processes

MDPI

Article

Simulation of Spiking Neural P Systems with Sparse Matrix-Vector Operations

Miguel Ángel Martínez-del-Amor [1,2,*], David Orellana-Martín [1], Ignacio Pérez-Hurtado [1], Francis George C. Cabarle [3] and Henry N. Adorna [3]

[1] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, 41012 Seville, Spain; dorellana@us.es (D.O.-M.); perezh@us.es (I.P.-H.)
[2] Smart Computer Systems Research and Engineering Lab (SCORE), Research Institute of Computer Engineering (I3US), Universidad de Sevilla, 41012 Seville, Spain
[3] Algorithms and Complexity Laboratory, Department of Computer Science, University of the Philippines Diliman, Quezon City 1101, Philippines; fccabarle@up.edu.ph (F.G.C.C.); hnadorna@up.edu.ph (H.N.A.)
* Correspondence: mdelamor@us.es

Abstract: To date, parallel simulation algorithms for spiking neural P (SNP) systems are based on a matrix representation. This way, the simulation is implemented with linear algebra operations, which can be easily parallelized on high performance computing platforms such as GPUs. Although it has been convenient for the first generation of GPU-based simulators, such as CuSNP, there are some bottlenecks to sort out. For example, the proposed matrix representations of SNP systems lead to very sparse matrices, where the majority of values are zero. It is known that sparse matrices can compromise the performance of algorithms since they involve a waste of memory and time. This problem has been extensively studied in the literature of parallel computing. In this paper, we analyze some of these ideas and apply them to represent some variants of SNP systems. We also provide a new simulation algorithm based on a novel compressed representation for sparse matrices. We also conclude which SNP system variant better suits our new compressed matrix representation.

Keywords: spiking neural P systems; simulation algorithm; sparse matrix-vector operations; compressed matrix representation; GPU computing

Citation: Martínez-del-Amor, M.Á.; Orellana-Martín, D.; Pérez-Hurtado, I.; Cabarle, F.G.C.; Adorna, H.N. Simulation of Spiking Neural P Systems with Sparse Matrix-Vector Operations. *Processes* **2021**, *9*, 690. https://doi.org/10.3390/pr9040690

Academic Editors: Mengchu Zhou and Zhiwei Gao

Received: 28 February 2021
Accepted: 12 April 2021
Published: 14 April 2021

1. Introduction

Membrane computing [1,2] is an interdisciplinary research area in the intersection of computer science and cellular biology mainly [3], but also with many other fields such as engineering, neuroscience, systems biology, chemistry, etc. The aim is to study computational devices called P systems, taking inspiration from how living cells process information. Spiking neural P (SNP) systems [4] are a type of P system composed of a directed graph inspired by how neurons are interconnected by axons and synapses in the brain. Neurons communicate through spikes, and the time difference between them plays an important role in the computation. Therefore, this model belongs to the known third generation of artificial neural networks, i.e., based on spikes.

Aside from computing numbers, SNP systems can also compute strings, and hence, languages. More general ways to provide the input or receive the output include the use of spike trains, i.e., a stream or sequence of spikes entering or leaving the system. Further results and details on computability, complexity, and applications of spiking neural P systems are detailed in [5–7], a dedicated chapter in the Handbook in [8], and an extensive bibliography until February 2016 in [9]. Moreover, there is a wide range of SNP system variants: with delays, with weights [10], with astrocytes [11], with anti-spikes [12], dendrites [13], rules on synapses [14], scheduled synapses [15], stochastic firing [16], numerical [17], etc.

The research on applications and variants of SNP systems has required the development of simulators. The simulation of SNP systems was initially carried out through sequential simulators such as pLinguaCore [18]. In 2010, a matrix representation of SNP systems was introduced [19]. Since then, most simulation algorithms are based on matrices and vector representations, and consists of a set of linear algebra operations. This way, parallel simulators can be efficiently implemented, since matrix-vector multiplications are easy to parallelize. Moreover, there are efficient algebra libraries that can be used out-of-the-box, although they have not been explored yet for this purpose. For instance, GPUs are parallel devices optimized for certain matrix operations [20], and can handle matrix operations efficiently. We can say, without loss of generality, that these matrix representations of SNP systems fit well to the highly parallel architecture of these devices. This have been harnessed already by introducing CuSNP, a set of simulators for SNP systems implemented with CUDA [21–24]. Simulators for specific solutions have been also defined in the literature [5,25]. Moreover, this is not unique for SNP systems, many simulators for other P system variants have been accelerated on GPUs [26–28].

However, this matrix representation can be sparse (i.e., having a majority of zero values) because the directed graph of SNP systems is not usually fully connected. A first approach to tackle this problem was presented in [29], where some of the ideas described in this work were described. Following these ideas, in [30], the transition matrix was split to reduce the memory footprint of the SNP representation. In many disciplines, sparse vector-matrix operations are very usual, and hence, many solutions based on compressed implementations have been proposed in the literature [31].

In this paper, we introduce compressed representations for the simulation of SNP systems based on sparse vector-matrix operations. First, we provide two approaches to compress the transition matrix for the simulation of SNP systems with static graph. Second, we extend these algorithms and data structures for SNP systems with dynamic graphs (division, budding, and plasticity). Finally, we make a complexity analysis and comparison of the algorithms to draw some conclusions.

The paper is structured as follows: Section 2 provides required concepts for the methods and algorithms here defined; Section 3 defines the designs of the representations; Section 4 contains the detailed algorithms based on the compressed representations; Section 5 shows the results on complexity analyses of the algorithms; Section 6 provides final conclusions, remarks, and plans of future work.

2. Preliminaries

In this section we briefly introduce the concepts employed in this work. Firstly, we define the standard model of spiking neural P systems and three variants. Second, a matrix-based simulation algorithm for this model is revisited. Third, the fundamentals of compressed formats for sparse matrix-vector operations are given.

2.1. Spiking Neural P Systems

Let us first formally introduce the definition of spiking neural P system. This model was first introduced in [4].

Definition 1. *A* spiking neural P system *of degree* $q \geq 1$ *is a tuple*

$$\Pi = (O, syn, \sigma_1, \ldots, \sigma_q, i_{out})$$

where:

- $O = \{a\}$ *is the singleton alphabet (a is called spike);*
- $syn = \{(i,j)|, 1 \leq i, j \leq q, i \neq j\}$ *represents the arcs of a directed graph* $G = (V, syn)$ *whose nodes are* $V = \{1, \ldots, q\}$*;*
- $\sigma_1, \ldots, \sigma_q$ *are neurons of the form*

$$\sigma_i = (n_i, R_i), 1 \leq i \leq q,$$

where:

- $n_i \geq 0$ *is the initial number of spikes within neuron labeled by* i*; and*
- R_i *is a finite set of rules associated to neuron labeled by* i*, of one of the following forms:*
  - (1) $E/a^c \to a^p$*, being* E *a regular expression over* $\{a\}$*,* $c \geq p \geq 1$ *(firing rules);*
  - (2) $a^s \to \lambda$ *for some* $s \geq 1$*, with the restriction that for each rule* $E/a^c \to a^p$ *of type of type* (1) *from* R_i*, we have* $a^s \notin L(E)$ *(forgetting rules).*

- $i_{out} \in \{1, 2, \ldots, q\}$ *such that* $(i_{out}, j) \notin syn$*, for any* $1 \leq j \leq q$.

A spiking neural P system of degree $q \geq 1$ can be viewed as a set of q neurons $\{\sigma_1, \ldots, \sigma_q\}$ interconnected by the arcs of a directed graph *syn*, called *synapse graph*. There is a distinguished neuron label i_{out}, called output neuron ($\sigma_{i_{out}}$), which communicates with the environment.

If a neuron σ_i contains k spikes at an instant t, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a^p$ can be applied. By the application of that rule, c spikes are removed from neuron σ_i and the neuron fires producing p spikes immediately. Thus, each neuron σ_j such that $(\sigma_i, \sigma_j) \in G$ receives p spikes. For $\sigma_{i_{out}}$, the output neuron i_{out}, the spikes are sent to the environment.

The rules of type (2) are *forgetting* rules, and they are applied as follows: If neuron σ_i contains exactly s spikes, then the rule $a^s \to \lambda$ from R_i can be applied. By the application of this rule all s spikes are removed from σ_i.

In spiking neural P systems, a global clock is assumed, marking the time for the whole system. Only one rule can be executed in each neuron at step t. As models of computation, spiking neural P systems are *Turing complete*, i.e., as powerful as Turing machines. On one hand, a common way to introduce the input (instance of the problem to solve) to the system is to encode it into some or all of the initial spikes n_i's (inside each neuron i). On the other hand, a common way to obtain the output is by observing neuron i_{out}: either by getting the interval $t_2 - t_1 = n$, where $\sigma_{i_{out}}$ sent its first two spikes at times t_1 and t_2 (we say n is computed or generated by the system), or by counting all the spikes sent by $\sigma_{i_{out}}$ to the environment until the system halts.

For the rest of the paper, we call this model spiking neural P systems with static structure, or just static SNP, given that the graph associated with it does not change along the computation. Next, we briefly introduce and focus on three variants with a dynamic graph: division, budding, and plasticity. A broader explanation of them and more variants are provided at [32–35].

Finally, let us introduce some notations and definitions:

- $pres(i)$: for a neuron σ_i, the *presynapses* of this neuron is $pres(i) = \{j | (i, j) \in syn\}$.
- $outdegree(i)$: for a neuron σ_i, the *out degree* of this neuron is: $outdegree(i) = |pres(i)|$.
- $ines(i)$: for a neuron σ_i, the *insynapses* of this neuron is $ines(i) = \{j | (j, i) \in syn\}$.
- $indegree(i)$: for a neuron σ_i, the *in degree* of this neuron is: $outdegree(i) = |ines(i)|$.

2.1.1. Spiking Neural P Systems with Budding Rules

Based on the idea of neuronal budding, where a cell is divided into two new cells, we can abstract it to budding rules. In this process, the new neurons can differ in some aspects: their connections, contents, and shape. A budding rule has the following form:

$$[E]_i \to [\,]_i / [\,]_j,$$

where E is a regular expression and $i, j \in \{1, \ldots, q\}$.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, and there is no neuron σ_j such that there exists a synapse (i, j) in the system, then the rule $[E]_i \to [\,]_i / [\,]_j$ is enabled and it can be executed. A new neuron σ_j is created, and both neurons σ_i and σ_j are empty after the execution of the rule. This neuron σ_i keeps all the synapses that were going in, and this σ_j inherits all the synapses that were going out of σ_i in the previous configuration. There is

also a synapse (i, j) between neurons σ_i and σ_j, and the rest of synapses of σ_j are given to the neuron depending on the synapses of *syn*.

2.1.2. Spiking Neural P Systems with Division Rules

Inspired by the process of *mitosis*, division rules have been widely used within the field of *membrane computing*. In SNP systems, a division rule can be defined as follows:

$$[E]_i \rightarrow [\,]_j | | [\,]_k,$$

where E is a regular expression and $i, j, k \in \{1, \ldots, q\}$.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, and there is no neuron σ_g such that the synapse (g, i) or (i, g) exists in the system, $g \in \{j, k\}$, then the rule $[E]_i \rightarrow [\,]_j | | [\,]_k$ is enabled and it can be executed. Neuron σ_i is then divided into two new cells, σ_j and σ_k. The new cells are empty at the time of their creation. The new neurons keep the synapses previously associated to the original neuron σ_i, that is, if there was a synapse from σ_g to σ_i, then a new synapse from σ_g to σ_j and a new one to σ_k are created, and if there was a synapse from σ_i to σ_g, then a new synapse from σ_j to σ_g and a new one from σ_k to σ_g are created. The rest of synapses of σ_j and σ_k are given by the ones defined in *syn*.

2.1.3. Spiking Neural P Systems with Plasticity Rules

It is known that new synapses can be created in the brain thanks to the process of *synaptogenesis*. We can recreate this process in the framework of spiking neural P systems defining *plasticity rules* in the following form:

$$E/a^c \rightarrow \alpha k(i, N_j),$$

where E is a regular expression, $c \geq 1$, $\alpha \in \{+, -, \pm, \mp\}$, $k \geq 1$ and $N_j \subseteq \{1, \ldots, q\}$ (a.k.a. neuron set). Recall that $pres(i)$ is the set of presynapses of neuron σ_i.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, then the rule $E/a^c \rightarrow \alpha k(i, N_j)$ is enabled and can be executed. The rule consumes c spikes and, depending on the value of α, it performs one of the following:

- If $\alpha = +$ and $N_j - pres(i) = \emptyset$, or if $\alpha = -$ and $pres(i) = \emptyset$, then there is nothing more to do.
- If $\alpha = +$ and $|N_j - pres(i)| \leq k$, deterministically create a synapse to every σ_g, $g \in N_j - pres(i)$. Otherwise, if $|N_j - pres(i)| > k$, then non-deterministically select k neurons in $N_j - pres(i)$ and create one synapse to each selected neuron.
- If $\alpha = -$ and $|pres(i)| \leq k$, deterministically delete all synapses in $pres(i)$. Otherwise, if $|pres(i)| > k$, then non-deterministically select k neurons in $pres(i)$ and delete each synapse to the selected neurons.
- If $\alpha = \{\pm, \mp\}$, create (respectively, delete) synapses at time t and then delete (resp., create) synapses at time $t + 1$. Even when this rule is applied, neurons are still open, that is, they can continue receiving spikes.

Let us notice that if, for some σ_i, we apply a *plasticity rule* with $\alpha \in \{+, \pm, \mp\}$, when a synapse is created, a spike is sent from σ_i to the neuron that has been connected. That is, when σ_i attaches to σ_j through this method, we have immediately transferring one spike to σ_j.

2.2. Matrix Representation for SNP Systems

Usually, parallel, P system simulators make use of *ad-hoc* representations, tailored for a certain variant [26–28]. In order to ease the simulation of static SNP system and its deployment to parallel environments, a matrix representation was introduced [19]. By using a set of algebraic operations, it is possible to reproduce the transitions of a computation. Although the baseline representation only involves SNP systems without delays and static structure, many extensions have followed such as for enabling delays [21,22],

handling non-determinism [24], plasticity rules [36], rules on synapses [37], and dendrite P systems [38]. In this section we briefly introduce the definitions for the matrix representation of the basic model of spiking neural P systems without delays, as defined above. We also provide the pseudocode to simulate just one computation of any P system of the variant using this matrix representation. In our notation, we use capital letters for vectors and matrices, and [] for accessing values: $V[i]$ is the value at position i of the vector V, and $M[i,j]$ is the value at row i and column j of matrix M.

