Implementing Enzymatic Numerical P Systems for AI Applications by Means of Graphic Processing Units

Manuel García–Quismondo, Luis F. Macías–Ramos, and Mario J. Pérez–Jiménez

Abstract. A P system represents a distributed and parallel computing model in which basic data structures are, for instance, multisets and strings. Enzymatic Numerical P Systems (ENPS) are a type of P systems whose basic data structures are sets of numerical variables. Separately, GPGPU (general-purpose computing on graphics processing units) is a novel technological paradigm which focuses on the development of tools for graphic cards to solve general purpose problems. This paper proposes an ENPS simulator based on GPUs and presents general concepts about its design and some future ideas and perspectives.

Introduction

Membrane computing is a bio-inspired branch of natural computing, abstracting computing models from the structure and functioning of living cells and from the organization of cells in tissues or other higher order structures [29]. This branch of natural computing studies the design and properties of membrane systems or *P systems*. P systems are non–deterministic distributed and parallel computing models structured in compartments known as *membranes*. Basic data structures such as multisets, strings or numerical variables [30] are associated with membranes. According to the way in which membranes are structured, there are several types of P systems. For instance, there exist cell-like P systems [29], tissue P systems [26] and spiking neural P systems [17], along with other types. In P systems, membranes and their associated data structures are processed by means of rewriting rules or programs associated to the cells, in order to perform sequences of configurations (*computations*) [29][30]. P systems have been successfully applied in a wide range of domains [4]. For instance, they have been applied in microbiological modelling

Manuel García–Quismondo · Luis F. Macías–Ramos · Mario J. Pérez–Jiménez
Research Group on Natural Computing, Dpt. of Computer Science and Artificial Intelligence, University of Sevilla, Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
e-mail: `{mgarciaquismondo,lfmaciasr,marper}@us.es`

-

in order to model phenomena such as *quorum sensing* in *Vibrio fischeri* populations [38] and ecological modelling to predict the evolution of the bearded vulture [3] and the Pyrenean chamois [9] populations in the Catalan Pyrenees, as well as image thresholding [7]. Such a versatility makes P systems a useful tool for gaining knowledge about a vast variety of different domains, thus providing a promising tool whithin the range of disciplines which composes the field of study of artificial intelligence.

A special type or cell-like P systems are Enzimatic Numerical P Systems (ENPSs) [36]. ENPSs describe a deterministic, maximally–parallel model in which the basic data structures associated to membranes are numerical values which evolve by means of *programs* associated to them [30]. In order for a program to be applied, a certain value of a specific variables (enzyme) may be needed. Otherwhise, the program cannot be applied [36]. This model of computation has already been successfully used in model robot controllers, in which a robot needs to avoid obstacles situated in a closed circuit [37].

Separately, *GPGPU* (general-purpose computing on graphics processing units) is a novel technological discipline which consists of the application of graphic cards (GPUs) in order to perform parallel, distributed algorithms [40]. The basic idea is to take advantage of the parallel architecture of GPUs, traditionally used for graphics processing, to execute algorithms which can be performed in parallel, thus accelerating these algorithms by dividing them in concurrent tasks and executing these tasks in a parallel mode.

In this paper, we propose a GPU simulator for ENPSs. The parallel architecture of ENPS makes the simulations of their computations a suitable task to be parallelized, thus expecting an acceleration in the simulation times if compared to their sequential counterparts.

This paper is structured as follows. Section 10.2.2 provides a quick introduction to Numerical P Systems (NPSs) as a model of computation. Section 10.3 describes ENPSs as an extension of NPSs. Section 10.4 provides a general overview of the current state-of-the-art about the results obtained by previous GPU simulators within the field of membrane computing. Finally, section 10.7 presents the conclusions obtained and proposes some directions for future work.

2 Preliminaries

2.1 P Systems

Membrane Computing is a young and emergent branch of Natural Computing introduced by G. Păun [30]. It has received important attention from the scientific community since then, with contributions by computer scientists, biologists, formal linguists and complexity theoreticians, enriching each others with results, open problems and promising new research lines. In fact, membrane computing was selected by the Institute for Scientific Information, USA, as a fast *Emerging Research Front* in computer science, and [35] was mentioned in [42] as a highly cited paper

in October 2003. This new model of computation starts from the observation that the cell is the smallest living thing, and at the same time it is a marvellous tiny machinery, with a complex structure, and from the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. The challenge is to take the cell itself as a support for computations, to find in the structure and the functioning of the cell seen as a whole those elements useful for computations. Computations in general, at the mathematical level, but with the hope to bring something useful to practical computing, either in the same style as genetic algorithms and neural computing, of improving the use of the existing computers, or proposing new types of electronic computers, or, possibly to lead to ways to use the cells themselves as computing supports. The devices of this model are called P systems. Roughly speaking, a P system consists of a cell-like membrane structure, in the compartments of which one places multisets of objects which evolve according to given rules.

