant is a machine with one semi-infinite tape bounded to the left. This tape can store strings over the alphabet $U \setminus \{B\}$; initially it contains the input string which is a string over the alphabet V . The machine has a tape head that can scan just one symbol of the input string at any moment, and a control unit that can be in a state from Q . An *operation* of M is defined as follows: the tape head scans a symbol from the input string and, depending on the current state of the control unit and the symbol scanned, writes a symbol different from B over the scanned symbol, moves to the right or to the left, provided that the scanned cell is not the leftmost one of the tape, while the control unit may change its state.

A computation of M on an input string is a finite or infinite sequence of operations as above. When the control unit enters a final state, the computation halts and the input string is accepted, hence the language accepted by M consists of all accepted strings. A Turing machine is deterministic if for every state and symbol, the machine can make at most one operation.

The proof of Theorem 1 from [19] can be used to prove that deterministic Turing machines can be efficiently simulated by NUSP. More precisely,

Theorem 1. *If a language is accepted by a deterministic Turing machine in $\mathcal{O}(f(n))$ time, for some function from $\mathbb{N}$ to $\mathbb{N}$, then it is accepted by an NUSP of size 2 in $\mathcal{O}(f(n))$ time.*

Consequently,

Corollary 1. *The class of polynomially recognizable languages is included in the class of languages accepted by NUSPs of size 2 in polynomial time.*

A natural problem arises: Are NUSPs able to efficiently simulate a nondeterministic Turing machine? In the affirmative, does the size remain constant? It is known that every nondeterministic Turing machine can be simulated by a deterministic one, which in its turn can be efficiently simulated by NUSPs of size 2. However, it is also known that the former simulation increases the time complexity of the deterministic machine. We prove now that a nondeterministic Turing machine can be efficiently simulated (preserving the same time complexity) by an NUSP, but the size of the network depends linearly on the cardinality of the Turing machine alphabet.

Theorem 2. *If a language is accepted by a nondeterministic Turing machine with an alphabet U in $\mathcal{O}(f(n))$ time, then it is accepted by an NUSP of size $2 \cdot \text{card}(U) + 2$ in time $\mathcal{O}(f(n))$.*

Proof. The network contains along the input and halting node, two nodes for each symbol of the working alphabet of the Turing machine, and two further nodes Sim and Res. We shall now give the formal description of all the nodes (set of rules, axioms, permitting and forbidding symbols, respectively).

Let $M = (Q, V, U, \delta, q_0, B, F)$ be a nondeterministic Turing machine; we now define the NUSP $\Gamma = (V, U', \langle, \rangle, G, \mathcal{N}, \alpha, \underline{In}, \underline{Halt})$, where

$$\begin{aligned} U' = & (U \setminus \{B\}) \cup \{\langle^q \mid q \in Q \setminus F\} \cup \\ & \{\langle^{q,b,R} \mid q \in Q, b \in U \setminus \{B\}\} \cup \\ & \{\langle^{q'} \mid q \in Q\} \cup \{\langle^{q,L} \mid q \in Q\} \cup \{\$, \#, \rangle', \&, \ddagger\}, \end{aligned}$$

G is a complete graph, while the nodes of Γ are defined as follows:

- **In**
 - $S_{\underline{In}} = \{[(\langle, \lambda); (\langle^{q_0}, \#)], [(\lambda, \rangle); (\#, \$)']\}$.
 - $A_{\underline{In}} = \{\langle^{q_0\#}, \#\$'\}$.
 - $P_{\underline{In}} = \{\rangle'\}$.
 - $F_{\underline{In}} = \emptyset$.
 - $\alpha(\underline{In}) = w$.
- **Sim**
 - $S_{\underline{Sim}} = \{[(\langle^a, \lambda); (\langle^{s,b,R}, \#)] \mid a \in U \setminus \{B\}, q \in Q \setminus F, (s, b, R) \in \delta(q, a)\} \cup$
 $\{[(\langle^q, \lambda); (\langle^{s,b,R}, \#)] \mid q \in Q \setminus F, (s, b, R) \in \delta(q, B)\} \cup$
 $\{[(\langle^a, \lambda); (\langle^{s,L}, \#)] \mid a, b \in U \setminus \{B\}, q \in Q \setminus F, (s, b, L) \in \delta(q, a)\} \cup$
 $\{[(\langle^q, \lambda); (\langle^{s,L}, \#)] \mid b \in U \setminus \{B\}, q \in Q \setminus F, (s, L) \in \delta(q, B)\}$.
 - $A_{\underline{Sim}} = \{\langle^{q,b,R}, \# \mid q \in Q, b \in U \setminus \{B\}\} \cup$
 $\{\langle^{q,L}, \# \mid q \in Q, b \in U \setminus \{B\}\}$.
 - $P_{\underline{Sim}} = \{\langle^q \mid q \in Q \setminus F\} \cup \{\langle^{q,b,R} \mid q \in Q, b \in U \setminus \{B\}\} \cup \{\langle^{q,L} \mid q \in Q\}$.