For an SNP system Π of degree (q,m) (q neurons and m rules, where $m = \sum_{i=1}^{q} |R_i|$), we define the following vectors and matrices:

Configuration vector: C_k is the vector containing all spikes in every neuron on the kth computation step/time, where C_0 denotes the initial configuration; i.e., $C_0[i] = n_i$, for neuron $\sigma_i = (n_i, R_i)$. It contains q elements.

Spiking vector: S_k shows if a rule is going to fire at the transition step k (having value *1*) or not (having value *0*). Given the non-determinism nature of SNP systems, it would be possible to have a set of valid spiking vectors, which is denoted as SV_k. However, for the computation of the next configuration vector, only a spiking vector is used. It contains m elements.

Transition matrix: M_Π is a matrix comprised of $m \cdot q$ elements given as

$$M[i,j] = \begin{cases} -c, & \text{rule } r_i \text{ is in } \sigma_j \text{ and is applied consuming } c \text{ spikes;} \\ p, & \text{rule } r_i \text{ is in } \sigma_s, \text{ with } s \neq j \text{ and } (s,j) \in syn, \\ & \text{and is applied producing } p \text{ spikes;} \\ 0, & \text{rule } r_i \text{ is in } \sigma_s \text{ with } s \neq j \text{ and } (s,j) \notin syn. \end{cases}$$

In this representation, rows represent rules and columns represent neurons in the spiking transition matrix. Note also that a negative entry corresponds to the consumption of spikes. Thus, it is easy to observe that each row has exactly one negative entry, and each column has at least one negative entry [19].

Hence, to compute the transition k, it is enough to select a spiking vector S_k from all possibilities SV_k and calculate: $C_k = S_k \cdot M_\Pi + C_{k-1}$.

The pseudocode to simulate a computation of an SNP system is as described in Algorithm 1. The selection of valid spiking vectors can be done in different ways, as in [21,22]. This returns a set of valid spiking vectors. In this work, we focus on just one computation, but non-determinism can be tackled by maintaining a queue of generated configurations [24].

Algorithm 1 MAIN PROCEDURE: simulating one computation for static spiking neural P systems.

Require: A SNP system Π of degree (q,m), and a limit L of time steps.
Ensure: A computation of the system
1: $(M_\Pi, C_0) \leftarrow$ INIT(Π)
2: $k \leftarrow 0$
3: **repeat**
4: $SV_k \leftarrow$ SPIKING_VECTORS(C_k, Π) ▷ Calculate all possible spiking vectors
5: $S_k \leftarrow$ GET_ONE_RANDOMLY(SV_k) ▷ Pick one spiking vector randomly
6: $C_{k+1} \leftarrow C_k + S_k \cdot M_\Pi$ ▷ Compute next configuration
7: $k \leftarrow k+1$
8: **until** $k \geq L \vee SV_k = \emptyset$ ▷ Stop condition: maximum steps or no more applicable rules
9: **return** $C_0 \dots C_{k-1}$

In this work we focus on compressing the representation, specifically the transition matrix, so the determination of the spiking vector is not affecting these designs. Therefore, we use a straightforward approach and select just one valid spiking vector randomly. The representations here depicted only affect how the computation of the next configuration is done (matrix-vector multiplication at line 6 in Algorithm 1).

2.3. Sparse Matrix-Vector Operations

Algebraic operations have been studied deeply in parallel computing solutions. Specifically, GPU computing provides large speedups when accelerating such kind of operations. This technology allows us to run scientific computations in parallel on the GPU, given that a GPU device typically contains thousands of cores and high memory bandwidth [39]. However, parallel computing on a GPU has more constraints than on a CPU: threads have to run in an SPMD fashion while accessing data in a coalesced way; that is, best performance is achieved when the execution of threads is synchronized and accessing contiguous data from memory. In fact, GPUs have been employed for P system simulations since the introduction of CUDA.

Matrix computation is a highly optimized operation in CUDA [40], and there are many efficient libraries for algebra computations like cuBLAS. It is usual that when working with large matrices, these are almost "empty", or with a majority of zero values. This is known as *sparse matrix*, and this downgrades the runtime in two ways: lot of memory is wasted, and lot of operations are redundant.

Given the importance of linear algebra in many computational disciplines, sparse vector-matrix operations (SpMV) have been subject of study in parallel computing (and so, on GPUs). Today there exists many approaches to tackle this problem [41]. Let us focus on two formats to represent sparse matrices in a compressed way, assuming that threads access rows in parallel:

- *CSR* format. Only non-null values are represented by using three arrays: row pointers, non-zero values, and columns (see Figure 1 for an example). First, the row-pointers array is accessed, which contains a position per row of the original matrix. Each position says the index where the row start in the non-zero values and columns arrays. The non-zero values and the columns arrays can be seen as a single array of pairs, since every entry has to be accessed at the same time. Once a row is indexed, then a loop over the values in that row has to be performed, so that the corresponding column is found, and therefore, the value. If the column is not present, then the value is assumed to be zero, since this data structure contains all non-zero values. The main advantage is that it is a full-compressed format if $NumNonZeroValues \cdot 2 > NumZeroValues$, where $NumNonZeroValues$ and $NumZeroValues$ are the number of non-zero and zero values in the original matrix, respectively. However, the drawbacks is that the search of elements in the non-zero values and columns arrays is not coalesced when using parallelism per row. Moreover, since it is a full-compressed format, there is no room for modifying the values, such as introducing new non-zero values;
- *ELL* format. This representation aims at increasing the memory coalescing access of threads in CUDA. This is achieved by using a matrix of pairs, containing a trimmed, transposed version of the original matrix (see Figure 2 for an example). Each column of the ELL matrix is devoted to each row of the matrix, even though the row is empty (all elements are zero). Every element is a pair, where the first position denotes the column and the second is the value, of only the non-zero elements in the corresponding row. However, the size of the matrix is fixed, so the number of columns equals the number of rows of the original matrix, but the number of rows is the maximum length of a row in terms of non-zero values; in other words, the maximum amount of non-zero elements in a row of the original matrix. Rows containing fewer elements pad the difference with null elements. The main advantage of this format is that threads always access the elements of all rows in coalesced way, and the null elements padded by short rows can be utilized to incorporate new data. However, there is a waste of memory, which is worst when the rows are unbalanced in terms of number of zeros.

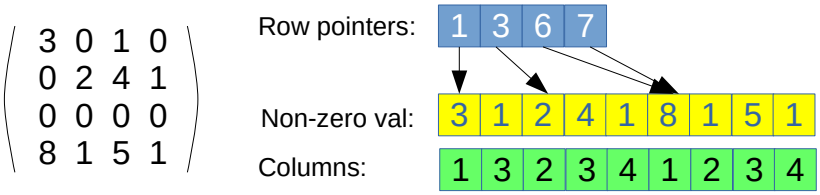

Figure 1. CSR format example. Non-zero val array stores the non-null values, columns array stores the column indexes, and row pointers are the positions where each row starts in the previous arrays.

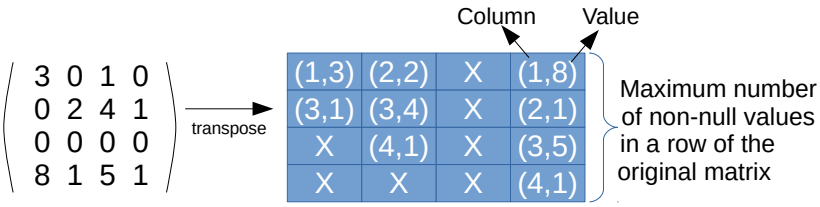

Figure 2. ELL format example. Note that it represents the transpose of the original matrix to increase the coalesced access in GPU devices. It includes pairs of column and value for every row. The compressed matrix has a number of columns equals to the number of original rows, and a number of rows equals to the maximum amount of non-null values in an original row.

3. Methods

SNP systems in the literature typically do not contain fully connected graphs. This means that the transition matrix gets very sparse and, therefore, both computing time and memory are wasted. However, further optimizations based on SpMV can be conveyed. In the following subsections we discuss some approaches. Of course, if the graph inherent to an SNP system leads to a compressed transition matrix, then a normal (sparse) format can be employed, because using compressed formats will increase the memory footprint.

In this work, we focus on the basic model of spiking neural P systems without delays as defined above, as well as three variants with dynamic network: budding, division and plasticity. The set of algorithms defined next are designed to take advantage of data parallelism, what is convenient for GPUs and vector co-processors. Their pseudocodes are detailed in Section 4 of this paper.

In Algorithm 2 we generalize the pseudocode disposed in Algorithm 1 to be able to handle both static and dynamic networks. This way, each variant requires to re-define just some functions from the ones defined for the static SNP system variant using vector and matrices without compression (i.e., sparse representation). In order to understand the algorithms, we will present in this section the main new data structures and their behavior. The detailed pseudocodes are available in Section 4.

Algorithm 2 MAIN PROCEDURE: simulating one computation for spiking neural P systems.

Require: An SNP system Π of degree (q, m), and a limit L of time steps.
Ensure: A computation of the system
1: $(C_0, M_0) \leftarrow \text{INIT}(\Pi)$
2: $k \leftarrow 0$
3: **repeat**
4: $SV_k \leftarrow \text{SPIKING\_VECTORS}(C_k, M_k)$ ▷ Calculate all possible spiking vectors
5: $S_k \leftarrow \text{GET\_ONE\_RANDOMLY}(SV_k)$ ▷ Pick one spiking vector randomly
6: $(C_{k+1}, M_{k+1}) \leftarrow \text{COMPUTE\_NEXT}(C_k, M_k, S_K)$ ▷ Compute next configuration
7: $k \leftarrow k + 1$
8: **until** $k \geq L \vee SV_k = \emptyset$ ▷ Stop condition: maximum steps or no more applicable rules
9: **return** $C_0 \dots C_{k-1}$

As a convention, those vectors and matrices using subindex k are dynamic and can change during the simulation time, while those with Π subindex are constructed at the

beginning and are invariant. Capital letters refer to vectors and matrices, and small letters are scalar numbers. A total order over the rules defined in the system is assumed, which is denoted as $R = \bigwedge_{i=1}^{q} R_i$. For the sake of simplicity, we represent each rule $r_j \equiv E_j/a^{c_j} \rightarrow a^{p_j}$, with $1 \leq j \leq m$, as a tuple (i, E_j, c_j, p_j), where i is the subindex of the set R_i where r_j belongs (i.e., the neuron where it is contained). Specifically, forgetting rules just have $p_j = 0$.

For static SNP systems using sparse representation, we use the following vectors and matrices:

- **Preconditions vector** P_Π is a vector storing the preconditions of the rules; that is, both the regular expression and the consumed spikes. Initially, $P_\Pi[j] = (E_j, c_j)$, for each $r_j \in R$, $r_j = (i_j, E_j, c_j, p_j)$, $1 \leq j \leq m$.
- **Neuron-rule map vector** N_Π is a vector that maps each neuron index with its rules indexes. Specifically, $N_\Pi[i]$ is the index of the first rule in the neuron. Given that rules have been ordered in R as mentioned above, rules belonging to the same neuron have contiguous indexes. Thus, it is enough to store just the first index. In this sense, the first rule in neuron i is $N_\Pi[i]$ and the last one is $N_\Pi[i+1] - 1$. In other words, N_Π contains $q + 1$ elements, and it is initialized as follows: $N_\Pi[i] = \sum_{h=1}^{i-1} 1 + |R_h|$. Specifically, $N_\Pi[1] = 1$ and $N_\Pi[i] = N_0[i-1] + |R_{i-1}|$, for $2 \leq i \leq q + 1$
- M_k is the **transition tuple**, where $M_k = (M_\Pi, P_\Pi, N_\Pi)$. If the variant has a dynamic network, the transition matrix needs to be modified. Therefore, we start with M_0. The following algorithms show how they are constructed.

Algorithm 2 can be easily transformed to Algorithm 1 by defining the INIT and COMPUTE_NEXT functions as in Algorithm 3. They work exactly as already specified in Section 2.2; that is, using the usual vector-matrix multiplication operation to calculate the next configuration vector. We will also detail how the selection of spiking vectors can be done. This is defined in Algorithm 4, and it is based on previous ideas already presented in [21,22]. First, SPIKING_VECTORS function calculates the set of all possible spiking vectors by using a recursive function over neuron index i. It gathers all spiking vectors that can be generated for neurons $i' > i$ and then. If neuron i contains applicable rules, it populates a spiking vector for each of these rules, and from each of the generated spiking vectors form neurons $i' > i$. Finally, neuron i propagates these spiking vectors to the next neuron $i - 1$.

3.1. Approach with ELL Format

Our first approach to compress the representation of the transition matrix, M_Π, is to use the ELL format (see Figure 3 for an example). The reason for using ELL and not other compressed formats for sparse matrices (CSR, COO, BSR, ...) is to enable extensions for dynamic networks, as seen later. ELL can give some room for modifications without much memory re-allocations, while CSR requires us to modify the whole matrix to add new elements.

ELL format leads to the new compressed matrix M_Π^s. The following aspects have been taken into consideration:

- The ELL format represents the transpose of the original matrix, so now rows correspond to neurons and columns to rules. This is convenient for SIMD processors such as GPUs.
- The number of rows of M_Π^s equals the maximum amount of non-zero values in a row of M_Π, denoted by z'. It can be shown that $z' = z + 1$, where z is the maximum output degree found in the neurons of the SNP system. Specifically, $z = max\{outdegree(i) | 1 \leq i \leq q\}$ (see definition in Section 2.1). z' can be derived from the composition of the transition matrix, where row j devoted for rule $r_j \equiv (i_j, E_j, c_j, p_j)$ contains the values $+p_j$ for every neuron i (columns) connected though an output synapse with the neuron where the rule belongs to (i.e., $i \in pres(i_j)$), and a value $-c_j$ for consuming the spikes in the neuron the rule belongs to (i.e., i_j).