The main syntactic ingredients of a cell-like membrane system are the membrane structure, the multisets of objects, and the evolution rules. A membrane structure consists of several membranes arranged in a hierarchical structure inside a main membrane (the skin), and delimiting regions (the space in–between a membrane and the immediately inner membranes, if any). Each membrane identifies a region inside the system Regions defined by a membrane structure contain objects corresponding to chemical substances present in the compartments of a cell. The objects can be described by symbols or by strings of symbols, in such a way that multiset of objects are placed in regions of the membrane structure. The objects can evolve according to given evolution rules, associated with the regions (hence, with the membranes).

The semantics of the cell-like membrane systems is defined through a non deterministic and synchronous model (in the sense that a global clock is assumed) as follows: A configuration of a cell–like membrane system consists of a membrane structure and a family of multisets of objects associated with each region of the structure. At the beginning, there is a configuration called the initial configuration of the system. In each time unit we can transform a given configuration in another configuration by applying the evolution rules to the objects placed inside the regions of the configurations, in a non-deterministic, and maximally parallel manner (the rules are chosen in a non-deterministic way, and in each region all objects that can evolve must do it). In this way, we get transitions from one configuration of the system to the next one.

In the last years, many different models of P systems have been proposed. In particular, computational devices inspired from the cell inter–communication in tissues, and adding the ingredient of cell division rules of the same form as in cell–like membrane systems with active membranes, but without using polarizations. In these systems, the rules are used in the non-deterministic maximally parallel way, but we suppose that when a cell is divided, its interaction with other cells or with the environment is blocked; that is, if a division rule is used for dividing a cell, then this cell does not participate in any other rule, for division or communication. The set of communication rules implicitely provides the graph associated with the system through the labels of the membranes. The cells obtained by division have the same

labels as the mother cell, hence the rules to be used for evolving them or their objects are inherited.

The idea of spiking neurons, currently an active research topic in neural computing (see, e.g., [16], [22], [23]), was recently incorporated in membrane computing (see [18]) – the resulting formal systems are called *spiking neural P systems*, abbreviated as SN P systems. The structure of an SN P system has a form of a directed graph with nodes representing neurons, and edges representing synapses. The neurons contain *spikes*, objects of a unique type. A neuron (node) sends signals (spikes) along its outgoing synapses (edges). Each neuron has its own rules for either sending spikes (firing rules) or for internally consuming spikes (forgetting rules). the rules of the first type consume some spikes and produce a new spike, which is sent to all neurons linked by a synapse to the neuron where the rule was used, while the forgetting rules just remove spikes from neurons. In the initial configuration a neuron stores the initial number of spikes, and at any time moment the currently stored number of spikes (*current contents*) is determined by the initial contents and the history of functioning of σ (the spikes it received from other neurons, the spikes it sent out, and the spikes it internally consumed/forgot). One of the neurons is the *output* one, and its spikes can also exit into the environment, thus providing a trace of the system evolution. Like in neurobiology, we call this trace – sequence of moments when a spike exits the system – *spike train*.

2.2 Numerical P Systems

As years went by, different types of P systems have been introduced. In the foundational transition P system model, the cell structure consists of a rooted tree, in which each node represents a membrane of the structure. Edges represent the hierarchical relationships between membranes existent in the structure. However, some models propose new types of cell structures. For instance, SN P Systems describe an architecture based on a directed graph, in which cells or *neurons* act as nodes, whereas firing rules act as arcs. These rules send information from one neuron to another after a specific amount of time or delay [19]. Similarly, in Tissue P systems, instead of a hierarchical structure, membranes are placed at the nodes of a non-directed graph. The edges of the graph represent symport/antiport rules which communicate the membranes in the graph, thus moving objects across membranes [26]. Also, even variants of these ones have evolved. For instance, in the case of SN P Systems, new features such as SN P Systems with several kinds of spikes [19] and SN P systems with neuron division and budding [27]. As regards to Tissue P Systems, there exist Tissue P Systems with cell division [33], Tissue P Systems without environment [8] as an example.

Besides, not only have membrane structures evolved across the Membrane Computing literature. The data structures which evolve by means of applications of rules through computation steps have also been affected. As a proof of that, in String P Systems sets of strings are considered instead of multisets of objects. These strings are rewritten by means of rewriting–like rules on each computation step [6].