- $F_{\underline{Sim}} = \{\&, \# \ddagger\}$.
- $\alpha(\underline{Sim}) = w$.
- **Rins_b**, $b \in U \setminus \{B\}$
 - $S_{\underline{Rins}_b} = \{[(X, \rangle'); (\#, b\&)] \mid X \in (U \setminus \{B\}) \cup \{\$\}\} \cup \{[(\langle^{q,b,R}, X); (\langle^{q'}, \#)] \mid q \in Q, X \in (U \setminus \{B\}) \cup \{\$\}\}$.
 - $A_{\underline{Rins}_b} = \{(\#b\&) \cup \{\langle^{q'} \# \mid q \in Q\}$.
 - $P_{\underline{Rins}_b} = \{\langle^{q,b,R} \mid q \in Q\} \cup \{\&\}$.
 - $F_{\underline{Rins}_b} = \{\#, \ddagger\}$.
 - $\alpha(\underline{Rins}_b) = w$.
- **Lins_b**, $b \in U \setminus \{B\}$
 - $S_{\underline{Lins}_b} = \{[(X, b \rangle'); (\#, \ddagger)] \mid X \in (U \setminus \{B\}) \cup \{\$\}\} \cup \{[(\langle^{q,L}, X); (\langle^{q'} b, \ddagger)] \mid q \in Q, X \in (U \setminus \{B\}) \cup \{\$\}\}$.
 - $A_{\underline{Lins}_b} = \{(\# \ddagger) \cup \{\langle^{q'} b \ddagger \mid q \in Q\}$.
 - $P_{\underline{Lins}_b} = \{\langle^{q,L} \mid q \in Q\} \cup \{\ddagger\}$.
 - $F_{\underline{Lins}_b} = \{\#, \&\}$.
 - $\alpha(\underline{Lins}_b) = w$.
- **Res**
 - $S_{\underline{Res}} = \{[(X, Y); (\#, \rangle')] \mid X \in (U \setminus \{B\}) \cup \{\$\}, Y \in \{\&, \ddagger\} \cup \{[(\langle^{q'}, X); (\langle^{q'}, \#)] \mid q \in Q, X \in (U \setminus \{B\}) \cup \{\$\}\}$.
 - $A_{\underline{Res}} = \{(\# \&, \# \ddagger) \cup \{\langle^{q'} \# \mid q \in Q\}$.
 - $P_{\underline{Res}} = \{\&, \ddagger, \rangle'\}$.
 - $F_{\underline{Res}} = \{\langle^{q,b,R} \mid q \in Q, b \in U \setminus \{B\}\} \cup \{\langle^{q,L} \mid q \in Q\} \{\#, \langle\}$.
 - $\alpha(\underline{Res}) = w$.
- **Out**
 - $S_{\underline{Halt}} = A_{\underline{Out}} = F_{\underline{Halt}} = \emptyset$.
 - $P_{\underline{Halt}} = \{\langle^q \mid q \in F\}$.
 - $\alpha(\underline{Halt}) = w$.

The input string $\langle w \rangle$ is transformed into $\langle^{q_0} w \$ \rangle'$ by two splicing steps in In, where q_0 is the initial state of the Turing machine. Now the string goes out from In and one copy enters Sim. If only one of the splicing step were done in In, two strings would be obtained, namely $\langle^{q_0} w \rangle$ and $\langle w \$ \rangle'$. We analyze what could happen with the two strings. The first cannot leave In, while the second one goes out from In but it cannot enter any node, hence it is lost. Now the network performs a “rotate-and-simulate” strategy. In Sim, the string $\langle^{q_0} w \$ \rangle'$ is processed as follows. Its first symbol $\langle^{q_0}$ (inductively, $\langle^q$) is simultaneously replaced by $\langle^{s,b,R}$ or $\langle^{s,b,L}$, where $(s, b, R) \in \delta(q_0, a)$ and $(s, b, L) \in \delta(q_0, a)$, in different copies of $\langle^{q_0} w \$ \rangle'$, where a is the first symbol of w . Now, the new strings enter the nodes associated with the corresponding transitions where those transitions are to be simulated. More precisely, they simultaneously enter Rins_b and/or Lins_b. Each node Rins_b and Lins_b, $b \in U \setminus \{B\}$, performs two splicing steps over the strings it receives. Note that the strings generated by only one splicing step cannot affect the computation. Indeed, they either cannot leave their nodes or can leave their nodes but there is no node that can accept them.

Let us follow the computation on the strings of the form $\langle^{s,b,R} x \$ y \rangle'$, $s \in Q$, $b \in U \setminus \{B\}$, $x, y \in (U \setminus \{B\})^*$, arriving in Rins_b. After one splicing step, the node yields $\langle^{s,b,R} x \$ y b \&, \# \rangle'$, $\langle^{s,b,R} \#$ and $\langle^{s'} x \$ y \rangle'$. All these strings, except the first one, are unable to exit the node, hence they will

remain there for a further splicing step. The first string goes out but it cannot enter any node, therefore it is lost. After the next splicing step, the strings $\#'$, $\langle^{s,b,R}\#$ disappear, while a new string is produced, namely $\langle^{s'}x\$yb\&$, which goes out and enters Res only. In this way, Γ simulates a transition of the form $(s, b, R) \in \delta(q_0, a)$ and rotates the string to the right.