- The values inside columns can be sorted, so that the consumption of spikes ($-c$ values) are placed at the first row. In this way, if implemented in parallel, all threads can start by doing the same task: consuming spikes.
- Every position of M_Π^s is a pair (as illustrated in Figure 3), where the first element is a neuron label, and the second is the number of spikes produced ($+p$).

A parallel code can be implemented with this design by assigning a thread to each rule, and so, one per column of the spiking vector S_k and one per column of M_Π^s (rows of the original transition matrix). For the vector-matrix multiplication, it is enough to have a loop of z' steps at maximum through the columns. In the loop of each column j, the corresponding value in the spiking vector $S_k[j]$ (either 0 or 1) is multiplied to the value x_j in the pair (i_j, x_j), and added to the neuron id n_j in the configuration vector $C_k[n_j]$. In case the SNP network contains hubs (nodes with high amount of input synapses), then we can opt for a parallel reduction per column. Since some threads might write to same positions in the configuration vector at the same time, a solution would be to use atomic adding operations, which are available on devices such as GPUs.

Algorithm 3 Functions for static SNP systems with sparse matrix representation.

procedure INIT(Π)
  $(C_0, N_\Pi) \leftarrow$ INIT_NEURON_VECTORS(Π) ▷ Initialize vectors only related to neurons.
  $(M_\Pi, P_\Pi) \leftarrow$ INIT_RULE_MATRICES(Π) ▷ Initialize matrices related to rules.
  $M_0 \leftarrow (M_\Pi, P_\Pi, N_\Pi)$
  return (C_0, M_0)
end procedure

procedure INIT_NEURON_VECTORS(Π)
  $(q, \sigma_1, \ldots, \sigma_q) \leftarrow \Pi$ ▷ Get information from Π
  $C_0 \leftarrow$ EMPTY_VECTOR(q) ▷ Create initial configuration
  $N_\Pi \leftarrow$ EMPTY_VECTOR($q+1$) ▷ Create neuron-rule vector
  $N_\Pi[1] \leftarrow 1$
  for all $i \leftarrow 1 \ldots q$ **do** ▷ For each neuron
    $(n_i, R_i) \leftarrow \sigma_i$ ▷ Get info of the neuron from Π
    $C_0[i] \leftarrow n_i$ ▷ Initial configuration
    $N_\Pi[i+1] \leftarrow N_\Pi[i] + |R_i|$ ▷ Neuron-rule map vector initialization
  end for
  return (C_0, N_Π)
end procedure

procedure INIT_RULE_MATRICES(Π)
  $(R, m, q, pres) \leftarrow \Pi$ ▷ Get information from Π
  $P_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create preconditions vector
  $M_\Pi \leftarrow$ EMPTY_MATRIX(m, q) ▷ Create transition matrix
  for all $r_j \in R, j \leftarrow 1 \ldots m$ **do** ▷ For each rule (column). This loop is parallelizable.
    $r_j \equiv (i_j, E_j, c_j, p_j)$ ▷ Get info of the rule
    $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
    $M_\Pi[j, i_j] \leftarrow -c_j$ ▷ Construct transition matrix
    for all $i \in pres(i_j)$ **do** ▷ For each connected neuron to i_j
      $M_\Pi[j, i] \leftarrow p_j$ ▷ Construct transition matrix
    end for
  end for
  return (M_Π, P_Π)
end procedure

procedure COMPUTE_NEXT(C_k, M_k, S_K)
  $(M_\Pi, \_, \_) \leftarrow M_k$ ▷ Get some content of transition tuple
  return $(C_k + S_k \cdot M_\Pi, M_k)$ ▷ Only the configuration is updated.
end procedure

Algorithm 4 Spiking vectors selection with static SNP systems and sparse representation.

```
1:  procedure SPIKING_VECTORS(C_k, M_k)
2:      return COMBINATIONS(1, C_k, M_k)          ▷ Start calculating combinations from neuron 1
3:  end procedure

4:  procedure COMBINATIONS(i, C_k, M_k)
5:      q_k ← |C_k|                                ▷ With dynamic networks, q_k ≥ q
6:      (_, _, N_Π) ← M_k                          ▷ Get some content of transition tuple
7:      if i > q_k then                            ▷ If neuron i is out of index.
8:          return ∅                               ▷ An empty set.
9:      else
10:         SV ← ∅                                 ▷ The set for the rest of neurons.
11:         SV' ← COMBINATIONS(i + 1, C_k, P_Π, N_Π)   ▷ All combinations for rest of neurons.
12:         if SV' = ∅ then                        ▷ No spiking vectors yet for rest of neurons
13:             S ← EMPTY_VECTOR(m)                ▷ Create an empty spiking vector.
14:             SV'' ← {S}                         ▷ The set to loop over just contains S
15:         else                                   ▷ There are spiking vectors for rest of neurons
16:             SV'' ← SV'                         ▷ The set to loop over is just SV'
17:         end if
18:         for j ← N_Π[i] ... N_Π[i + 1] − 1 do   ▷ For each rule in neuron i
19:             if APPLICABLE(i, j, C_k, M_k) then ▷ If rule j is applicable
20:                 for all S ∈ SV'' do            ▷ For each spiking vector, either SV' or empty vector
21:                     S' ← S                     ▷ Create a copy
22:                     S'[j] ← 1                  ▷ Mark rule j as applicable
23:                     SV ← SV ∪ {S'}             ▷ Add it to the solution
24:                 end for
25:             end if
26:         end for
27:         if SV = ∅ then                         ▷ If there are no applicable rules
28:             return SV'                         ▷ Just propagate combinations
29:         else
30:             return SV                          ▷ Return calculated combinations
31:         end if
32:     end if
33: end procedure

34: procedure APPLICABLE(i, j, C_k, M_k)
35:     (_, P_Π, _) ← M_k                          ▷ Get some content of transition tuple
36:     (E_j, c_j) ← P_Π[j]                        ▷ Preconditions of the rule
37:     return C_k[i] ∈ L(E_j) ∧ C_k[i] ≥ c_j      ▷ If rule j is applicable in neuron i
38: end procedure

39: procedure GET_ONE_RANDOMLY(SV_k)
40:     s' ← RANDOM(1, |SV_k|))
41:     return s'-th spiking vector in SV_k        ▷ Returns just one randomly chosen
42: end procedure
```

In order to use this representation in Algorithm 2, we only need to re-define functions INIT_RULE_MATRICES and COMPUTE_NEXT from Algorithm 3 (for sparse representation) as shown in Algorithm 5. The rest of functions remain unchanged.

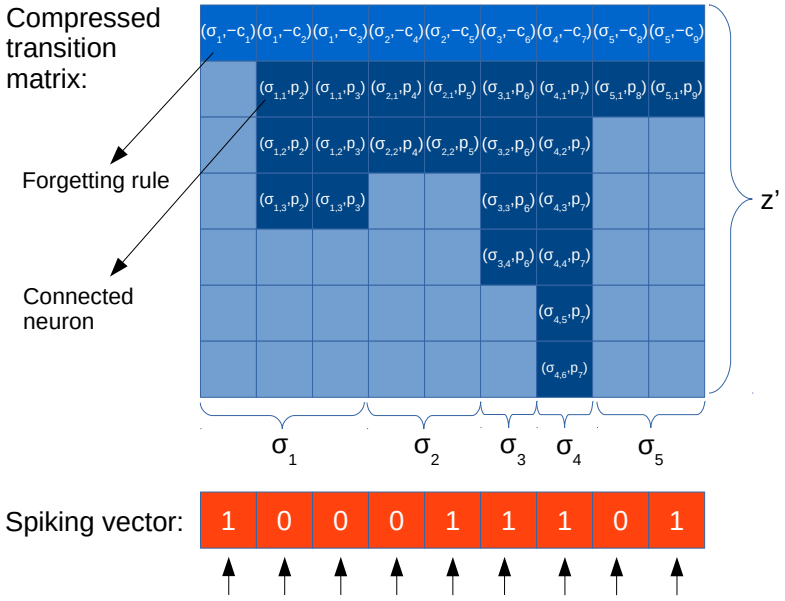

Figure 3. Illustration of compressed representation based on the ELL format of a static spiking neural P (SNP) system. Light cells are empty values (0,0). The first column is a forgetting rule (there is no need to use $p = 0$). The rows below the spiking vector illustrate threads, showing the level of parallelism that can be achieved; i.e., each column and each position of the spiking vector in parallel can be processed in parallel.

Algorithm 5 Functions for static SNP systems with ELL-based matrix representation.

procedure INIT_RULE_MATRICES(Π)
  $(R, m, q, z', pres) \leftarrow \Pi$ ▷ Get information from Π
  $P_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create preconditions vector
  $M^s_\Pi \leftarrow$ EMPTY_MATRIX(z', m) ▷ Create transition matrix
  for all $r_j \in R, j \leftarrow 1 \ldots m$ **do** ▷ For each rule (column). This loop is parallelizable.
    $r_j \equiv (i_j, E_j, c_j, p_j)$ ▷ Get info of the rule
    $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
    $M^s_\Pi[1, j] \leftarrow (i_j, -c_j)$ ▷ Only for the first row
    if $p_j > 0$ **then** ▷ Only if r_j is not a forgetting rule
      $k \leftarrow 2$ ▷ k is our iterator, compacting the rows
      for all $i \in pres(i_j)$ **do** ▷ For each out synapse
        $M^s_\Pi[k, j] \leftarrow (i, p_j)$ ▷ Add out neuron and produced spikes
        $k \leftarrow k + 1$ ▷ We know that $k \leq z'$
      end for
    end if
  end for
  return (M_Π, P_Π)
end procedure

procedure COMPUTE_NEXT(C_k, M_k, S_K)
  $C_{k+1} \leftarrow C_k$ ▷ Create a copy of C_k
  $(M^s_\Pi, \_, \_) \leftarrow M_k$ ▷ Get some content of transition tuple
  for $j \leftarrow 1 \ldots m$ **do** ▷ For each rule (column). This loop is parallelizable.
    $i \leftarrow 1$
    repeat
      $(i_j, x_j) \leftarrow M^s_\Pi[i, j]$ ▷ Get info from transition matrix
      $C_{k+1}[i_j] \leftarrow C_{k+1}[i_j] + S_k[j] \cdot x_j$ ▷ Update configuration, if not applicable, $S_k[j] = 0$
      $i \leftarrow i + 1$
    until $M^s_\Pi[i, j] = (0, 0) \vee i > z'$ ▷ Until reaching an empty value or the maximum
  end for
  return (C_{k+1}, M_k)
end procedure

3.2. Optimized Approach for Static Networks

If, in general, more than one rule are associated to each neuron, many of the iterations in the main loop in COMPUTE_NEXT function are wasted. Indeed, if the loop is parallelized and each iteration is assigned to a thread, then many of them will be inactive (having a 0 in the spiking vector), causing performance drops such as branch divergence and non-coalesced memory access in GPUs. Moreover, note in Figure 3 that columns corresponding to rules belonging to the same neuron contain redundant information: the generation of spikes ($+p$) is replicated for all synapses.

Therefore, a more efficient compressed matrix representation can be obtained when maintaining the synapses separated from the rule information. This is called **optimized matrix representation**, and can be done with the following data structures:

- **Rule vector**, Ru_Π. By using a CSR-like format (see Figure 4 for an example), rules of the form $E/a^c \rightarrow a^p$ (also forgetting rules are included, assuming $p = 0$) can be represented by an array storing the values c and p in a pair. We can use the already defined neuron-rule map vector N_k to relate the subset of rules associated to each neuron.
- **Synapse matrix**, Sy_Π. It is a transposed matrix as with ELL representation (to better fit to SIMD architectures such as GPU devices), but it has a column per neuron i and a row for every neuron j such that $(i, j) \in Syn$ (there is a synapse). That is, every element of the matrix corresponds to a synapse (the neuron id) or a null value otherwise. Null values are employed for padding the columns, since the number of rows equals z (the maximum output degree in the neurons of the SNP system). See Figure 4 for an example.
- **Spiking vector** is modified, containing only q positions instead of n (i.e., one per neuron), and states which rule is selected.

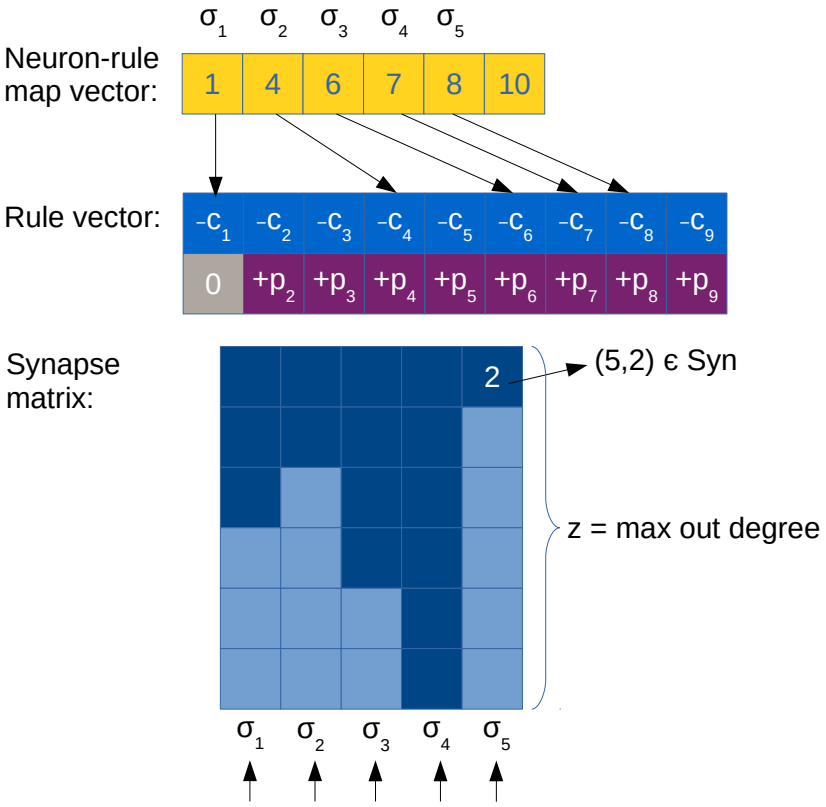

Figure 4. Illustration of optimized compressed matrix representation. Light cells in the synapse matrix are empty values (0), dark cells are positions with values greater than 0 (i.e., with neuron labels). The rows below illustrate threads, showing the level of parallelism that can be achieved (each column/neuron in parallel). The first column in the rule vector is a forgetting rule, where $p = 0$.