Following this trend, a new kind of P system was introduced by Gheorghe and Andrei Păun in 2006. In these P systems, known as Numerical P Systems (*NPSs* [29]), the traditional multisets of objects associated to membranes are replaced by sets of numerical variables. These variables evolve by means of programs associated to the membranes. As in the foundational model, the membrane structure is a tree-nested hierarchy, so no new membrane architecture is introduced in this model.

A numerical P system of degree $m \geq 1$ is a tuple:

$$\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \dots (Var_m, Pr_m, Var_m(0)))$$

where:

- H is an **alphabet** with m symbols used as labels of the m membranes of the system. The labels contained in H are the labels of the membranes in Π.
- μ is a **membrane structure**, a rooted tree with m membranes.
- $Var_i = \{x_{1,i} \dots x_{k_i,i}\}$ is the set finite of **variables** associated with compartment i, $(1 \leq i \leq m)$
- $Var_i(0) = \{\lambda_{1,i} \dots \lambda_{k_i,i}\}$ are numerical values (*real numbers*) for the variables in Var_i. These values are considered as initial values; at time = of the system evolution we have $x_{j,i} = \lambda j, i(1 \leq i \leq m, 1 \leq j \leq k_i)$.
- $Pr_i = Pr_{1,i} \dots Pr_{q_i,i}$ is the set of programs from comparment i of $\mu(1 \leq i \leq m)$. The *l-th* program $Pr_{l,i}$ from compartment i is of the form $Pr_{l,i} = (F_{l,i}(x_{1,i}, \dots, x_{k_i,i}), c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$ where $F_{l,i}(x_{1,i}, \dots, x_{k_i,i})$ is the *l-th* **production function** from compartment i and $c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$ describes the **repartition protocol**.

The production function $F_{l,i}(x_{1,i}, \dots, x_{k_i,i})$ from compartment i is a a real function having as variables those from this compartment. The expresion $c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i}$ describes the repartition protocol which has the following meaning: let $v_1 \dots v_{n_i}$ be the set of variables from compartment i, from the parent membrane of i and for all compartments corresponding to children of comparment i. The coefficients $c_{l,1} \dots + c_{l,n_i}$ are natural numbers that specify the proportion of tehe current production distributed to each variable $v_1 \dots v_{n_i}$.

More precisely, at any instant $t \geq 0$, a program $Pr_{l,i}$ on each set Pr_i $(1 \leq i \leq m)$ is non–deterministically chosen. Then, we compute $F_{l,i}(x_{1,i}(t), \dots, x_{k_i,i}(t))$ and $C_{l,i} = \sum_{j=1}^{n_i} c_{l,j}$. The values of all variables on which $F_{l,i}$ depends are *consumed* and reset to 0. The value $q = \frac{F_{l,i}(x_{1,i}(t), \dots, x_{k_i,i}(t))}{C_{l,i}}$ represents the "unitary portion" to be distributed to variables $v_1, \dots, v_{n_i}$, according to coefficients $c_{l,i}, \dots, c_{l,n_i}$ in order to obtain the values of these variables at time $t+1$. Specifically, variable $v_{l,j}$ will receive $q \times c_{l,j}(1 \leq j \leq n_i)$ from compartment i. If a variable receives such "contributions" from several neighbouring compartments, then they are added in order to produce the value of the variable at time $t+1$.

This model of computation was initially aimed to capture the nature and behaviour of economic processes [29]. There had been some previous works on the modelling of economic processes by means of Membrane Computing [34], and this work proposed some research lines on the application of NPSs for the modelling of economic phenomena.

3 Enzymatic Numerical P Systems

3.1 Description of Enzymatic Numerical P Systems

As it is usual on membrane computing models, a new kind of P systems has risen as an extension of NPSs. This model is known as Enzymatic Numerical P Systems (*ENPSs*). Although this parallel model of computation has many points in common with Numerical P Systems, there are some aspects which differenciates both models. This way, in contrast to Numerical P Systems, Enzymatic Numerical P Systems describe a deterministic model of computation. Thus, instead of non–deterministically chosen, the programs to be applied are controlled by specific variables known as *enzyme–like* variables.

An Enzymatic Numerical P System of degree $m \geq 1$ is a tuple:

$$\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \ldots (Var_m, Pr_m, Var_m(0)))$$

where:

- H, μ and $(Var_1, Var_1(0)) \ldots (Var_m, Var_m(0))$ have the same meaning than in Numerical P Systems described in section 10.2.2.
- Pr_i is the set of programs associated to membrane i. Each l-th program in set Pr_i may have one of the following forms:

  - $Pr_{l,i} = (F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}), c_{l,1}|v_1 + \ldots + c_{l,n_i}|v_{n_i})$
  - $Pr_{l,i} = (F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}), (e_{l,i} \rightarrow), c_{l,1}|v_1 + \ldots + c_{l,n_i}|v_{n_i})$

  In both forms, all values which also appear in section 10.2.2 have the same meaning, with $e_{l,i}$ being a variable in Var_i. This variable is known as the *enzyme–like* variable associated to $Pr_{l,i}$ and its value cannot be consumed by this program. Enzyme–like variables are exclusive ingredients of ENPSs. That is, they do not appear in NPSs.