The process is similar for Lins_b. A string $\langle^{s,L}x\$y\rangle'$ enters Lins_b. The role of this node is threefold: delete a b from the end of y (if any), insert a b to the left hand end of x , replace $\langle^{s,L}$ by $\langle^{s'}$. We describe below how this can be done. If $y = y'b$, a string $\langle^{s,L}x\$y'\ddagger$ is obtained in one splicing step. In the same splicing step, the string $\langle^{s'}bx\$y\rangle'$ is also obtained. The string $\langle^{s,L}x\$y'\ddagger$ leaves Lins_b, but it is lost because it cannot enter any node. The string $\langle^{s'}bx\$y\rangle'$ cannot leave Lins_b, hence it remains there for a further splicing step in which y is replaced by y' . Now the new strings go out and enter Res. Note that if a string $\langle^{s,L}x\$y\rangle'$, such that y does not end by b , enters Lins_b, it cannot produce any string that is able to go out from Lins_b and enter Res only. However, the string $\langle^{s,L}x\$y\rangle'$ is simultaneously processed successfully in a node Lins_c, where c is the last symbol of y . In this way, Γ simulates a transition of the form $(s, c, L) \in \delta(q_0, a)$ and rotate the string to the left, if this is possible.

All the strings successfully processed in the nodes Rins_b and Lins_b, $b \in U \setminus \{B\}$, can enter Res only. These strings are of the form $\langle^{q'}x\$y\ddagger$ or $\langle^{q'}x\$y\&$, with $q \in Q$, $x, y \in (U \setminus \{B\})^*$. In Rins_b, by a sequence of two splicing steps as above, $\langle^{q'}$ is replaced by $\langle^q$, while $\ddagger$ and $\&$ are replaced by $\rangle'$. All these new strings go out from Res and enter either Sim, provided that $q \in Q \setminus F$, or Halt, provided that $q \in F$. In the last case, the simulation is stopped because the computation of M (whose transitions have been simulated by Γ) reached a final state.

As one can see, each transition of M is simulated by Γ with a constant number of splicing and communicating steps. Therefore, if M accepts in $\mathcal{O}(f(n))$, then $\text{Time}_\Gamma(n) \in \mathcal{O}(f(n))$. Furthermore, the size of Γ is $2 \cdot (\text{card}(U) - 1) + 4 = 2 \cdot (\text{card}(U)) + 2$, which concludes the proof. $\square$

Actually, in the previous proof the symbol b could be considered in $U \setminus (V \cup \{B\})$ only. As it is known that this set can be reduced to the binary alphabet, we may state:

Theorem 3. Every recursively enumerable language can be accepted by an NUSP of size 6.

Furthermore, a closer look at the communication in the network constructed in the previous proof, leads to the fact that the underlying graph G could be replaced by a “double interconnected star” graph as shown in the Figure 1, where $U \setminus \{B\} = \{a_1, a_2, \dots, a_t\}$, for some $t \geq 1$.

Figure 1. Communicating underlying graph.

4. Efficient Simulations of 2-Tag Systems

We define a 2-tag system following a slightly modified version of the definition that appears in Section 8 of [20]. It is worth mentioning that this definition of a 2-tag-system is slightly different

from those proposed in [21,22], but they are equivalent. Formally, a 2-tag system is a pair $T = (V, \mu)$, where V is a finite alphabet containing a special symbol π , called the *halting symbol* (denoted by *STOP* in [20]), and a finite set of rules $\mu : V \setminus \{\pi\} \rightarrow V^+$ such that $|\mu(x)| \geq 1$ or $\mu(x) = \pi$. Furthermore, $\mu(x) = \pi$ for just one $x \in V \setminus \{\pi\}$. A string that contains the halting symbol or is of length less than 2 is called a halting string.

We now define the mapping t_T (called the tag operation) on the set of non-halting strings. For a non-halting string w , $t_T(w)$ is the string produced as follows: the first two symbols of w are deleted and $\mu(a)$ is appended, provided that a is the first symbol of w . As one can see, this operation deletes two symbols from the beginning of the string and appends to the end a string that depends solely on the first symbol read. A computation by a 2-tag system as above on the initial non-halting string w is a sequence of strings produced by applying iteratively the operation t_T starting with w . A computation as above halts on w if a halting string is produced in some iteration. The halting strings of a length of at least two are defined in [20] in a different way, namely such a string is halting if it starts with a symbol a such that $\mu(a) = \pi$. It is a trivial exercise to show that there exists a bijection between the finite computations obtained in each of the two cases. As shown in [20], such 2-tag systems are computationally complete.