Note that we replace the transition matrix for a pair with rule vector and synapse matrix: $M'_\Pi = (Ru_\Pi, Sy_\Pi)$. In order to compute the next configuration, it is enough to loop over the neurons. Then, for each neuron i, we check which rule j is selected, according to the spiking vector at position $S_k[i]$. This is used to grab the pair (c_j, p_j) from the rule vector, and therefore consume c_j spikes in the neuron i and add p_j spikes in the neurons at the column i of the synapse matrix. The loop over the column can end prematurely if the out degree of neuron i is not z (that is, when encountering a null value). This operation can be easily parallelized by assigning a thread to each column of the synapse matrix (requiring q threads, one per neuron).

In order to use this optimized representation in Algorithm 2, we need to re-define the spiking selection function, since this vector works differently. To do this, it is enough to just modify two lines in the definition of COMBINATIONS function at Algorithm 4, in order to keep the spiking vector with size q and storing the rule id instead of just 1 or 0 (see Section 4 for more detail). Moreover, we need to define tailored INIT_RULE_MATRICES and COMPUTE_NEXT functions as shown in Algorithm 6, replacing those from Algorithm 3.

Algorithm 6 Functions for static SNP systems with optimized compressed matrix representation.

procedure INIT_RULE_MATRICES(Π)
  $(R, m, q, z, pres) \leftarrow \Pi$ ▷ Get information from Π
  $P_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create preconditions vector
  $Ru_\Pi \leftarrow$ EMPTY_VECTOR(m)
  for all $r_j \in R(j \leftarrow 1 \dots m)$ **do** ▷ For each rule (column). This loop is parallelizable.
    $r_j = (i_j, E_j, c_j, p_j)$ ▷ Get info of the rule
    $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
    $Ru_\Pi[j] \leftarrow (c_j, p_j)$ ▷ Store it in rule vector
  end for
  $Sy_\Pi \leftarrow$ EMPTY_MATRIX(z, q)
  for $i \leftarrow 1 \dots q$ **do** ▷ For each neuron (column in synapse matrix)
    $k \leftarrow 1$ ▷ k is our iterator, compacting the rows
    for all $h \in pres(i)$ **do** ▷ For each out synapse
      $Sy_\Pi[k, i] \leftarrow h$
      $k \leftarrow k + 1$ ▷ We know that $k \leq z$
    end for
  end for
  $M'_\Pi \leftarrow (Ru_\Pi, Sy_\Pi)$
  return (M'_Π, P_Π)
end procedure

procedure COMPUTE_NEXT(C_k, M_k, S_k)
  $C_{k+1} \leftarrow C_k$ ▷ Create a copy of C_k
  $(M'_\Pi, \_, \_) \leftarrow M_k$ ▷ Get some content of transition tuple
  $(Ru_\Pi, Sy_\Pi) \leftarrow M'_\Pi$
  for $i \leftarrow 1 \dots q$ **do** ▷ For each neuron. This loop is parallelizable.
    $j \leftarrow S_k[i]$ ▷ Index of rule to fire in the neuron
    if $j \neq 0$ **then** ▷ Only if there is a rule.
      $(c_j, p_j) \leftarrow Ru_\Pi[j]$ ▷ Get rule info
      $C_{k+1}[i] \leftarrow C_{k+1}[i] - c_j$ ▷ Consume spikes in firing neuron
      $w \leftarrow 1$ ▷ Next while stops if $p_j = 0$, i.e., a firing rule
      while $p_j > 0 \wedge Sy_\Pi[w, i] \neq 0 \wedge w \leq z$ **do** ▷ Until an empty value or the maximum
        $h \leftarrow Sy_\Pi[w, i]$ ▷ Get connected neuron by a synapse
        $C_{k+1}[h] \leftarrow C_{k+1}[h] + p_j$ ▷ Produce spikes in connected neuron
        $w \leftarrow w + 1$
      end while
    end if
  end for
  return (C_{k+1}, M_k)
end procedure

3.3. Optimized Approach for Dynamic Networks

The optimized compressed matrix representation discussed in Section 3.2 can be further extended to support rules that modify the network, such as budding, division, or plasticity.

3.3.1. Budding and Division Rules

We start by analyzing how to simulate dynamic SNP systems with budding and division rules. They are supported at the same time in order to unify the pseudocode and also because both kind of rules are usually together in the model.

First of all, the synapse matrix has to be flexible enough to host new neurons. This can be accomplished by allocating a matrix large enough to populate new neurons (probably up to fill the whole memory available). We denote q_{max} as the maximum amount of neurons that the simulator is able to support, and q_k the amount of neurons in a given step k. The formula to calculate q_{max} is in Section 5. It is important to point out that the simulator needs to differentiate between neuron label and neuron id [35]. The reason for this separation is that we can have more than one neuron (with different ids) with the same label (and hence, rules).

In order to achieve this separation, it is enough to have a vector to map each neuron id to its label. We will call this new vector, **neuron-id map vector** Q_k, and the following holds at step k: $q_k = |Q_k| = |C_k| = |S_k| \leq q_{max}$. That is, neuron-id map vector, configuration vector and spiking vector have a size of q_{max} as well. Once the label of a neuron is obtained, the information of its corresponding rules can be accessed as usual, like the neuron-rule map vector N_Π. For simplicity, we attach the neuron-id map vector to the transition tuple. Moreover, the synapse matrix becomes dynamic, thus using k sub-index: Sy_k; hence, the transition matrix is also dynamic. Let us now introduce this new notation for transition tuple and transition matrix:

- The transition matrix is now a dynamic pair: $M'_k = (Ru_\Pi, Sy_k)$.
- The transition tuple is extended as follows: $M_k = (M'_k, Q_k, P_\Pi, N_\Pi)$.

We use the following encoding for each type of rule. Spiking and forgetting rules remain unchanged:

- For a budding rule $r \equiv [E]_i \rightarrow []_i/[]_j$ as $r = (i, E, 0, j)$. Given that all pairs in the rule vector Ru_Π are of the form (c, p), and c is always greater equal than 1, then we can encode a budding rule as a pair $(0, j)$.
- For a division rule $r \equiv [E]_i \rightarrow []_j||[]_k$ as $r = (i, E, -j, k)$. Given that all pairs in the rule vector Ru_Π are of the form (c, p), and c is always greater equal than 1, then we can encode a division rule as a pair $(-j, k)$.

The execution of a budding rule $[E]_i \rightarrow []_i/[]_l$ requires the following operations (see Figure 5 for an illustration):

1. Let i' be the neuron id executing this rule.
2. Allocate a column l' to the synapse matrix Sy_k for the new neuron, and use this index as its neuron id.
3. Add an entry to the neuron-id map vector Q_k at position l' with the label l.
4. Copy column i' to the new column l' in Sy_k.
5. Delete the content of column i' and add only one element at the first row with the id l'.

For a division rule $[E]_i \rightarrow []_j||[]_l$, the following operations have to be performed (see Figure 6 for an example):

1. Let i' be the neuron id executing this rule.
2. Allocate a new column l' for the created neuron l in the synapse matrix Sy_k.
3. Modify the neuron-id map vector Q_k as follows: replace the value at position i' for label j, and add a new entry for l' to associate it with label k.
4. Copy column i' to l' in Sy_k (the generated neuron gets the out synapses of the parent).

5. Find all occurrences of i' in the synapse matrix, and add l' to the columns where it is found.

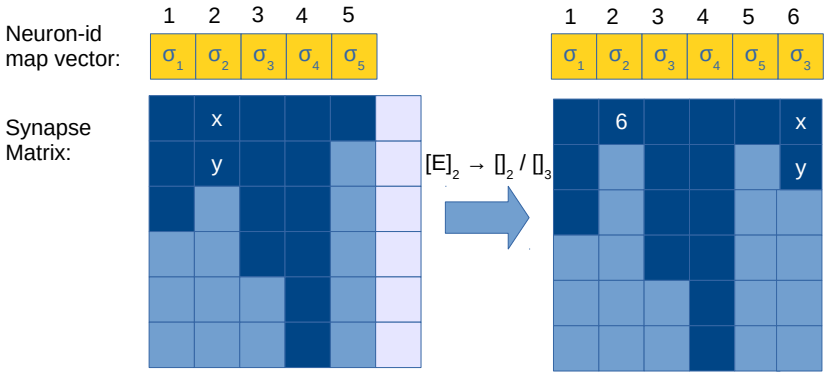

Figure 5. Illustration of application of a budding rule in the synapse matrix with compressed representation. Light blue cells in the synapse matrix are empty values (0), dark cells are positions with values greater than 0 (i.e., with neuron id), and light cells are empty columns allocated in memory (a total of q_{max}). Neuron 1 is applying budding, and its content is copied to an empty column (5) and replaced by a single synapse to the created neuron.

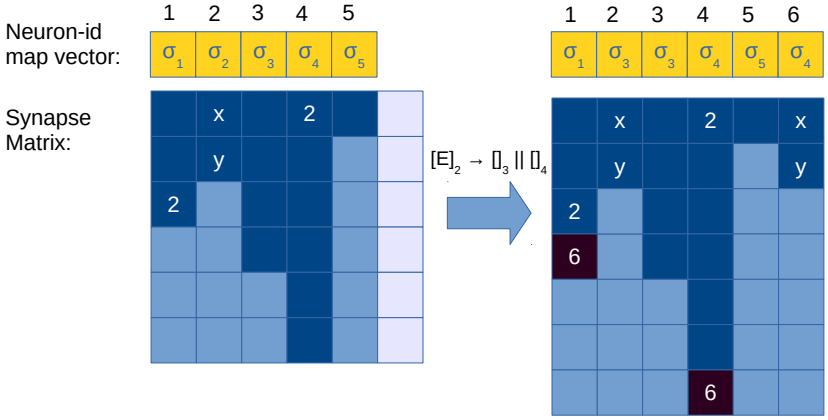

Figure 6. Illustration of application of a division rule in the synapse matrix with compressed representation. Light blue cells in the synapse matrix are empty values (0), dark cells are positions with values greater than 0 (i.e., with neuron id), and light cells are empty columns allocated in memory (a total of q_{max}). Neuron 1 is being divided, and its content is copied to an empty column (5). Columns 0 and 3 represent neurons with a synapse to the neuron being divided (1), so we need to update them as well with the synapse to the created neuron (5). Neuron 3 has reached its limit of maximum out degree, therefore we need to expand the matrix with a new row, or use a COO-like system to store these exceeded elements.

The last operation can be very expensive if the amount of neurons is large, since it requires to loop all over the synapse matrix. Moreover, when adding l' in all the columns containing i', it would be possible to exceed the predetermined size z. For this situation, a special array of overflows is needed, like ELL + COO format for SpMV [41]. For simplicity, we will assume this situation is weird and the algorithm will allocate a new row for the synapse matrix.

Some functions in the pseudocode are re-defined to support dynamic networks with division and budding:

- INIT functions as in Algorithm 7. They now take into account the initialization of structures at its maximum amount q_{max}, including the new neuron-id map vector.
- SPIKING_VECTORS function, as defined in Algorithm 4 and modified in Section 3.2

for optimized matrix representation, is slightly modified (just two lines) to support the neuron-id map vector.

- APPLICABLE function as in Algorithm 8. This function, when dealing with division rules, has to search if there are existing synapses for the neurons involved. If they exist, the division rule does not apply.
- COMPUTE_NEXT function as in Algorithm 9, to include the operations described above. It now needs to expand the synapse matrix Sy_k either by columns (when new neurons are created) or by rows if there is a neuron from which we need to create a synapse to the new neuron and it has already the maximum out degree z. In this case, we need to re-allocate the synapse matrix in order to extend it by one row (this is written in the pseudocode with the function EXPAND_MATRIX). Finally, let us remark that we can easily detect if type of a rule r_j at it's associated c_j value: if 0, it is a budding rule, if it is positive number, a spiking rule, otherwise (negative value) it is a division rule.

Algorithm 7 Initialization functions for dynamic SNP systems with budding and division rules over optimized compressed matrix representation.

procedure INIT(Π)
  $(C_0, N_\Pi) \leftarrow$ INIT_NEURON_VECTORS(Π) ▷ Initialize vectors only related to neurons.
  $(M'_0, Q_0, P_\Pi) \leftarrow$ INIT_RULE_MATRICES(Π) ▷ Initialize matrices related to rules.
  $M_0 \leftarrow (M'_0, Q_0, P_\Pi, N_\Pi)$
  return (C_0, M_0)
end procedure

procedure INIT_NEURON_VECTORS(Π)
  $(q, q_{max}, \sigma_1, \ldots, \sigma_q) \leftarrow \Pi$ ▷ Get information from Π
  $C_0 \leftarrow$ EMPTY_VECTOR(q_{max}) ▷ Create initial configuration
  $N_\Pi \leftarrow$ EMPTY_VECTOR($q+1$) ▷ Create neuron-rule vector
  $N_\Pi[1] \leftarrow 1$
  for all $i \leftarrow 1 \ldots q$ **do** ▷ For each neuron
    $(n_i, R_i) \leftarrow \sigma_i$ ▷ Get info of the neuron from Π
    $C_0[i] \leftarrow n_i$ ▷ Initial configuration
    $N_\Pi[i+1] \leftarrow N_\Pi[i] + |R_i|$ ▷ Neuron-rule map vector initialization
  end for
  return (C_0, N_Π)
end procedure

procedure INIT_RULE_MATRICES(Π)
  $(R, m, q, q_{max}, z, pres) \leftarrow \Pi$ ▷ Get information from Π
  $P_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create preconditions vector
  $Ru_\Pi \leftarrow$ EMPTY_VECTOR(m)
  for all $r_j \in R (j \leftarrow 1 \ldots m)$ **do** ▷ For each rule (column). This loop is parallelizable.
    $r_j \equiv (i_j, E_j, c_j, p_j)$ ▷ Get info of the rule
    $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
    $Ru_\Pi[j] \leftarrow (c_j, p_j)$ ▷ Store it in rule vector
  end for
  $Q_0 \leftarrow$ EMPTY_VECTOR(q_{max})
  $Sy_0 \leftarrow$ EMPTY_MATRIX(z, q_{max})
  for $i \leftarrow 1 \ldots q$ **do** ▷ For each neuron label
    $Q_0[i] \leftarrow i$
    $k \leftarrow 1$ ▷ k is our iterator, compacting the rows
    for all $h \in pres(i)$ **do** ▷ For each out synapse
      $Sy_0[k, i] \leftarrow h$
      $k \leftarrow k+1$ ▷ We know that $k \leq z$
    end for
  end for
  $M'_0 \leftarrow (Ru_\Pi, Sy_0)$
  return (M'_0, Q_0, P_Π)
end procedure