The main novelty introduced by ENPSs has to do with the use of enzyme–like variables to control the execution flow of programs. This way, each program may have an associated enzyme–like variable which controls its application. If a program is to be applied at time t, then this program is *active* at this time. On each computation step, all active programs in each membrane are applied in parallel. Programs in ENPSs are applied the same way than in NPSs. However, a program is active only in the following cases:

- The program does not have an associated enzyme.
- The program has an associated enzyme and the value of this enzyme is greater than the minimum of the values of the variables consumed by the program.

ENPSs have been successfully applied within the field of robotics. For instance, they have been used to model deterministic mobile robot controllers for obstacle avoidance. In this model, the speed of the two robot motors is set according to the values assigned to two variables of the system. Thus, the dynamical evolution of these variables describes the behavior of the robot through a closed circuit [37]. More information about ENPSs can be found in [36][37].

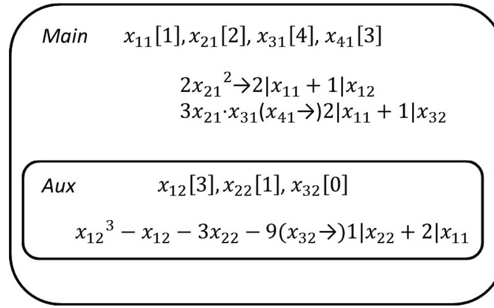

Fig. 1 Enzymatic Numerical P System

3.2 ENPSs and Artificial Intelligence

Mobile robot control problems, such as obstacle avoidance and odometric localizacion, can be considered as artificial intelligence problems. For instance, obstacle avoidance can be considered as a high-level planning problem [21]. In obstacle avoidance, the objective is to find a sequence of movements in a static or dynamical environment. The objective of this sequence is for robots which follow it to avoid crashing with any obstacles they might find in the environment. The input data is given as a series of sensor lectures obtained from the environment. This type of path planning problems arising from the field of robotics has already been attacked by using artificial intelligence techniques such as ant colony algorithms [12][11].

Odometric localization is a widely used method for estimation of the momentary pose of a mobile robot with respect to its starting pose [20]. This estimation is affected by several error sources, such as imprecission in the mobile robot kinematic parameters and errors in the sensor lectures [1]. Thus, odometric localization entails an optimization problem, i.e., minimizing the global error in the pose estimation. As an optimization problem, odometric localization has been previously tackled by using well-known artificial intelligence paradigms, such as genetic algorithms [15] and artificial neural networks [10]. All in all, ENPs propose a new framework which can be applied in order to solve artificial intelligence problems arising from robotics [37].

3.3 Simulation of ENPSs

ENPSs describe a parallel model. Therefore, the huge computational power required by extensive models (for instance, those necessary for massive robot swarms and robots with complex sensor networks) accounts for the need for high performance computing platforms to simulate them. Besides, their parallel structure makes them appropriate to be simulated by means of parallel architectures such as GPUs, FPGAs and computer clusters.

4 The Compute Unified Device Architecture (CUDA) Standard for GPU Computing

4.1 Outline of the CUDA Programming Model

Modern GPUs consist of a large number or processing units. For instance, Fermi cards contain up to 448 processor cores and 1.536 processing units per core, thus resulting in a total number of $448 \times 1.536 = 688.128$ threads [41]. These threads are executed in parallel with a certain degree of dependency from each other [40].

In order to make the most of this massively parallel architecture, it is necessary to make use of standards specifically designed for these devices. Two of these main standards in GPGPU are OpenCL [39] and CUDA [41].

The CUDA programming model is an abstract GPU model provided by NVIDIA. This model is an abstraction of the specific parallel device where the program is to be executed. The model defines a *grid*. This grid is an abstraction of the current GPU card where the code is to be executed. The grid is composed of multiprocessing computing devices known as *blocks*. Similarly, each block is composed of several stream monoprocessing units known as *threads*(see figure 10.2). Threads execute parallel pieces of code or *kernels*. On any instant in the execution of a GPU program, the same kernel is run on every thread at the same time.