The *time complexity* of the finite computation of T on $w \in V^*$: $w_0 = w, w_1 = t_T(w_0), w_2 = t_T(w_1), \dots, w_p = t_T(w_{p-1}) = \alpha\pi$, with $w_i \in (V \setminus \{\pi\})^+, |w_i| \geq 2$ for all $0 \leq i \leq p-1$, and $\alpha \in (V \setminus \{\pi\})^*$, is denoted by $\text{Time}_T(w)$ and equals p . In other words, the time complexity of a finite computation of T on a string is the number of applications of the tag operations necessary for the 2-tag system to produce a halting string. The time complexity of T is the partial function from $\mathbb{N}$ to $\mathbb{N}$, $\text{Time}_T(n) = \max\{\text{Time}_T(w) \mid w \in V^*, |w| = n\}$.

We recall here that many small universal Turing machines proposed in the literature have been obtained by simulations of 2-tag systems, see, e.g., [20], and more recently [23] together with the references therein.

It is known that 2-tag systems can simulate Turing machines but this simulation is exponentially slow [24]. An efficient simulation of Turing machines via a simulation of cyclic tag systems is presented in [23]. The main result in [23] states that given a single tape deterministic Turing machine M that computes in time t then there is a 2-tag system T_M that simulates the computation of M in time $\mathcal{O}(t^4(\log t)^2)$. By the previous simulation it follows that given a single tape deterministic Turing machine M that computes in time t , there is an NUSP Γ_M that simulates the computation of M in time $\mathcal{O}(t^4(\log t)^2)$. The direct simulation presented in the next theorem is more efficient at the price of a larger network size.

Theorem 4. *Given a 2-tag system $T = (V, \mu)$ there exists an NUSP Γ such that every $w \in V^*$ is accepted by Γ if and only if T halts on w . Moreover, if $\text{Time}_T(w) = f(|w|)$, then $\text{Time}_\Gamma(w) \in \mathcal{O}(f(|w|))$.*

Proof. Let $V' = V \setminus \{\pi\}$; we construct the NUSP $\Gamma = (V', U, \langle, \rangle, G, \mathcal{N}, \alpha, \text{In}, \text{Halt})$, where $U = V \cup \{\langle, \#, \$, \rangle', \&\}$, and the nodes $x \in X_G$ are defined as follows:

- **In**
 - S_{In} is the union of the following sets:
 - (i) $\{[(\langle, Y); (\$, \#)] \mid Y \in V' \cup \{\}\}\}$,
 - (ii) $\{[(\$aX, Y); (\langle', \# \mu(a) \#)'] \mid a \in V' \text{ such that } \mu(a) \neq \pi, \text{ and } X \in V', Y \in V' \cup \{\}\}\}$,
 - (iii) $\{[(Y, \rangle); (\$aX\#, \mu(a)\#)'] \mid a \in V' \text{ such that } \mu(a) \neq \pi, \text{ and } X, Y \in V'\}$,
 - (iv) $\{[(X\mu(a), \#)']; (\$aY\#, \rangle)] \mid a \in V', \text{ such that } \mu(a) \neq \pi, \text{ and } Y \in V', X \in V' \cup \{\langle'\}\}$,
 - (v) $\{[(\langle', a); (\$, aY\#\#)'] \mid a \in V' \text{ such that } \mu(a) \neq \pi, \text{ and } Y \in V'\}$,
 - (vi) $\{[(\$aX, Y); (\&, \$)] \mid a \in V' \text{ such that } \mu(a) = \pi, X \in V', Y \in V' \cup \{\}\}\}$,
 - (vii) $\{[(Xa, \rangle); (\&, \&)] \mid X \in \{\langle, \$\}\}$.

- $A_{\underline{In}} = \{\langle \# \mu(a) \# \rangle' \mid a \in V' \text{ and } \mu(a) \neq \pi\} \cup \{\$ \#, \langle \#, \& \$ \rangle\}.$
 - $P_{\underline{In}} = \{\langle \& \rangle\}$
 - $F_{\underline{In}} = \{\$ \}.$
 - $\alpha(\underline{In}) = (w).$
- **Halt**
- $S_{\underline{Halt}} = A_{\underline{Halt}} = \emptyset.$
 - $P_{\underline{Halt}} = \{\& \}.$
 - $F_{\underline{Halt}} = \{\# \}.$
 - $\alpha(\underline{Halt}) = w.$

We show that Γ accepts a string w that does not contain π if and only if T eventually halts on w . Let $w = aby, a, b \in V, y \in V^*$ be a string that does not contain π such that T eventually halts on w . We show how w can be accepted by Γ . To this aim we show how a tag operation in T is simulated by Γ . We first assume that $\mu(a) \neq \pi$. At the beginning of the computation of Γ , in the node $\underline{In}$, a splicing rule in the set (i) is applied to the input string $\langle aby \rangle$ and the axiom $\$ \#$. This operation yields two strings: $\$ aby \rangle$ and $\langle \#$. The second string gets through the filters of $\underline{In}$ and leaves it. However, it is lost because it cannot be accepted by the other node in the network, namely $\underline{Halt}$, as a result of the forbidden character $\#$. On the other hand, the first string $\$ aby \rangle$ cannot pass the forbidding filter of $\underline{In}$ and remains in this node for further operations.