Algorithm 8 Applicable functions for dynamic SNP systems with budding rules over optimized compressed matrix representation.

procedure APPLICABLE(i, j, C_k, M_k)
  $(M'_k, Q_k, P_\Pi, \_) \leftarrow M_k$ ▷ Get some content of transition tuple
  $(Ru_\Pi, Sy_k) \leftarrow M'_k$ ▷ Get some content of transition matrix
  $(l_j, h_j) \leftarrow Ru_\Pi[j]$ ▷ Preconditions of the rule
  $(E_j, c_j) \leftarrow P_\Pi[j]$ ▷ Preconditions of the rule
  if $c_j > 0$ **then** ▷ If a spiking or forgetting rule
    return $C_k[i] \in L(E_j) \ \wedge \ C_k[i] \geq c_j$ ▷ If rule j is applicable in neuron $i c_j = 0$ ▷ If a budding rule
    $b \leftarrow False$ ▷ Check if synapse (i, h_j) exists
    $w \leftarrow 1$
    while $\neg b \wedge Sy_k[w, i] \neq 0 \wedge w \leq z$ **do** ▷ Until an empty value or the maximum
      $b \leftarrow Q_k[Sy_k[w, i]] = h_j$ ▷ If synapse exists
      $w \leftarrow w + 1$
    end while
    return $C_k[i] \in L(E_j) \ \wedge \ \neg b$
  else ▷ If a division rule
    $b \leftarrow False$ ▷ Check if synapse (i, h_j) or $(i, -l_j)$ exists
    $w \leftarrow 1$
    while $\neg b \wedge Sy_k[w, i] \neq 0 \wedge w \leq z$ **do** ▷ Until an empty value
      $b \leftarrow Q_k[Sy_k[w, i]] = h_j \vee Q_k[Sy_k[w, i]] = -l_j$ ▷ If either synapse exists
      $w \leftarrow w + 1$
    end while
    for $x \leftarrow 1 \ldots q_k$ **do** ▷ Search for neurons with label h_j or l_j
      $b' \leftarrow Q_k[x] = h_j \vee Q_k[x] = -l_j$ ▷ Either h_j or l_j
      $w \leftarrow 1$
      while $b' \wedge \neg b \wedge Sy_k[w, x] \neq 0 \wedge w \leq z$ **do** ▷ Combination of conditions
        $b \leftarrow Q_k[Sy_k[w, x]] = i$ ▷ If the synapse exists
        $w \leftarrow w + 1$
      end while
    end for
    return $C_k[i] \in L(E_j) \ \wedge \ \neg b$
  end if
end procedure

3.3.2. Plasticity Rules

For dynamic SNP systems with plasticity rules, the synapse matrix can be allocated in advance to the exact size q, since no new neurons are created. Thus, there is no need of using a neuron-id map vector as before. However, enough rows (value z) in the synapse matrix have to be pre-established to support the maximum amount of synapses. Fortunately, this can be pre-computed by looking to the initial out degrees of the neurons and the size of the neuron sets in the plasticity rules adding synapses. We encode a plasticity rule $r_j \equiv E_j / a_j^c \rightarrow \alpha_j k_j (i_j, N_j)$, with $\alpha_j = +/-/\pm/\mp$ as follows: $r_j = (i_j, E_j, c_j, \alpha_j, k_j, N_j)$. Next, we define the value of z_p for SNP systems with plasticity rules: $z_p = max\{|pres(i) \cup Nt_i|, 1 \leq i \leq q\}$, where $Nt_i = \bigcup_{j=1}^{j=m} N_j, r_j \in R_i, r_j = (i_j, E_j, c_j, \alpha_j, k_j, N_j), \alpha_j \in \{+, \pm, \mp\}$. In other words, z_p is the maximum out degree (z) that a neuron can have initially plus those new connections that can be created with plasticity rules inside that neuron. This result can be refined for plasticity rules having $\alpha \in \{\pm, \mp\}$, because we know up to k new synapses can be created at a time. However, for simplicity, we will use the formula above.

Algorithm 9 Compute next function for dynamic SNP systems with budding and division rules using optimized compressed matrix representation.

```
1: procedure COMPUTE_NEXT(C_k, M_k, S_k)
2:     C_{k+1} ← C_k                                   ▷ Create a copy of C_k
3:     (M'_k, Q_k, _, _) ← M_k                         ▷ Extract info from transition tuple
4:     (Ru_Π, Sy_k) ← M'_k                             ▷ Extract info from transition matrix
5:     q_k ← |C_k|                                     ▷ Get current amount of neurons.
6:     for i ← 1 ... q_k do                            ▷ For each neuron. This loop is parallelizable.
7:         j ← S_k[i]                                  ▷ Index of rule to fire in the neuron
8:         if j ≠ 0 then                               ▷ Only if there is a rule.
9:             (c_j, p_j) ← Ru_Π[j]                    ▷ Get rule info
10:            if c_j > 0 then                         ▷ Execution of a spiking or forgetting rule
11:                C_{k+1}[i] ← C_{k+1}[i] − c_j       ▷ Consume spikes in firing neuron
12:                w ← 1                               ▷ Next while stops if p_j = 0, i.e., a firing rule
13:                while p_j > 0 ∧ Sy_k[w, i] ≠ 0 ∧ w ≤ z do   ▷ Until an empty value
14:                    h ← Sy_k[w, i]                  ▷ Get connected neuron by a synapse
15:                    C_{k+1}[h] ← C_{k+1}[h] + p_j   ▷ Produce spikes in connected neuron
16:                    w ← w + 1
17:                end while
18:            else if c_j = 0 then                    ▷ Execution of a budding rule
19:                q_k ← q_k + 1                       ▷ Increment counter of neurons
20:                C_{k+1}[i] ← 0                      ▷ Empty the neuron i, neuron q_k is 0 already
21:                Q_k[q_k] ← p_j                      ▷ p_j is the label of the new neuron
22:                for w ← 1 ... z do
23:                    Sy_k[w, q_k] ← Sy_k[w, i]       ▷ Copy column i to the new one
24:                    if w = 1 then                   ▷ Update out synapses of i
25:                        Sy_k[w, i] ← q_k            ▷ The only new out synapse of i
26:                    else
27:                        Sy_k[w, i] ← 0              ▷ No more out synapses for i
28:                    end if
29:                end for
30:            else                                    ▷ Execution of a division rule
31:                (h_j, l_j) ← (c_j, −p_j)            ▷ Get new neurons labels
32:                q_k ← q_k + 1                       ▷ Increment counter of neurons
33:                C_{k+1}[i] ← 0                      ▷ Empty the neuron i, neuron q_k is 0 already
34:                Q_k[i] ← h_j                        ▷ The new label of the neuron
35:                Q_k[q_k] ← l_j                      ▷ The label of the new neuron
36:                for w ← 1 ... z do                  ▷ Copy out synapses to new neuron
37:                    Sy_k[w, q_k] ← Sy_k[w, i]       ▷ Copy column i to the new one
38:                end for
39:                for x ← 1 ... q_k − 1 do            ▷ Search for in synapses of neuron i
40:                    b ← False                       ▷ Boolean saying the synapse was found
41:                    w ← 1
42:                    while Sy_k[w, x] ≠ 0 ∧ w ≤ z do   ▷ Search the end of the column
43:                        b ← b ∨ Sy_k[w, x] = i      ▷ Search for neuron i
44:                        w ← w + 1
45:                    end while
46:                    if b then   ▷ If synapse was found, add new neuron at the end of the column
47:                        if w = z then
48:                            z ← z + 1               ▷ The neuron x has a larger out degree than z
49:                            Sy_k ← EXPAND_MATRIX(Sy_k, z, q_k)   ▷ Extend with one more row
50:                        end if
51:                        Sy_k[w, x] ← q_k            ▷ This can lead to overflows if w = z
52:                    end if
53:                end for
54:            end if
55:        end if
56:    end for
57:    return (C_{k+1}, M_k)
58: end procedure
```

First, we need to represent plasticity rules into vectors. We assume that np is the total amount of plasticity rules in the system, and that there is a total order between these rules. Given a plasticity rule, we can initialize the neuron-map and the precondition vector as with spiking rules. But in this case, we need a couple of new vectors and modify existing ones in order to represent all plasticity rules $r_j = (i_j, E_j, c_j, \alpha_j, k_j, N_j)$, with $j \in \{1 \dots np\}$ (following the imposed total order):

- Rule vector Ru_Π stores the following pair for a plasticity rule r_j: $(c_j, -j)$, that is, the consumed spikes c_j and the unique index of the plasticity rule j. This index is used to access the following vector, and it is stored as a negative value in order to detect that this is a plasticity rule.
- **Plasticity rule vector** Pr_Π, of size np, contains a tuple for each plasticity rule r_j of the form $(\alpha_j, k_j, ni_j, ne_j)$. The values ni_j and ne_j are used as indexes, from ni_j (start) to ne_j (end) for the following vector.
- **Plasticity neuron vector** Pn_Π, of size $npr = \sum_{j=1}^{j=np} |N_j|$, represents all neuron sets of plasticity rules. Thus, the elements of N_j are stored, in an ordered way, between $Pn_\Pi[ni_j]$ to $Pn_\Pi[ne_j]$.
- **Time vector** T_k is used to prevent neurons from applying rules during one step if the plasticity rule applied was of the type $\alpha_j \in \{\pm, \mp\}$. It contains binary (0 or 1) values.
- Transition matrix is therefore $M'_k = (Ru_\Pi, Sy_k, T_k, Pr_\Pi, Pn_\Pi)$. Note that the Synapse matrix can be modified at each step, so we use sub-index k.

The following operations have to be performed to reproduce the behavior plasticity rules (see Figure 7 for an illustration):

1. For each column in the synapse matrix executing a plasticity rule deleting x synapses:
   (a) If the intersection of the rule's neuron set and the current synapses in Sy_k is larger than x, then randomly select x synapses.
   (b) Loop through the rows (up to z_p iterations) to search the selected neurons and set them to null. Given that holes might appear in the column, its values can be sorted (or compacted).
2. For each column in the synapse matrix executing a plasticity rule adding x synapses:
   (a) If the difference of the rule's neuron set and the current synapses in Sy_k is larger than x, then randomly select x neurons.
   (b) Loop through the rows (up to z_p iterations) to insert the selected new synapses while keeping the order.

Checking the applicability of plasticity rules is much simpler than for division rules, given that the preconditions only affect to the local neuron and we do not need to know if there are existing synapses. However, for a plasticity rule r in a neuron i, and in order to create new or delete existing synapses, we need to check which neurons declared in r are already in the column i in the synapse matrix. This search can be $O(z_p \cdot n_r)$, being n_r the length of the neuron set in r. Nevertheless, by maintaining always the order in the column, this search can be done easily in $O(z_p + n_r)$.

Given that it is not usual to have budding and division rules together with plasticity rules, the pseudocode is based on the optimized matrix representation for static SNP systems (and not for division and budding) in Section 3.2. Algorithm 10 shows the re-definition of INIT_RULE_MATRICES and COMPUTE_NEXT functions, replacing those from Algorithm 3. For COMPUTE_NEXT, the implementation is very similar to the original one, but it just call to a new function, PLASTICITY, which actually modify the synapses of the neuron (by just modifying its corresponding column in the synapse matrix). This function and its auxiliaries are defined in Algorithms 11 and 12, respectively.

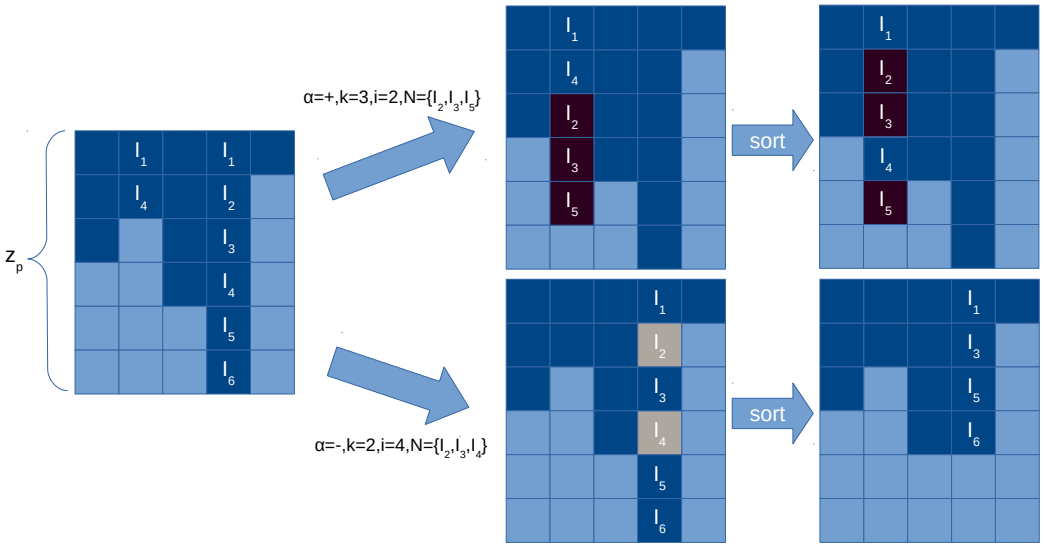

Figure 7. Illustration of application of a plasticity rule in the synapse matrix with compressed representation. Light blue cells in the synapse matrix are empty values (0), dark cells are positions with values greater than 0 (i.e., with neuron label). Two examples are give, in case of adding new synapses (**top**) and in case of deleting synapses (**bottom**). We sort the synapses per column for more efficiency.

4. Algorithms

In this section we define the algorithms implementing the methods described in Section 3.

Let us first define a generic function to create a new, empty (all values to 0) vector of size s as follows: EMPTY_VECTOR(s). In order to create an empty matrix with f rows and c columns, we will use the following function: EMPTY_MATRIX(f, c). Next, the pseudocodes for simulating **static SNP systems with sparse representation** are given. Algorithm 3 shows the INIT and COMPUTE_NEXT functions, while Algorithm 4 shows the selection of spiking vectors.