It is convenient to batch threads which perform operations in common in the same block. The reason is that threads in the same block can communicate with each other through fast on-chip memory, whereas threads in different blocks use slow off-chip memory to communicate. Thus, it is important to minimize the communication between threads from different blocks, turning it into communication between threads in the same block when possible. Besides, they are allowed to synchronize with each other via barriers. On the other hand, the only way of synchronizing threads of different blocks is by ending the kernel execution. The CUDA programming model requires thread blocks in the same kernel to be independent. It means that the final result of the computation cannot depend on the order in which the blocks are executed, giving the same result without depending on their order of execution.

4.2 The CUDA–C *Programming Language*

CUDA–C is an extension of the C language to work against the CUDA programming model. This language is designed to make the most of the GPGPU approach by enabling programmers to encode parallel applications to be run on GPUs [5]. That is, programmers are able to develop code to be executed on each GPU thread at the same time. This way they can take advantage of the GPU parallel architecture in order to obtain enormous speed-up if compared to sequential versions of the same code.

The structure of CUDA-C programs consists of two main parts: The *host* part and the *device* part. The main difference between them consists of the specific device in which they are executed. Thus, the host part is executed on the CPU, whilst the device part is executed on the GPU [5]. The host part includes calls to kernels. The

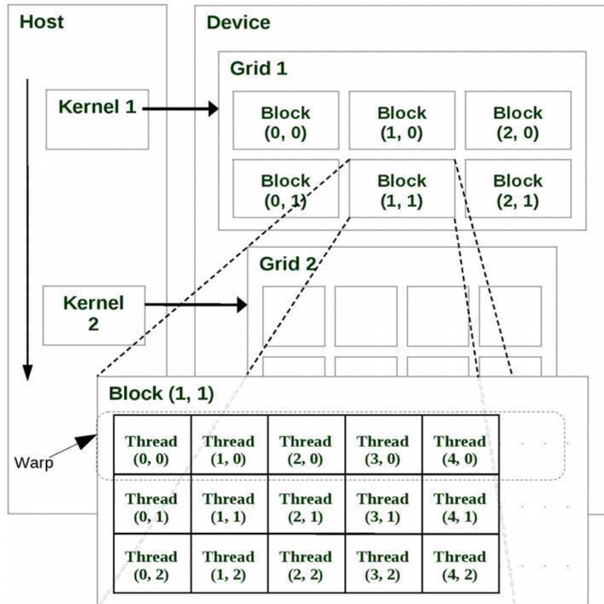

Fig. 2 The CUDA programming model

device part is composed of kernels which define the operations to be performed in parallel. The developer organizes the threads to execute the kernels in two hierarchical levels of parallelism. These levels are a reflection of those in which the CUDA programming model is organized. In order the organize the threads to execute the kernels, the programmer defines the structure of the thread blocks. This is done by programmatically setting the number of threads per block, as well as the total number of blocks in the grid. This way, both parts of the program can cooperate in order to obtain a global result. More information about the CUDA programming model and the CUDA-C language can be found on [41][24].

A sample code of a typical high-performance operation on GPU can be found on [4]. In this sample, the summing of the elements in two vectors is computed. Each pair of elements are assigned to a different thread. Therefore, each pair of elements are added in a parallel way. Although this example may seem too simple, it illustrates quite well the way in which the CUDA parallel mode can be applied to parallelize operations, thus obtaining a tremendous speed-up due to the parallel computing approach.

GPGPU and CUDA–C have been already successfully applied in order to simulate different kinds of P systems. To the best of our knowledge, they have been applied to simulate cell-like object-based P systems [5] and SN P systems [2]. Their results include data which show noticeable speed-ups in comparison to their sequential counterparts. These results demonstrate the suitability of the GPGPU approach for simulating P systems in a parallel mode.

```
//Vector size in elements
const int N = 1048576;
//Vector size in bytes
const int dataSize = N * sizeof(float);

//CPU memory allocation
float *h_A = (float *)malloc(dataSize);
float *h_B = (float *)malloc(dataSize);
float *h_C = (float *)malloc(dataSize);

//GPU memory allocation
float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, dataSize));
cudaMalloc((void **)&d_B, dataSize));
cudaMalloc((void **)&d_C, dataSize));

//Initialize h_A[], h_B[]...

//Copy input data to GPU for processing
cudaMemcpy(d_A, h_A, dataSize, cudaMemcpyHostToDevice) );
cudaMemcpy(d_B, h_B, dataSize, cudaMemcpyHostToDevice) );

//Run the core of N / 256 units,256 streams each
//Assuming that N is multiple of 256
vectorAdd<<<N / 256, 256>>>(d_C, d_A, d_B);

//Read GPU results
cudaMemcpy(h C, d C, dataSize, cudaMemcpyDeviceToHost) );
```

Fig. 3 A sample of CUDA–C host code

5 A GPU–Based Simulator for Enzymatic Numerical P Systems

Taking into account these previous results, we propose a new simulator for ENPSs developed on CUDA-C. Thus, we expect our simulator to achieve an important acceleration of the execution times in comparison to currently existent ENPS sequential simulators [36].