In the next splicing step in $\underline{In}$ a rule from (ii) is applied to the previously obtained string $\$ aby \rangle$ and to the axiom $\langle \# \mu(a) \# \rangle'$. The result of this splicing step is the pair of strings $\langle y \rangle$ and $\$ ab \# \mu(a) \# \rangle'$. Since these new strings cannot pass the filter of $\underline{In}$, they remain there for the next splicing step. At this moment, only one action is possible, namely to apply a rule in set (iii) to the pair of strings just obtained. This splicing step yields a new pair of strings, namely $\langle y \mu(a) \# \rangle'$ and $\$ ab \#$. These two strings, which cannot leave $\underline{In}$, are to be further involved into a splicing step by using a rule in set (iv) producing the new pair of strings $\langle y \mu(a) \rangle$ and $\$ ab \# \# \rangle'$. Again, the next splicing step is uniquely determined. Indeed, only a rule from (v) can be applied to the pair of strings $\langle y \mu(a) \rangle$ and $\$ ab \# \# \rangle'$ yielding $\$ y \mu(a) \rangle$ and $\langle ab \# \# \rangle'$. This splicing step finishes the simulation of the tag operation $\mu(a)$ applied to w in T .

More precisely, $\$ y \mu(a) \rangle$ corresponds exactly to the transformation of aby in T by applying $\mu(a) \neq \pi$, while the second string is to be lost in the next splicing step because there is no string that can be paired up with it in a splicing step. The process is resumed with the new string $\$ y \mu(a) \rangle$ until a string $\$ bz \rangle$, with $\mu(b) = \pi$ is obtained. When such a string is obtained, say $\$ bcz \rangle$, by using a rule in set (vi) and the axiom $\& \$$ the string $\& z \rangle$ is obtained which leaves $\underline{In}$ and enters $\underline{Halt}$ and the computation of Γ on w halts.

Finally, let us consider the special halting cases of a string of length 1. There are two cases: the current string is either an input string $\langle a \rangle$, for some $a \in V$, or a string obtained via a series of computations as above $\$ a \rangle, a \in V$. A rule in (vii) can be applied to such a string and the axiom $\& \&$ producing two strings such that at least one of them goes out from $\underline{In}$ and enters $\underline{Halt}$. Note that the axiom $\& \&$ disappears after every splicing step when there is no string as above in $\underline{In}$. In this way, it cannot halts the computation illegally.

Conversely, it is clear that a computation of Γ on an input string halts if and only if the input string is of length one or it eventually leads to the string containing $\&$. This means that T halts on that input string.

It is easy to note that Γ emulates one transformation of T in four steps. Consequently, $Time_{\Gamma}(w) \in \mathcal{O}(f(|w|))$ holds, and the proof is complete. $\square$

5. Towards a Computational Simulation of NUSP

Over the last decade, several implementations of Networks of Bio-inspired Processors (NBP for short) on software computational platforms have been reported in the literature. Various architectures, technologies, and strategies have been used in order to handle the large data generated by NBP algorithms within a massively parallel and distributed environment. With the advent of ultra-scalable computational platforms in Big Data scenarios, a new line for software development of NBP simulators was opened. In particular, a highly scalable engine developed with *Apache Giraph* on top of the *Hadoop* platform was proposed in [25]. This engine, NPEPE, is able to run Networks of Polarized Evolutionary Processors (NPEP) algorithms and some other NBP variants' algorithms ([26]). *Apache Giraph* ([27]) is an iterative graph processing system built for highly scalable and fault tolerant distributed computations. *Giraph* was developed as the open-source counterpart to *Pregel*, which was proposed by Google. Its architecture for parallel computation is based on the well-known "Bulk Synchronous Parallel" (BSP) model defined in [28]. BSP was introduced for guiding the design and implementation of parallel algorithms for large-graph processing. Basically, the BSP model implemented by *Pregel* and *Giraph* defines two step processes, which are performed iteratively and synchronously: (1) one process performing computations on local data and (2) one process communicating the results. Each iteration (computation or communication) is called a *superstep* and represents atomic units of the parallel computation.

The *Apache Hadoop* software library is a framework able to execute distributed processing of large data sets across clusters of computers using a simple programming model based on *Map* and *Reduce* operations.

Apache Spark has emerged as a fast and general-purpose cluster computing framework able to execute high performance parallel computations in an efficient way. *Spark* was proposed to resolve some drawbacks found in existing *Map Reduce* implementations (bottlenecks in the cluster communication process, inefficiencies in accessing data, among others). In particular, *Spark* resolves these drawbacks proposing a new programming model that uses a distributed, immutable and fault-tolerant memory abstraction named *Resilient Distributed Datasets* (RDDs). RDDs are able to maintain data in memory, disk, or in combination of both rather than only in disk (such as previous *Map Reduce* implementations). Some experiments demonstrated that *Spark* outperforms, by up to two orders of magnitude, conventional *Map Reduce* jobs in terms of speed, see, e.g., [29,30].