For **ELL-based matrix representation for static SNP systems**, we need to re-define only two functions (INIT_RULE_MATRICES and COMPUTE_NEXT) from Algorithm 3 (static SNP systems with sparse representation) as shown in Algorithm 5.

For our **optimized matrix representation for static SNP systems**, we need to re-define only two functions (INIT_RULE_MATRICES and COMPUTE_NEXT) from Algorithm 3 (static SNP systems with sparse representation) as shown in Algorithm 6. Moreover, the following two lines in the definition of COMBINATIONS function at Algorithm 4 are required, in order to support a spiking vector of size q:

- Line 13 at Algorithm 4: $S \leftarrow$ EMPTY_VECTOR(q)
- Line 22 at Algorithm 4: $S'[i] \leftarrow j$

For **dynamic SNP systems with budding and division rules**, the following functions are redefined: INIT functions as in Algorithm 7, APPLICABLE function as in Algorithm 8, and COMPUTE_NEXT function as in Algorithm 9. The SPIKING_VECTORS function, as defined in Algorithm 4 and modified in Section 3.2 for optimized matrix representation, is slightly modified (just two lines) to support the neuron-id map vector as follows:

- Line 6 at Algorithm 4: $(\_, Q_k, P_\Pi, N_\Pi) \leftarrow M_k$
- Line 18 at Algorithm 4: **for** $j \leftarrow N_\Pi[Q_k[i]] \ldots N_\Pi[Q_k[i]+1]-1$

Algorithm 10 Functions for dynamic SNP systems with plasticity rules using optimized compressed matrix representation.

procedure INIT_RULE_MATRICES(Π)
  $(R, m, q, z_p, np, npr, pres) \leftarrow \Pi$ ▷ Get information from Π, z_p is different for plasticity.
  $P_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create preconditions vector
  $Ru_\Pi \leftarrow$ EMPTY_VECTOR(m) ▷ Create rule vector
  $Pr_\Pi \leftarrow$ EMPTY_VECTOR(np) ▷ Create plasticity rule vector
  $Pn_\Pi \leftarrow$ EMPTY_VECTOR(npr) ▷ Create plasticity neuron vector
  $pk \leftarrow 1$ ▷ Counter for plasticity rule vector
  $ni \leftarrow 1$ ▷ Counter for plasticity neuron vector
  for all $r_j \in R, j \leftarrow 1 \dots m$ **do** ▷ For each rule (column). This loop is parallelizable.
    if $r_j = (i_j, E_j, c_j, p_j)$ **then** ▷ If spiking rule
      $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
      $Ru_\Pi[j] \leftarrow (c_j, p_j)$ ▷ Store it in rule vector
    else if $r_j = (i_j, E_j, c_j, \alpha_j, k_j, N_j)$ **then** ▷ If plasticity rule
      $P_\Pi[j] \leftarrow (E_j, c_j)$ ▷ Store it in precondition vector
      $Ru_\Pi[j] \leftarrow (c_j, -pk)$ ▷ Store it in rule vector
      $Pr_\Pi[pk] \leftarrow (\alpha_j, k_j, ni, ni + |N_j|)$ ▷ Store it in rule vector
      $Pn_\Pi[ni] \leftarrow$ SORT(N_j) ▷ Sort and store N_j in Pn after position ni
      $pk \leftarrow pk + 1$
      $ni \leftarrow ni + |N_j|$
    end if
  end for
  $T_0 \leftarrow$ EMPTY_VECTOR(q) ▷ Create time vector
  $Sy_0 \leftarrow$ EMPTY_MATRIX(z_p, q) ▷ Create synapse matrix
  for all $i \leftarrow 1 \dots q$ **do** ▷ For each neuron (column in synapse matrix)
    $k \leftarrow 1$ ▷ k is our iterator, compacting the rows
    for all $h \in pres(i)$ **do** ▷ For each out synapse
      $Sy_0[k, i] \leftarrow h$
      $k \leftarrow k + 1$ ▷ We know that $k \leq z_p$
    end for
  end for
  $M'_0 \leftarrow (Ru_\Pi, Sy_0, T_0, Pr_\Pi, Pn_\Pi)$ ▷ New transition matrix
  return (M'_0, P_Π)
end procedure

procedure COMPUTE_NEXT(C_k, M_k, S_k)
  $C_{k+1} \leftarrow C_k$ ▷ Create a copy of C_k
  $(M'_k, P_\Pi, N_\Pi) \leftarrow M_k$ ▷ Get some content of transition tuple
  $(Ru_\Pi, Sy_k, Pr_\Pi, Pn_\Pi) \leftarrow M'_k$
  for $i \leftarrow 1 \dots q$ **do** ▷ For each neuron. This loop is parallelizable.
    $j \leftarrow S_k[i]$ ▷ Index of rule to fire in the neuron
    if $j \neq 0 \vee T_k[i] = 1$ **then** ▷ Only if there is a rule or blocked neuron
      $(c_j, p_j) \leftarrow Ru_\Pi[j]$ ▷ Get rule info
      $C_{k+1}[i] \leftarrow C_{k+1}[i] - c_j$ ▷ Consume spikes in firing neuron
      if $p_j > 0$ **then** ▷ If a spiking rule
        $w \leftarrow 1$
        while $Sy_\Pi[w, i] \neq 0 \wedge w \leq z_p$ **do** ▷ Until an empty value or the maximum
          $h \leftarrow Sy_\Pi[w, i]$ ▷ Get connected neuron by a synapse
          $C_{k+1}[h] \leftarrow C_{k+1}[h] + p_j$ ▷ Produce spikes in connected neuron
          $w \leftarrow w + 1$
        end while
      else if $p_j < 0$ **then** ▷ If a plasticity rule
        $(A, t) \leftarrow$ PLASTICITY($Sy_k[, i], -p_j, Pr_\Pi, Pn_\Pi$) ▷ Modify only column i
        $Sy_k[, i] \leftarrow A \; T_k[i] \leftarrow t$
      end if
    end if
    $T_k[i] \leftarrow 0$ ▷ Reset time vector
  end for
  $M'_{k+1} \leftarrow (Ru_\Pi, Sy_k, T_k, Pr_\Pi, Pn_\Pi)$ ▷ Next transition matrix
  return $(C_{k+1}, (M'_\Pi, P_\Pi, N_\Pi)$
end procedure

For **dynamic SNP systems with plasticity rules**, the pseudocode is based on the optimized matrix representation for static SNP systems (and not for division and budding) in Section 3.2. Algorithm 10 shows the re-definition of INIT_RULE_MATRICES and COMPUTE_NEXT functions, replacing those from Algorithm 3. As for line 17, we assume that the function SORT exists, which takes a set of neurons, sorts them by id, and generates a vector. Moreover, we can copy vectors directly from one position by just one assignation. The new PLASTICITY function is defined in Algorithm 11, and its auxiliaries are defined in Algorithm 12.

Algorithm 11 Function for plasticity mechanism using optimized compressed matrix representation.

```
1: procedure PLASTICITY(A_i, j, Pr_Π, Pn_Π)
2:     (α_j, k_j, ni_j, ne_j) gets Pr_Π[j]                ▷ Get info of plasticity rule
3:     npr_j ← ne_j − ni_j                                ▷ Number of neurons in the neuron set
4:     N_j ← EMPTY_VECTOR(npr_j)                          ▷ Create a vector with the neuron set
5:     for x ← 1 ... npr_j do
6:         N_j[x] ← Pn_Π[ni_j + x]                        ▷ Copy the contents of the neuron set
7:     end for
8:     t ← 0                                              ▷ Vale for time vector, only 1 for mp, pm
9:     if α_j = − then                                    ▷ Delete synapses
10:        A_i ← DEL_SYNAPSES(A_i, k_j, N_j)
11:    else if α_j = + then                               ▷ Add synapses
12:        A_i ← ADD_SYNAPSES(A_i, k_j, N_j)
13:    else if α_j = ± then                               ▷ Add and delete synapses
14:        A_i ← ADD_SYNAPSES(A_i, k_j, N_j)
15:        A_i ← DEL_SYNAPSES(A_i, k_j, N_j)
16:        t ← 1
17:    else if α_j = ∓ then                               ▷ Delete and add synapses
18:        A_i ← DEL_SYNAPSES(A_i, k_j, N_j)
19:        A_i ← ADD_SYNAPSES(A_i, k_j, N_j)
20:        t ← 1
21:    end if
22:    return (A_i, t)
23: end procedure
```

In order to keep Algorithm 12 simple, we assume that the functions INTERSEC, DIFF, and DELETE_RANDOM are already defined. As mentioned above, INTERSEC and DIFF can be implemented with algorithms of complexity $O(z_p + n_r)$, given that the vectors (a column of synapse matrix and a chunk of plasticity neuron vector) are already sorted. We also assume that DELETE_RANDOM is a function that randomly select k elements from a total of n while keeping the order between elements. This can be done with an algorithm of complexity $O(k^2)$.

5. Results

In this section we conduct a complexity analysis (for both time and memory) of the algorithms. In order to define the formulas, we need to introduce a set of descriptors for a spiking neural P system Π. These are described in Table 1. Moreover, Table 2 summarizes the vectors and matrices employed by each representation, and their corresponding sizes defined according to the descriptors. We use the following short names for the representations: Sparse (original sparse representation as Section 3), ELL (ELL compressed representation as in Section 3.1), optimized static (optimized static compressed representation as in Section 3.2), division and budding (optimized dynamic compressed representation for division and budding as in Section 3.3.1), and plasticity (optimized dynamic compressed representation for plasticity as in Section 3.3.2).

Algorithm 12 Auxiliary functions for plasticity mechanism using optimized compressed matrix representation.

procedure DEL_SYNAPSES(A, k, N)
  $N' \leftarrow$ INTERSEC(A, N) ▷ Calculate $A \cap N$ (involved synapses to be deleted)
  if $|N'| > k$ **then** ▷ If more than k neurons, select randomly
    $N' \leftarrow$ DELETE_RANDOM($N', |N'| - k$) ▷ A random set of k neurons
  end if
  $k \leftarrow |N'|$ ▷ The new amount of synapses to delete
  $w \leftarrow p \leftarrow s \leftarrow 1$ ▷ Initialize iterators
  while $w \leq z_p \wedge A[w] \neq 0$ **do** ▷ Loop over the column
    if $A[w] = N'[s]$ **then** ▷ Synapse to delete
      $A[w] \leftarrow 0$ ▷ Delete the synapse
      $s \leftarrow s + 1$ ▷ Advance in N' vector
    else
      if $p < w$ **then** ▷ Need to compact the vector
        $A[p] \leftarrow A[w]$ ▷ p is the last compacted position
        $A[w] \leftarrow 0$
      end if
      $p \leftarrow p + 1$ ▷ Advance the last compacted position p
    end if
    $w \leftarrow w + 1$
  end while
  return A
end procedure

procedure ADD_SYNAPSES(A, k, N, n)
  $N' \leftarrow$ DIFF(N, A) ▷ Calculate $N \setminus A$ (not involved synapses to create)
  if $|N'| > k$ **then** ▷ If more than k neurons, select randomly
    $N' \leftarrow$ DELETE_RANDOM($N', |N'| - k$) ▷ A random set of k neurons
  end if
  $k \leftarrow |N'|$ ▷ The new amount of synapses to delete
  $B \leftarrow$ EMPTY_VECTOR(z_p) ▷ Create the output
  $w \leftarrow p \leftarrow s \leftarrow 1$ ▷ Initialize iterators
  while $w \leq z_p \wedge \neg(A[w] = 0 \wedge s \leq k)$ **do** ▷ Loop over the column
    if $A[w] > N'[s]$ **then** ▷ Synapse to add
      $B[p] \leftarrow N'[s]$ ▷ Add the synapse
      $s \leftarrow s + 1$
    else
      $B[p] \leftarrow A[w]$ ▷ Keep the synapse
      $w \leftarrow w + 1$
    end if
    $p \leftarrow p + 1$
  end while
  return B
end procedure

Table 1. Descriptors of an SNP system.

Descriptor	Description
q	Number of initial neurons
m	Total number of rules
z	Number of rows (column size) for optimized matrix representation. Also, maximum out degree of a neuron
z'	Number of rows (column size) for ELL matrix representation. $z' = z + 1$
q_{max}	Maximum amount of neurons to handle during simulation for division and budding. $q' \geq q$
z_p	Number of rows (column size) for optimized matrix representation for plasticity
n_p	Maximum size of a neuron set in plasticity rules.
npr	Sum of neuron set sizes of all plasticity rules.
k_p	Maximum value of k in plasticity rules.

Table 2. Size of matrices employed in the representations for SNP systems. Those whose name are in bold were used for the total calculation, which assumes just one spiking vector.

Notation/Name	Sparse	ELL	Optimized Static	Division and Budding	Plasticity
C_k **Configuration Vector**	q	q	q	q_{max}	q
S_k **Spiking vector**	m	m	q	q_{max}	q
P_{Π} Preconditions vector	$2m$	$2m$	$2m$	$2m$	$2m$
N_{Π} Neuron-rule map vector	$q+1$	$q+1$	$q+1$	$q+1$	$q+1$
Sy_k Synapse matrix			$q \cdot z$	$q_{max} \cdot z$	$q \cdot z_p$
Ru_{Π} Rule vector			$2m$	$2m$	$2m$
Q_k Neuron-id map vector				q_{max}	
Pr_{Π} Rule vector					$4m$
Pn_{Π} Rule vector					npr
T_k Rule vector					q
M'_{Π} Transition matrix	$m \cdot q$	$2 \cdot m \cdot z'$	$q \cdot z + 2m$	$q_{max} \cdot z + 2m$	$q(z_p+1)+6m+npr$
M_k **Transition tuple**	$m \cdot q + 2m + q + 1$	$m(2z'+2)+q+1$	$q(z+1)+4m+1$	$q_{max}(z+1)+4m+q+1$	$q(z_p+2)+8m+npr+1$
TOTAL	$m \cdot q + 3m + 2q + 1$	$m(2z'+3)+2q+1$	$q(z+3)+4m+1$	$q_{max}(z+2)+4m+2q+1$	$q(z_p+4)+8m+npr+1$

According to Table 2, we can limit the value of q_{max} for dynamic SNP systems with division and budding with the following formula: $q_{max} = \lfloor \frac{MT-4m+2q+1}{z+2} \rfloor$, where MT is the maximum amount of memory in the system (measured in the word size employed to encode the elements of all the matrices and vectors; e.g., 4 Bytes). Moreover, we can infer when the matrix representation will be smaller for static SNP systems: ELL is better than sparse when $z < \frac{q-2}{2}$; optimized is better than ELL when $z > \frac{q-m}{2m-q}$; optimized is better than sparse when $z < m-2$. In other words, our optimized compressed representation is worth when the the maximum out degree of the neurons is less than the total number of rules minus 2.