In this section, the guidelines for the design of the current version of the simulator are outlined. Moreover, a thorough description of the data structures and the functioning of the simulator is explained.

The objective of the proposed ENPS GPU-based simulator is to fully simulate the behaviour of enzymatic numerical P systems, performing operations in parallel whenever possible. In order to do that, it is crucial to identify which operations are susceptible for parallelization and write parallel kernels for them. This way the simulator can take advantage of the underlying parallel architecture.

5.1 Data Representation

As it is usual in GPU computing [4], the data handled by the simulator is stored by means of arrays. The simulator uses three different kinds of arrays, according to the nature of the information stored in them:

Program arrays: These arrays are used to store the programs of the ENPS model simulated. These arrays can be organized in three different types:

- **Production function:** These arrays are used to store the information regarding the production functions of rules. They are described in subsection 10.5.4.1 in detail.
- **Repartition protocol:** These arrays are used to store the information about the repartition protocols. They are described in subsection 10.5.6 in detail.
- **Enzymes:** Each program in the simulated ENPS model has an associated position in this array. This position contains the index of the enzymatic variable

associated to the program. In the case that the program is in non-enzymatic form, the position contains a specific marker, such as -1.

Variables: This array stores the value of the variables associated to the compartments in the ENPS model simulated. These values evolve as programs in the model are applied.

Auxiliary data: These arrays store the auxiliary data needed in order to check and apply the programs. Specifically, these arrays are:

Minimum values: This array stores the minimum values of the variables consumed by programs

Production function results: This array stores the results of the calculations of the applied production functions

Program applications: This array stores themarkers to set if programs are applied. These markers can be *Active* or *Inactive*.

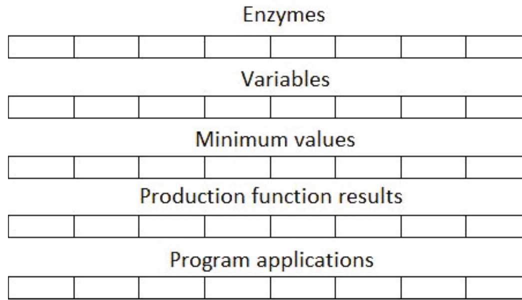

Fig. 4 Arrays used by the simulator to store general information

5.2 Repartition Coefficients Normalization

This step is only taken once, as it is a pre–processing operation in order to improve the efficiency of the simulator. It is not performed in parallel, thus being part of the *host* code. For each repartition protocol in each program $P_{l,i}$, each coefficient $c_{l,s}$ associated to the repartition protocol in is replaced by $\frac{c_{l,s}}{\sum_{j=1}^{n_i} c_{l,j}}$ $(1 \leq s \leq n_i)$. Although the formal definition of both NPSs and ENPSs establishes that $c_{l,s} \in \mathbb{N}(1 \leq s \leq n_i)$, these new repartition coefficients are real numbers, as they are temporary values calculated in order to improve the efficiency of the simulation algorithm.

5.3 Program Checking

The first stage of the implemented algorithm consists of selecting which programs can be applied on the current step of computation. Therefore, for each program, one should distinguish two different cases:

- The program is in non-enzymatic form. This means that no enzyme–like variable is associated to the program. In this case, the program is always executed. In such a case, the program's associated position in *Enzymes* should be a specific marker, as shown in subsection 10.5.1. Hence, the program's associated position in *Program applications* is set to *Active*.
- The program is in enzymatic form. This means that an enzyme–like variable is associated to the program. In this case, the simulator needs to find the minimum value of the variables consumed by the program. Then, it is necessary to calculate the minimum value of all variables consumed by the program, in order to distinguish two different cases:

  – The value of this minimum is greater than or equal to the value of the associated enzyme–like variable. In this case, the program cannot be applied.
  – The value of this minimum is lower than the value of the associated enzyme-like variable. In this case, the program has to be applied.

  In terms of implementation, each thread has an associated index i in the production function arrays (see subsection 10.5.4 for more details). Thus, each thread has an associated program. Each thread whose associated program is in enzymatic form performs the following steps:

  – Step through the region of the arrays in subsection 10.5.4 associated to the production function and checking those positions in which the array *Production function node types* contains the value **variable**.
  – Check the value of the array *Production function variables* in these positions.
  – Use the value of these positions on each thread as indexes to access the array *Variables*.
  – Calculate the minimum of the positions in this array.
  – Compare this minimum to the value of the enzyme–like variable of its associated program.
    · If the value of the enzyme–like variable is greater, then the program's associated position in *Program applications* is set to *Active*.
    · If the value of the enzyme–like variable is lower or equal, the program's associated position in *Program applications* is set to *Inactive*.