For specific graph-parallel processing, *Spark* offers the *GraphX* library. *GraphX* uses the concept of *Property Graph* (directed multigraph) in combination with RDDs to execute graph algorithms through partitioning and distributing techniques. *GraphX* reduces the excessive data movement and processing inefficiencies exhibited by their predecessors by proposing the unification of data-parallel and graph-parallel methods into one unified library working on *Spark* framework.

As *Giraph* became a natural architecture to implement NPEP models through the NPEPE engine [25], we consider *Spark GraphX* to be a suitable framework to offer high performance and efficiency in the computations (by improving the data movement in the communication processes) in some NBP models requiring these capabilities.

Therefore, we consider that *Spark GraphX* is an adequate framework to fit the specific and novel nature of NUSP model. In particular, splicing rules are more complex than evolutionary rules (used in different variants of networks of evolutionary processors) due to, among other reasons, the fact that these operations involve two strings and several suboperations: identify the splicing sites in each string to which the splicing rule is to be applied, cut these strings and finally, rejoin the obtained substrings in accordance with their order in the original strings. It is clear that simulating splicing rules requires more computational resources, such as memory, therefore an efficient use of all the computational resources of the model.

To our best knowledge, there is no hardware or software implementations of NBP models using splicing steps. In this section we introduce a conceptual and informal discussion proposing the main

guidelines to address the development of an NUSP engine on Spark GraphX which seems more suitable to address the complexity of this bio-inspired computational model due to its splicing rules.

We start with an informal discussion, hence we consider an architecture for general purpose which is shown in Figure 2.

Figure 2. Spark GraphX architecture proposed for NSUP framework.

Following this architecture, our suggestion is to develop the NSUP engine using GraphX component within the Spark framework working in collaboration with two additional components, namely: an Input component or **NSUP editor** and an Output component or **NSUP collector**.

The **NSUP editor** should be developed in order to provide functionalities that allow the final users to define and describe the NSUP algorithm in a friendly way. This component must facilitate a more expressive manner to design an NSUP algorithm avoiding mistakes in the notation when defining splicing rules in text editors. Following the same idea, the **NSUP collector** should be able to obtain the output of an NSUP network computation from the halting node and show it in a friendly manner such that it can be comprehensible to the final user.

On the other hand, the **NSUP engine** should be able to extend the class utilities offered by GraphX in order to execute the splicing and communication steps defined in the NSUP model dynamics. These three modules and the way in which they are either connected or mapped into the GraphX framework are described as follows:

NSUP editor: this component will be the input connection with the NSUP engine and GraphX. Given that GraphX supports large scale graphs, this component should be able to handle several input file formats such as text and JSON, in order to encode underlying NSUP graph definitions. In particular, these formats are suitable for loading graph configurations using the utility class of GraphX called *GraphLoader*. Therefore, this component should generate a NSUP network configuration from final user's definition using these input file formats. We consider that JSON format is more suitable to define the structure of NSUP algorithms and the configuration of its components: splicing rules, axioms, alphabets, nodes, etc. The details of these components should be introduced by final user through a graphical user interface. Then, NSUP editor should generate two configuration files: (1) "configFile.json" representing the definition for NSUP graph components and (2) "configFile.txt" containing the definition of the input word encoding the problem instance which is to be solve by this NSUP configuration.

NSUP engine: the first component to be implemented within the NSUP engine is the **Input module**. This module reads the JSON file generated by NSUP editor and uses the *GraphLoader* class utility (GL) to instantiate the specific *Property Graph* and *Vertex* classes from GraphX. The interaction between NSUP editor and the Input module is illustrated in Figure 3.

Figure 3. NSUP editor and Input module components.

The second component, named the **NSUP Computation Module**, should use the *Property Graph* instance created by the **Input module** to define the *Spark Context* and the main GraphX components: *Vertex Table* and *Edge Table*. Our NSUP Computation Module will require other additional elements: alphabets (V and U), filters (the sets P and F), axioms, and rules. Specifically, axioms and rules will be instantiated using the stored information into each JSON object defining a splicing node processor. They should be associated with their respective vertex (a row in the *Vertex Table*) by means an unique identifier (ID) into *Axiom* and *Rule Tables*. Similarly, the filters (P and F sets) will be instantiated into *PContext* and *FContext Tables* using their corresponding ID from the *Edge Table*, as it is illustrated in Figure 4. On the other hand, the definitions of the two alphabets should be associated with the respective attributes of the *Spark GraphContext* instance. Note that NSUP elements, such as axioms, rules, and filters, will be converted into GraphX RDD structures. Especial vertices such as the input or halting splicing processor nodes, will be defined as extended *Vertex Property* instances. Particularly, the NSUP Computation Module should read the “configFile.txt” file in order to extract the encoded input word and store it as the only message into the Message Table (Msg Table) of the vertex representing the input node. Each vertex will have associated a Message Table in which it can store its received or processed words. Initially, all vertices except the input one, will have their Message Table empty. We remark that, the NSUP notation is conserved into the GraphX mapping.

Figure 4. NSUP configuration translated into a PropertyGraph of GraphX.