For dynamic SNP systems, given can say that a solution to a problem using an SNP with plasticity rules is better than a solution based on division and budding, if $q_{max} > \frac{q(z_p+2)+4m+npr}{z+2}$; in other words, if we can know the maximum amount of neurons to generate, and this number is greater than a formula based on number of initial neurons, number of rules, and number of elements in the neuron set and the max out degree, then the solution will need less memory using plasticity.

Finally, Table 3 shows the order of complexity of each function as defined for each representation. We can see that COMPUTE_NEXT gets reduced in complexity as well when using optimized static representation against ELL and sparse, given that we expect that $m \leq q$, and also $z < m-2$. However, we can see that implementing division and budding explodes the complexity of the algorithms, since they need to loop over all the neurons checking for in-synapses. This also depends on the total amount of generated neurons in a given step. This is also the case for the generation of spiking vectors, because the applicability function also needs to loop over all existing neurons. However, for dynamic networks, plasticity keeps the complexity with the amount of neurons, the value z, and the descriptors of plasticity rules (max value of k and amount of neurons in a neuron set n_p).

Table 3. Algorithmic order of complexity of the main functions employed in the simulation loop (i.e., excluding init functions) for each representation.

Function	Sparse	ELL	Optimized Static	Division & Budding	Plasticity
APPLICABLE	$O(1)$	$O(1)$	$O(1)$	$O(q_{max} \cdot z)$	$O(1)$
SPIKING_VECTORS	$O(m)$	$O(m)$	$O(m)$	$O(m \cdot q_{max} \cdot z)$	$O(m)$
PLASTICITY					$O(n_p + k_p^2 + z_p)$
COMPUTE_NEXT	$O(q \cdot m)$	$O(z' \cdot m)$	$O(z \cdot q)$	$O(q_{max} \cdot z)$	$O(q \cdot (n_p + k_p^2 + z_p))$

Therefore, we can see that using our compressed representations, both the memory footprint of the simulators and their complexity are reduced, as long as the maximum out degree of neurons is a low number. Furthermore, we can see that for dynamic networks, plasticity is an option that keeps the complexity balanced, since we know in advance the amount of neurons and synapses.

Let us make an easy example of comparison with an example from the literature. For example, if we take the SNP system for sorting natural numbers as defined in [42], then we have that $q = 3n$, $m = n + n^2$ and $z = n$, where n is the amount of natural numbers to sort. Thus:

- The size of the sparse representation is $3n^3 + 6n^2 + 5n + 1$ and the complexity of COMPUTE_NEXT is $O(n^3)$.
- The size of the ELL representation is $2n^3 + 7n^2 + 11n + 1$ and the complexity of COMPUTE_NEXT is $O(n^3)$.
- The size of the optimized representation is $7n^2 + 13n + 1$ and the complexity of COMPUTE_NEXT is $O(n^2)$.

The optimized representation drastically decreases the order of complexity and amount of memory spent for the algorithms, going from orders of n^3 to n^2. ELL has a similar order of complexity to that of sparse, but the amount of memory is just a bit decreased. Figure 8 shows that the reduction of the memory footprint achieved with the compressed representations takes effect after $n > 3$. Figure 9 shows that the optimized representation scales better than ELL and sparse. ELL is only a bit better than the sparse representation, demonstrating the need for using the optimized one, which significantly scales much better.

Finally, we also analyze a uniform solution to 3SAT with SNP systems without delays as in [43] (Figure 10). We can see that $q = 8n^3 + 3n + 3$, $m = 64n^3 + 6n + 3$ and $z = 8n^3$, where n is the amount of variables in the 3SAT instance. We can see that $z < m - 3$, so our optimized implementation will be able to save some memory. Therefore:

- The size of the sparse representation is $512n^6 + 240n^4 + 424n^3 + 18n^2 + 51n + 25$ and the complexity of COMPUTE_NEXT is $O(n^6)$.
- The size of the ELL representation is $1024n^6 + 96n^4 + 384n^3 + 36n + 22$ and the complexity of COMPUTE_NEXT is $O(n^6)$.
- The size of the optimized representation is $64n^6 + 24n^4 + 304n^3 + 33n + 22$ and the complexity of COMPUTE_NEXT is $O(n^6)$.

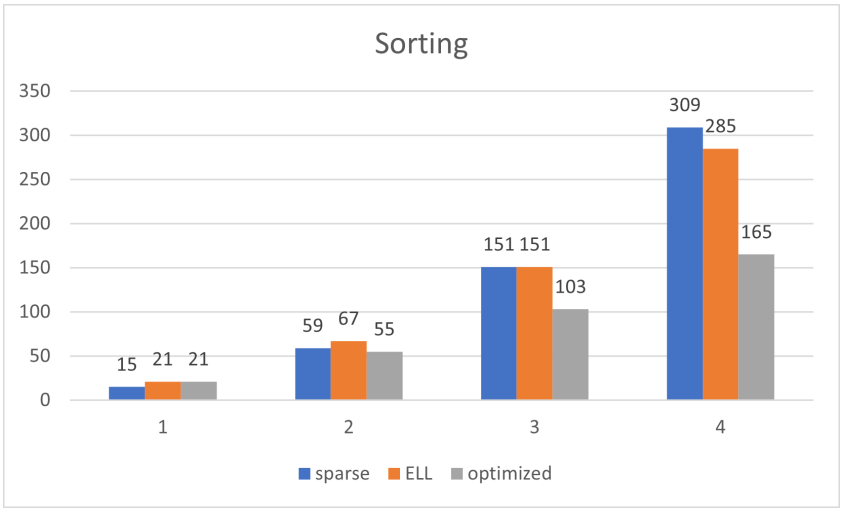

Figure 8. Memory size of the matrix representation (Y-axis) depending on the amount of natural numbers (n, X-axis) for the model of natural number sorting in [42], using sparse, ELL and optimized representation, for only $n = 1, 2, 3, 4$.

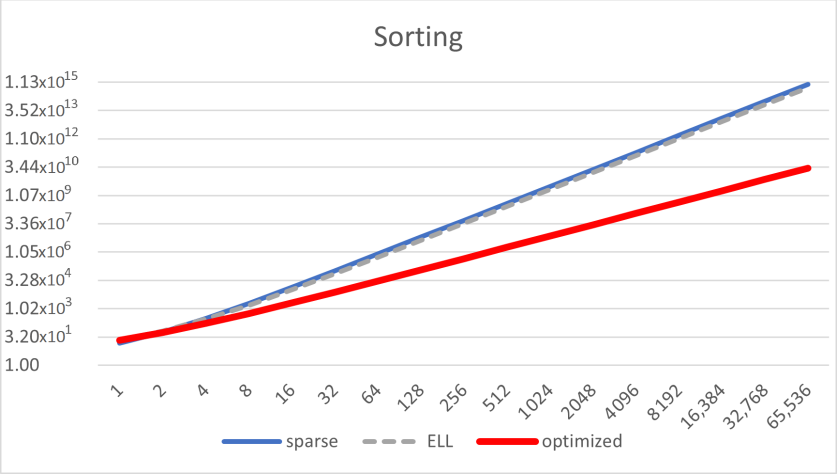

Figure 9. Memory size of the matrix representation (Y-axis in log scale) depending on the amount of natural numbers (n, X-axis in log scale) for the model of natural number sorting, using sparse, ELL, and optimized representation, for $1 \leq n \leq 65536$.

We can see that the memory footprint is decreased but it is still of the same order of magnitude ($O(n^6)$), and the same happens with the computing complexity. Thus, our representation helps to reduce memory, although not significantly for this specific solution. This is mainly due to having a high value of z. We can see in Figure 10 how the reduction of memory takes place only for optimized representation as long as n increases. It is interesting to see that the ELL representation is even worse than just using sparse representation.

Finally, let us analyze the size of the solution uniform solution to subset sum with plasticity rules in [34]. The descriptors for the matrix representation of a dynamic SNP system with plasticity rules are the following: $q = 4n + 9$, $m = 5n + 11$, $npr = 2n$, $z_p = 2$, where n is the number of sets V, therefore, the memory footprint is described as: $66n + 143$. If we were using a sparse representation where the transition matrix is of order $m \cdot q$, then the amount of memory is of order $O(n^2)$.

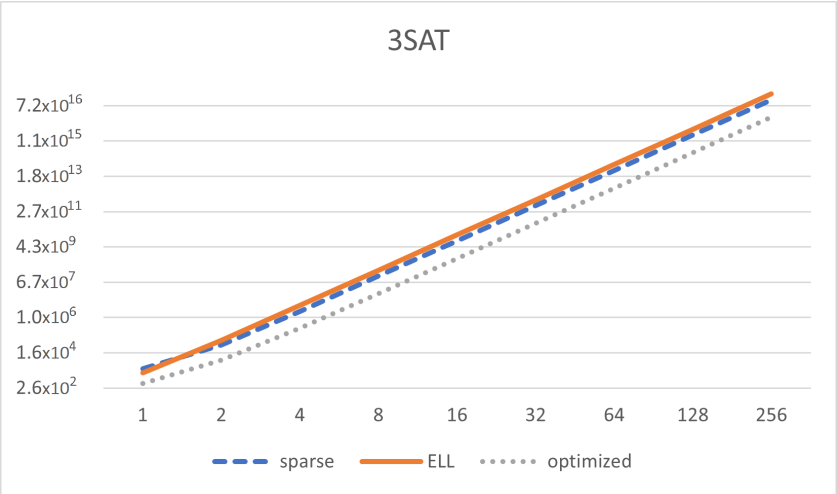

Figure 10. Memory size of the matrix representation (Y-axis in log scale) depending on the number of variables in the SAT formula ($1 \leq n \leq 256$, X-axis in log scale) for the model of 3SAT, using sparse, ELL, and optimized representation.

6. Conclusions

In this paper, we addressed the problem of having very sparse matrices in the matrix representation of SNP systems. Usually, the graph defined for an SNP system is not fully connected, leading to sparse matrices. This drastically downgrades the performance of the

simulators. However, sparse matrices are a known issue in other disciplines, and efficient representations have been introduced in the literature. There are even solutions tailored for parallel architectures such as GPUs.

We propose two efficient compressed representations for SNP systems, one based on the classic format ELL, and an optimized one based on a combination of CSR and ELL. This representation gives room to support rules for dynamic networks: division, budding, and plasticity. The representation for plasticity poses more advantages than the one for division and budding, since the synapse matrix size can be pre-computed. Thus, no label mapping nor empty columns to host new neurons are required. Moreover, simulating the creation of new neurons in parallel can damage the performance of the simulator significantly, because this operation can be sequential. Plasticity rules do not create new neurons, so this is avoided.

As future work, we plan to provide implementations of these designs within cuSNP [21] and P-Lingua [44] frameworks to provide high performance simulations with real examples from the literature. We believe that these concepts will help to bring efficient tools to simulate SNP systems on GPUs, enabling the simulation of large networks in parallel. Specifically, we will use these designs to develop a new framework for automatically designing SNP systems using genetic algorithms [45]. Another tool that could benefit from the inclusion of this new type of representation are visual tools for SNP systems [46]. Moreover, our optimized designs will enable the effective usage of spiking neural P systems on industrial processes such as [47–50], and to optimization applications as [51,52]. SNP systems have been used in many applications [5], and in order to be used in industrial applications we need efficient simulators where compressed representations of sparse matrices can help.

Numerical SNP systems (or NSNP systems) [17,53] are SNP system variants which are largely dissimilar to many variants of SNP systems, especially to the variants considered in this paper, for at least two main reasons: (1) rules in NSNP systems do not use regular expressions, and instead use linear functions, so that rules are applied when certain values or threshold of the variables in such functions are satisfied, and (2) the variables in the functions are real-valued, unlike the natural numbers associated with strings and regular expressions. One of the main goals in [17] for introducing NSNP systems is to create an SNP system variant, which in a future work may be more feasible for use with training algorithms in traditional neural networks [53]. For these reasons, we plan to extend our algorithms and compressed data structures for NSNP systems. We think that simulators for this variant can be effectively accelerated on GPUs. Specifically, GPUs are devices designed for floating point operations and not for integer arithmetic, although the latter is supported.

We also plan to include more models and ingredients into these new methods, such as delays, weights, dendrites, rules on synapses, and scheduled synapses, among others. Moreover, a recent work in SNP systems with plasticity shows that having the same set of rules in all neurons leads to Turing complete algorithms [54]. This means that m descriptor can be common to all neurons, leading to smaller representations for this kind of systems. We plan to study this deeper and combine it with our representations. Our aim on focusing on plasticity is also related to other results involving this ingredient in other fields such as machine learning [55].

Author Contributions: Conceptualization, M.Á.M.-d.-A., and F.G.C.C.; methodology, M.Á.M.-d.-A. and D.O.-M.; validation, M.Á.M.-d.-A., F.G.C.C. and D.O.-M.; formal analysis, D.O.-M., I.P.-H. and H.N.A.; investigation, M.Á.M.-d.-A. and F.G.C.C.; resources, D.O.-M., F.G.C.C. and I.P.-H.; writing—original draft preparation, M.Á.M.-d.-A., D.O.-M., I.P.-H., F.G.C.C., H.N.A.; writing—review and editing, M.Á.M.-d.-A., D.O.-M., I.P.-H., F.G.C.C., H.N.A.; supervision, I.P.-H. and H.N.A.; All authors have read and agreed to the published version of the manuscript.