5.4 Calculation of Production Functions

In this section, an outline of the performing of the calculation of production functions is presented. For doing so, firstly the data structures used to represent production functions are introduced. Secondly, the way in which these data structures are processed is described. Finally, a brief discussion about the expected speed–up factor ends this subsection.

5.4.1 Structural Design of Production Functions

In the presented simulator, production functions are represented as tree–like structures. In these tree–like structures there exist two different kinds of nodes:

Non-leaf nodes: These nodes represent binary operations. On these nodes, the operands could be constants, variables or the result of other operations.

Leaf nodes: These nodes represent constants or variables. In the case of variables, their value is the evaluation of its represented variable.

Production functions are implemented by means of five different arrays. Thus, each tree representing a production function is implemented as a region in these five arrays. Each node is implemented as a position in all these arrays. These arrays are described as follows:

Production function node types: For each node, this array denotes the type of the node. It could be *constant* or *variable* (leaf nodes) or anyone of the *operation* type (non-leaf nodes). Each node of the *operation* type tells the simulator to perform a binary operation on its children. The values for nodes of the *operation* type are:

> **Add:** Add the value of the left child to the value of the right child.
> **Subtract:** Subtract the value of the left child to the value of the right child.
> **Multiply:** Multiply the value of the left child by the value of the right child.
> **Divide:** Divide the value of the left child by the value of the right child.
> **Power:** Power the value of the left child to the value of the right child.

Production function left offsets: Given a position in this array, if its corresponding position in *Production function node types* is equal to *constant* or *variable* then this position has no meaning. Otherwise, if its corresponding position in *Production function node types* is equal to anyone of the *operation* type, then this position contains the relative offset where the **left** operand of the represented node is stored.

Production function right offsets: Given a position in this array, if its corresponding position in *Production function node types* is equal to *constant* or *variable* then this position has no meaning. Otherwise, if its corresponding position in *Production function node types* is equal to anyone of the *operation* type, then this position contains the relative offset where the **right** operand of the represented node is stored.

Production function constants: Given a position in this array, if its corresponding position in *Production function node types* is equal to *variable* or anyone of the *operation* type then this position has no meaning. Otherwise, if its corresponding position in *Production function node types* is equal to **constant**, then this position contains the value of the constant represented by its node.

Production function variables: Given a position in this array, if its corresponding position in *Production function node types* is equal to *constant* or *operation* then this position has no meaning. Otherwise, if its corresponding position in

Production function node types is equal to **variable**, then this position contains the position in *Variables* of the variable represented by its node.

This way, one can associate an index i in all of these arrays to every node N in every production function tree–like structure.

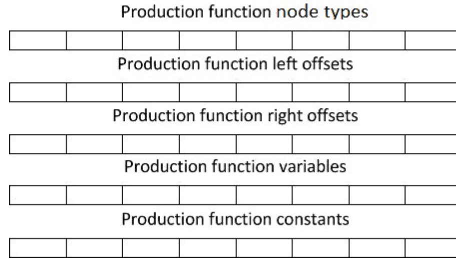

Fig. 5 Data structure for production functions

5.4.2 Functional Design of Production Functions

By making use of the data structures described above, calculating a production function can be simply reduced to stepping through the nodes of its representing tree. Then, the recursive algorithm used to calculate production functions is:

1. Given a node N, check the node type of N. In terms of implementation, it means checking its associated position i in *Production function node types* as listed above.
    a. If the node type of N is *constant*, then return the value of the constant associated to N. In terms of implementation, it means returning the value in position i in the array *Production function constants*.
    b. If the node type of N is *variable*, then return the value of the variable asociated to N. In terms of implementation, it means taking the value stored in position i in the array *Production function variables* and using this value j as an index to return position j in the array *Variables* described in section 10.5.1.
    c. If the node type of N is anyone of the *operator* type, then:
        i. Access position i in *Production function left offsets*. Let j be the content of this position.
        ii. Calculate the result of $i+j$. Let $k=i+j$.
        iii. Process the node N^l whose index is k. It means going back to step 1, but processing N^l instead of N.
        iv. Access position i in *Production function right offsets*. Let m be the content of this position.
        v. Calculate the result of $i+m$. Let $n=i+m$.
        vi. Process the node N^r whose index is n. It means going back to step 1, but processing N^r instead of N.

vii. Apply the operation indicated by position i in *Production function operators* to the result of processing N^l as left child and the result of processing N^r as right child.
viii. Return the result of this operation.

d. Store the result of the calculation in the program's associated position o in *Production function results*.