In order to simulate the NSUP model dynamics, the **NSUP Computation Module** should define an execution method to start the computation. This method should activate all vertices but only the input vertex is able to apply its splicing rules over its Message Table content (which contains the message encoding the input word). The splicing operations defined by the splicing rules, will be implemented into the *Vertex class* utility. After the starting step, each vertex should apply its rules (defined into the Rules Table) over incoming messages from input vertex and stores them into its Message Table. At this moment, the *NSUP Computation Module* will run a *splicing step*. When all

rules have been applied, the *NSUP Computation Module* will invoke the GraphX methods from *Pregel API* such that every vertex of a superstep will receive and/or send processed messages according to the associated vertices from the Edge Table. Then, a *communication step* will be executed. After that, the *NSUP Computation Module* should invoke other superstep. Finally, we note that in a *communication step* each vertex can discard the processed messages, store them (in its Msg RDD Table) or send them to all associated vertices depending on the results obtained by the filter application.

NSUP Collector represents a component which will be responsible for dumping the data content of the *Halt* vertex on an external file, which can be easily interpreted by the final users. In addition, this component could dump the messages that each vertex keeps in their respective *Msg Tables* when the computation halts creating a *file for final computation log*.

Summarizing, at the beginning of the NSUP algorithm computation, only the vertex dynamics representing the input node is executed and then this node sends out the obtained messages after applying its splicing rules to the initial message. Afterwards, during each superstep, the NSUP engine through the GraphX invokes all vertices in parallel so that they can process the received messages and can send out those messages able to pass their respective filters. This process runs until the scanning process for the halting vertex returns that the Msg Table for this node is non-empty. This process could be implemented using the dynamic *PageRank algorithm* by configuring the ranks converge property. At the end, all words in the *Msg Table* of the halting vertex should be sent to an *NSUP Collector*.

Finally, we consider that the guidelines introduced in this section are an useful starting point that might contribute to the development of software frameworks to simulate NBP models using splicing rules.

6. Conclusions and Further Work

In this paper, we proved that uniform networks of splicing processors are computationally complete by displaying efficient simulations of two known models with this property: 2-tag systems and Turing machines. While a Turing machine simulation is enough to provide definitive proof, the required number of nodes for the corresponding NUSP depends on the size of the tape alphabet U . This number can be decreased to six. On the other hand, a NUSP simulating 2-tag systems can be implemented with the minimal number of nodes, two: *In* and *Halt*. Additionally, it is known that direct simulations between Turing machines and 2-tag systems are not time efficient, while our simulations avoid this problem by preserving the time complexity of the simulated models.

An attractive problem of theoretical interest is to investigate whether or not the size of NUSP efficiently simulating Turing machines can be still decreased. Along the same theoretical lines, another problem is to design computationally complete NUSP having an underlying graph common in the network area: bus, star, grid, etc.

Another interesting theoretical problem is the following one. As we have mentioned above, there are three variants of NSP which differ from each other by the filtering process. Together with the results reported in this paper, we infer that all these variants are equivalent from the computational power point of view, but this equivalence has been obtained by simulations of other models. It is very natural to investigate the possibility of designing direct simulations between all these variants.

As far as our suggestion for a possible software simulation of NUSP is concerned, we will try to implement the architecture discussed here and make experiments with some algorithms based on NUSP. Depending on the results of these experiments we may improve the simulation or make some changes in the proposed architecture. Having implemented this software, we intend to use it for different algorithms especially in computational geometry like, for instance, [31,32].

Author Contributions: Conceptualization, V.M. and J.A.S.M.; software, S.G.-C.; formal analysis, V.M. and M.P.; investigation, J.R.S.C.; writing—original draft preparation, J.A.S.M.; writing—review and editing, V.M. and S.G.-C.; supervision, V.M.; funding acquisition, M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Autoritatea Națională pentru Cercetare Științifică: POC P-37-257.