Funding: Financiado por: FEDER/Ministerio de Ciencia e Innovación—Agencia Estatal de Investigación/_Proyecto (TIN2017-89842-P). F.G.C. Cabarle is supported in part by the ERDT program of the *DOST-SEI*, Philippines, and the Dean Ruben A. Garcia PCA from UP Diliman. H.N. Adorna

is supported by *Semirara Mining Corp Professorial Chair for Computer Science*, *RLC* grant from UPD OVCRD, and *ERDT-DOST* research grant.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

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

References

1. Păun, G. Computing with membranes. *J. Comput. Syst. Sci. TUCS Rep. No 208* **2000**, *61*, 108–143. [CrossRef]
2. Song, B.; Li, K.; Orellana-Martín, D.; Pérez-Jiménez, M.J.; Pérez-Hurtado, I. A Survey of Nature-Inspired Computing: Membrane Computing. *ACM Comput. Surv.* **2021**, *54*. [CrossRef]
3. Arteta Albert, A.; Díaz-Flores, E.; López, L. F.D.M.; Gómez Blas, N. An In Vivo Proposal of Cell Computing Inspired by Membrane Computing. *Processes* **2021**, *9*, 511. [CrossRef]
4. Ionescu, M.; Pundefinedun, G.; Yokomori, T. Spiking Neural P Systems. *Fundam. Inform.* **2006**, *71*, 279–308.
5. Fan, S.; Paul, P.; Wu, T.; Rong, H.; Zhang, G. On Applications of Spiking Neural P Systems. *Appl. Sci.* **2020**, *10*, 7011. [CrossRef]
6. Păun, G.; Pérez-Jiménez, M.J., Spiking Neural P Systems. Recent Results, Research Topics. *Algorithmic Bioprocess.* **2009**, 273–291. [CrossRef]
7. Rong, H.; Wu, T.; Pan, L.; Zhang, G. Spiking neural P systems: theoretical results and applications. In *Enjoying Natural Computing*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 256–268.
8. Ibarra, O.; Leporati, A.; Păun, A.; Woodworth, S., Spiking Neural P Systems. In *The Oxford Handbook of Membrane Computing*; Păun, G., Rozenberg, G., Salomaa, A., Eds.; Oxford University Press: Oxford, UK, 2010; pp. 337–362.
9. Pan, L.; Wu, T.; Zhang, Z. *A Bibliography of Spiking Neural P Systems*; Technical Report; Bulletin of the International Membrane Computing Society: Sevilla, Spain, 2016.
10. Wang, J.; Hoogeboom, H.J.; Pan, L.; Păun, G.; Pérez-Jiménez, M.J. Spiking Neural P Systems with Weights. *Neural Comput.* **2010**, *22*, 2615–2646. [CrossRef]
11. Pan, L.; Wang, J.; Hoogeboom, H.J. Spiking Neural P Systems with Astrocytes. *Neural Comput.* **2012**, *24*, 805–825. [CrossRef]
12. Song, X.; Wang, J.; Peng, H.; Ning, G.; Sun, Z.; Wang, T.; Yang, F. Spiking neural P systems with multiple channels and anti-spikes. *Biosystems* **2018**, *169–170*, 13–19. [CrossRef]
13. Peng, H.; Bao, T.; Luo, X.; Wang, J.; Song, X.; Nez, A.R.N.; Pérez-Jiménez, M.J. Dendrite P systems. *Neural Netw.* **2020**, *127*, 110–120. [CrossRef] [PubMed]
14. Song, T.; Pan, L.; Păun, G. Spiking neural P systems with rules on synapses. *Theor. Comput. Sci.* **2014**, *529*, 82–95. [CrossRef]
15. Cabarle, F.G.C.; Adorna, H.N.; Jiang, M.; Zeng, X. Spiking neural P systems with scheduled synapses. *IEEE Trans. Nanobiosci.* **2017**, *16*, 792–801. [CrossRef]
16. Lazo, P.P.L.; Cabarle, F.G.C.; Adorna, H.N.; Yap, J.M.C. A return to stochasticity and probability in spiking neural P systems. *J. Membr. Comput.* **2021**, 1–13. [CrossRef]
17. Wu, T.; Pan, L.; Yu, Q.; Tan, K.C. Numerical Spiking Neural P Systems. *IEEE Trans. Neural Netw. Learn. Syst.* **2020**. [CrossRef]
18. Macías-Ramos, L.F.; Pérez-Hurtado, I.; García-Quismondo, M.; Valencia-Cabrera, L.; Pérez-Jiménez, M.J.; Riscos-Núñez, A. A P—Lingua Based Simulator for Spiking Neural p Systems. In *Proceedings 12th International Conference on Membrane Computing*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 7184, pp. 257–281. [CrossRef]
19. Zeng, X.; Adorna, H.; Martínez-del-Amor, M.A.; Pan, L.; Pérez-Jiménez, M.J. Matrix Representation of Spiking Neural P Systems. In Proceedings of the 11th International Conference on Membrane Computing, Jena, Germany, 24–27 August 2010; Volume 6501, pp. 377–391. [CrossRef]
20. Fatahalian, K.; Sugerman, J.; Hanrahan, P. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*; Association for Computing Machinery: New York, NY, USA, 2004; pp. 133–137. [CrossRef]
21. Carandang, J.; Villaflores, J.; Cabarle, F.; Adorna, H.; Martínez-del-Amor, M. CuSNP: Spiking Neural P Systems Simulators in CUDA. *Rom. J. Inf. Sci. Technol.* **2017**, *20*, 57–70.
22. Carandang, J.; Cabarle, F.; Adorna, H.; Hernandez, N.; Martínez-del-Amor, M.A. Nondeterminism in Spiking Neural P Systems: Algorithms and Simulations. In Proceedings of the 6th Asian Conference on Membrane Computing, Chengdu, China, 21–25 September 2017.
23. Cabarle, F.G.C.; Adorna, H.N.; Martínez-del Amor, M.Á.; Pérez Jiménez, M.d.J. Improving GPU simulations of spiking neural P systems. *Rom. J. Inf. Sci. Technol.* **2012**, *15*, 5–20.
24. Carandang, J.P.; Cabarle, F.G.C.; Adorna, H.N.; Hernandez, N.H.S.; Martínez-del-Amor, M.Á. Handling Non-determinism in Spiking Neural P Systems: Algorithms and Simulations. *Fundam. Inform.* **2019**, *164*, 139–155. [CrossRef]
25. Ochirbat, O.; Ishdorj, T.O.; Cichon, G. An error-tolerant serial binary full-adder via a spiking neural P system using HP/LP basic neurons. *J. Membr. Comput.* **2022**, *2*, 42–48. [CrossRef]

26. Martínez-del-Amor, M.A.; García-Quismondo, M.; Macías-Ramos, L.F.; Valencia-Cabrera, L.; Riscos-Núñez, A.; Pérez-Jiménez, M.J. Simulating P systems on GPU devices: A survey. *Fundam. Inform.* **2015**, *136*, 269–284. [CrossRef]
27. Muniyandi, R.C.; Maroosi, A. A Representation of Membrane Computing with a Clustering Algorithm on the Graphical Processing Unit. *Processes* **2020**, *8*, 1199. [CrossRef]
28. Martínez-del-Amor, M.; Pérez-Hurtado, I.; Orellana-Martín, D.; Pérez-Jiménez, M.J. Adaptative parallel simulators for bioinspired computing models. *Future Gener. Comput. Syst.* **2020**, *107*, 469–484. [CrossRef]
29. Martínez-del-Amor, M.Á.; Orellana-Martín, D.; Cabarle, F.G.C.; Pérez-Jiménez, M.J.; Adorna, H.N. Sparse-matrix representation of spiking neural P systems for GPUs. In Proceedings of the 15th Brainstorming Week on Membrane Computing, Sevilla, Spain, 31 January–5 February 2017; pp. 161–170.
30. Aboy, B.C.D.; Bariring, E.J.A.; Carandang, J.P.; Cabarle, F.G.C.; de la Cruz, R.T.A.; Adorna, H.N.; Martínez-del-Amor, M.Á. Optimizations in CuSNP Simulator for Spiking Neural P Systems on CUDA GPUs. In Proceedings of the 17th International Conference on High Performance Computing & Simulation, Dublin, Ireland, 15–19 July 2019; pp. 535–542. [CrossRef]
31. AlAhmadi, S.; Mohammed, T.; Albeshri, A.; Katib, I.; Mehmood, R. Performance Analysis of Sparse Matrix-Vector Multiplication (SpMV) on Graphics Processing Units (GPUs). *Electronics* **2020**, *9*, 1675. [CrossRef]
32. Adorna, H.; Cabarle, F.; Macías-Ramos, L.; Pan, L.; Pérez-Jiménez, M.; Song, B.; Song, T.; Valencia-Cabrera, L. Taking the pulse of SN P systems: A Quick Survey. In *Multidisciplinary Creativity*; Spandugino: Bucharest, Romania, 2015; pp. 1–16.
33. Cabarle, F.G.; Adorna, H.N.; Pérez-Jiménez, M.J.; Song, T. Spiking Neural P Systems with Structural Plasticity. *Neural Comput. Appl.* **2015**, *26*, 1905–1917. [CrossRef]
34. Cabarle, F.G.C.; Hernandez, N.H.S.; Martínez-del-Amor, M.Á. Spiking neural P systems with structural plasticity: Attacking the subset sum problem. In *International Conference on Membrane Computing*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 106–116.
35. Pan, L.; Păun, G.; Pérez-Jiménez, M. Spiking neural P systems with neuron division and budding. *Sci. China Inf. Sci.* **2011**, *54*, 1596–1607. [CrossRef]
36. Jimenez, Z.; Cabarle, F.; de la Cruz, R.T.; Buño, K.; Adorna, H.; Hernandez, N.; Zeng, X. Matrix representation and simulation algorithm of spiking neural P systems with structural plasticity. *J. Membr. Comput.* **2019**, *1*, 145–160. [CrossRef]
37. Cabarle, F.G.C.; de la Cruz, R.T.A.; Cailipan, D.P.P.; Zhang, D.; Liu, X.; Zeng, X. On solutions and representations of spiking neural P systems with rules on synapses. *Inf. Sci.* **2019**, *501*, 30–49. [CrossRef]
38. Orellana-Martín, D.; Martínez-del-Amor, M.; Valencia-Cabrera, L.; Pérez-Hurtado, I.; Riscos-Núñez, A.; Pérez-Jiménez, M.J. Dendrite P Systems Toolbox: Representation, Algorithms and Simulators. *Int. J. Neural Syst.* **2021**, *31*, 2050071. [CrossRef] [PubMed]
39. Kirk, D.B.; Hwu, W.W. *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2016.
40. NVIDIA CUDA C Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 15 February 2021).
41. Bell, N.; Garland, M. *Efficient Sparse Matrix-Vector Multiplication on CUDA*; NVIDIA Technical Report NVR-2008-004; NVIDIA Corporation: Santa Clara, CA, USA, 2008.
42. Ionescu, M.; Sburlan, D. Some Applications of Spiking Neural P Systems. *Comput. Inform.* **2008**, *27*, 515–528.
43. Leporati, A.; Mauri, G.; Zandron, C.; Păun, G.; Pérez-Jiménez, M.J. Uniform Solutions to SAT and Subset Sum by Spiking Neural P Systems. *Nat. Comput. Int. J.* **2009**, *8*, 681–702. [CrossRef]
44. Pérez-Hurtado, I.; Orellana-Martín, D.; Zhang, G.; Pérez-Jiménez, M.J. P-Lingua in two steps: flexibility and efficiency. *J. Membr. Comput.* **2019**, *1*, 93–102. [CrossRef]
45. Casauay, L.J.P.; Cabarle, F.G.G.; Macababayao, I.C.H.; Adorna, H.N.; Zeng, X.; Martínez-del-Amor, M.Á. A Framework for Evolving Spiking Neural P Systems. *Int. J. Unconv. Comput.* **2021**, *16*, 121–139.
46. Fernandez, A.D.C.; Fresco, R.M.; Cabarle, F.G.C.; de la Cruz, R.T.A.; Macababayao, I.C.H.; Ballesteros, K.J.; Adorna, H.N. Snapse: A Visual Tool for Spiking Neural P Systems. *Processes* **2021**, *9*, 72. [CrossRef]
47. Lin, H.; Zhao, B.; Liu, D.; Alippi, C. Data-based fault tolerant control for affine nonlinear systems through particle swarm optimized neural networks. *IEEE/CAA J. Autom. Sin.* **2020**, *7*, 954–964. [CrossRef]
48. Zerari, N.; Chemachema, M.; Essounbouli, N. Neural network based adaptive tracking control for a class of pure feedback nonlinear systems with input saturation. *IEEE/CAA J. Autom. Sin.* **2019**, *6*, 278–290. [CrossRef]
49. Gao, S.; Zhou, M.; Wang, Y.; Cheng, J.; Yachi, H.; Wang, J. Dendritic Neuron Model With Effective Learning Algorithms for Classification, Approximation, and Prediction. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 601–614. [CrossRef] [PubMed]
50. Shang, M.; Luo, X.; Liu, Z.; Chen, J.; Yuan, Y.; Zhou, M. Randomized latent factor model for high-dimensional and sparse matrices from industrial applications. *IEEE/CAA J. Autom. Sin.* **2019**, *6*, 131–141. [CrossRef]
51. Liu, W.; Luo, F.; Liu, Y.; Ding, W. Optimal Siting and Sizing of Distributed Generation Based on Improved Nondominated Sorting Genetic Algorithm II. *Processes* **2019**, *7*, 955. [CrossRef]
52. Pan, J.S.; Hu, P.; Chu, S.C. Novel Parallel Heterogeneous Meta-Heuristic and Its Communication Strategies for the Prediction of Wind Power. *Processes* **2019**, *7*, 845. [CrossRef]
53. Yin, X.; Liu, X.; Sun, M.; Ren, Q. Novel Numerical Spiking Neural P Systems with a Variable Consumption Strategy. *Processes* **2021**, *9*, 549. [CrossRef]

54. de la Cruz, R.T.A.; Cabarle, F.G.C.; Macababayao, I.C.H.; Adorna, H.N.; Zeng, X. Homogeneous spiking neural P systems with structural plasticity. *J. Membr. Comput.* **2021**. [CrossRef]
55. Spiess, R.; George, R.; Cook, M.; Diehl, P.U. Structural plasticity denoises responses and improves learning speed. *Front. Comput. Neurosci.* **2016**, *10*, 93. [CrossRef] [PubMed]