Acknowledgments: Work supported by a grant of the Romanian National Authority for Scientific Research and Innovation, project number POC P-37-257.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rozenberg, G.; Back, T.; Kok, J.N. *Handbook of Natural Computing*; Springer: Berlin, Germany, 2012.
2. Martín-Vide, C.; Pazos, J.; Păun, G.; Rodríguez-Patón, A. A new class of symbolic abstract neural nets: tissue P systems. *Lect. Notes Comput. Sci.* **2002**, *2387*, 290–299.
3. Păun, G. *Membrane Computing. An Introduction*; Springer: Berlin/Heidelberg, Germany, 2002.
4. Csuha-Varjú, E.; Mitrana, V. Evolutionary systems: A language generating device inspired by evolving communities of cells. *Acta Inform.* **2000**, *36*, 913–926. [[CrossRef](#)]
5. Csuha-Varjú, E.; Salomaa, A. Networks of parallel language processors. *Lect. Notes Comput. Sci. (LNCS)* **1997**, *1218*, 299–318.
6. Morrison, J. *Flow-Based Programming: A New Approach to Application Development*; CreateSpace: Scotts Valley, CA, USA, 2010.
7. Hillis, D. *The Connection Machine*; MIT Press: Cambridge, MA, USA, 1986.
8. Gray, R.; Kotz, D.; Nog, S.; Rus, D.; Cybenko, G. Mobile agents: The next generation in distributed computing. In Proceedings of the IEEE International Symposium on Parallel Algorithms Architecture Synthesis, Aizu-Wakamatsu, Japan, 17–21 March 1997; pp. 8–24.
9. Manea, F.; Martín-Vide, C.; Mitrana, V. Accepting networks of splicing processors: Complexity results. *Theor. Comput. Sci.* **2007**, *371*, 72–82. [[CrossRef](#)]
10. Head, T.; Păun, G.; Pixton, D. Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. *Handb. Form. Lang.* **1997**, *2*, 295–360.
11. Head, T. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviours. *Bull. Math. Biol.* **1987**, *49*, 737–759. [[CrossRef](#)]
12. Csuha-Varjú, E.; Kari, L.; Păun, G. Test tube distributed systems based on splicing. *Comput. Artif. Intell.* **1996**, *15*, 211–232.
13. Păun, G. Distributed architectures in DNA computing based on splicing: Limiting the size of components. In Proceedings of the 1st International Conference on Unconventional Models of Computation, Auckland, New Zealand, 5–9 January 1998; pp. 323–335.
14. Păun, G. DNA computing; Distributed splicing systems. *Lect. Notes Comput. Sci.* **2005**, *1261*, 353–370.
15. Drăgoi, C.; Manea, F.; Mitrana, V. Accepting networks of evolutionary processors with filtered connections. *J. Univers. Comput. Sci.* **2007**, *13*, 1598–1614.
16. Bottoni, P.; Labella, A.; Manea, F.; Mitrana, V.; Petre, I.; Sempere, J.M. Complexity-preserving simulations among three variants of accepting networks of evolutionary processors. *Nat. Comput.* **2011**, *10*, 429–445. [[CrossRef](#)]
17. Rozenberg, G.; Salomaa, A. *Handbook of Formal Languages*; Springer: Berlin, Germany, 1997.
18. Arroyo, F.; Castellanos, J.; Mitrana, V.; Santos, E.; Sempere, J.M. Networks of bio-inspired processors. *Triangle Lang. Lit. Comput.* **2012**, *7*, 4–22.
19. Loos, R.; Manea, F.; Mitrana, V. On small, reduced, and fast universal accepting networks of splicing processors. *Theor. Comput. Sci.* **2009**, *410*, 406–416. [[CrossRef](#)]
20. Rogozhin, Y. Small universal Turing machines. *Theor. Comput. Sci.* **1996**, *168*, 215–240. [[CrossRef](#)]
21. Minsky, M. Size and structure of universal Turing machines using tag systems. *Recursive Funct. Theory Symp. Pure Math.* **1962**, *5*, 229–238.
22. Post, E.L. Formal reductions of the general combinatorial decision problem. *Am. J. Math.* **1943**, *65*, 197–215. [[CrossRef](#)]
23. Woods, D.; Neary, T. On the time complexity of 2-tag systems and small universal Turing machines. In Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science FOCS'06, Berkeley, CA, USA, 21–24 October 2006; pp. 439–448.
24. Cocke, J.; Minsky, M. Universality of tag systems with $p = 2$. *J. ACM* **1964**, *11*, 5–20. [[CrossRef](#)]

25. Gómez-Canaval, S.; Ordozgoiti, B.; Mozo, A. NPEPE: Massive natural computing engine for optimally solving NP-complete problems in Big Data scenarios. *Commun. Comput. Inf. Sci.* **2015**, *539*, 207–217.
26. Gómez-Canaval, S.; Mitrană, V.; Păun, M.; Vakaruk, S. Ultra-Scalable Simulations of Networks of Polarized Evolutionary Processors. In Proceedings of the 3rd International Conference on Advances in Artificial Intelligence (ICAAI), Istanbul, Turkey, 22–24 October 2019; pp. 73–81.
27. Apache Software Foundation. Apache Giraph. 2014. Available online: <http://giraph.apache.org/> (accessed on 23 March 2020).
28. Valiant, L. A bridging model for parallel computation. *Commun. ACM* **1990**, *33*, 103–111. [[CrossRef](#)]
29. Apache Software Foundation. Apache Spark. Available online: <http://spark.apache.org/> (accessed on 23 March 2020).
30. Reyes-Ortiz, J.; Oneto, L.; Anguita, D. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Comput. Sci.* **2015**, *53*, 121–130. [[CrossRef](#)]
31. Saracevic, M.; Selimi, A. Convex polygon triangulation based on Ballot problem and Planted Trivalent Binary Tree. *Turk. J. Electr. Eng. Comput. Sci.* **2019**, *27*, 346–361. [[CrossRef](#)]
32. Stanimirovic, P.; Krtolica, P.; Saracevic, M.; Masovic, S. Decomposition of Catalan numbers and convex polygon triangulations. *Int. J. Comput. Math.* **2014**, *91*, 1315–1328. [[CrossRef](#)]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).