This algorithm is executed by each of the threads of the kernels, if and only if their associated program is active. Thus, in this version of the simulator the theoretical speed–up factor on the calculation of production functions is equal to the number of programs of the simulated model. One could argue that some operations in these tree steppings could be performed in parallel, thus improving the theoretical speed–up factor. For, instance, in the production function represented in figure 10.6, $x_{1,2}+7$ and $x_{1,4}-3$ can be performed in parallel.

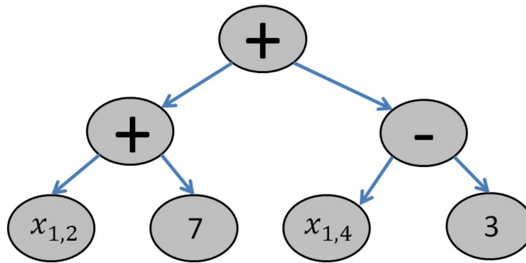

Fig. 6 Tree representing the production function $(x_{1,2}+7)+(x_{1,4}-3)$

However, the vast variety of different cases which can be found in these complex production functions makes it really difficult to design an efficient parallel implementation of their binary operations. Besides, production functions are usually short (though sometimes complex) functions [36], so the speed–up factor gain which can be obtained may not be worth increasing the design complexity of the simulator so much.

5.5 *Variable Clearing*

After calculating the result of the production functions of the applied programs, the next step on the algorithm is to clear the values of the variables on which these production functions depend. In practical terms, it means setting the value of these variables to 0. For doing so, each thread on the simulator has an associated production function element. In terms of implementation, each thread whose has an associated index i in the production function arrays. Thus, on every thread, the following operations are performed:

- Check its position i in *Program applications*. If the value of this position is *Inactive*, do not execute the following steps and exit the kernel. If the value of this position is *Active*, execute the following steps.
- Check if its position i in *Production function node types* is equal to *Variable*. In other case, abort the thread.
- Access its position i in *Production function variables*. Let j be the value of this coefficient.
- Set position j in *Variables* to 0.

5.6 Repartition Protocol Application

The last step in the algorithm consists on distributing the result of the production functions. For each thread, this implies reading the value stored in *Production function results* and distributing it over its program's contributed variables. As the normalization of coefficients is performed at the beginning of the algorithm, this step only entails multiplying this read value by the associated coefficient of each variable in the repartition protocol and adding the result of the multiplication to this variable. Before explaining in detail the implementation of this process, it is important to introduce the data structures used to represent the repartition protocols of the simulated model. Each repartition protocol is stored as a region in two arrays. Thus, each pair *coefficient–variable* has an associated index, which corresponds to an associated position in each of these arrays.

Repartition protocol coefficients: This array contains the coefficients associated to each variable existing in repartition protocols. On the repartition protocol step, the content of this array is already normalized, as it is performed at the beginning of the algorithm (see subsection 10.5.2).

Repartition protocol variables: This array contains the indexes of the variables to which the repartition protocols are contributed. These indexes are used to access the array *Variables*, in order to obtain their current value.

In terms of implementation, the distribution of the result of the production function of each program is performed the following way. Each thread has an associated pair *coefficient-variable* assigned. In terms of implementation, a position i in the repartition protocol arrays is asigned to each thread. Taking into account this consideration, each thread performs the following operations:

1. Check its position i in *Program applications*. If the value of this position is *Inactive*, do not execute the following steps and exit the kernel. If the value of this position is *Active*, execute the following steps.
2. Access its position i in *Repartition protocol coefficients*. Let c be the value of this coefficient.
3. Access the position o of the program of its repartition protocot in *Production function results*. Let f be the value of this result.
4. Perform the multiplcation of these values. Let $m = c \times f$.

5. Access its position *i* in *Repartition protocol variables*. Let *v* be the value of this position.
6. Add *m* to position *v* in *Variables*.

It is important to notice that, in this step, the theoretical speed-up factor can be greater than 1, in the case that there exist programs in the simulated model in which the number of pairs *coefficient–variable* is greater than 1. In contrast to the case of production functions, this is usual in the studied models [37], so a greater theoretical speed-up factor can be obtained in this stage.

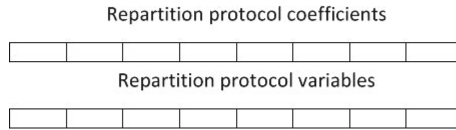

Fig. 7 Data structure for production functions

5.7 Execution of a Simulation Step

As described in the former subsections, the execution of a simulation step consists of the checking and application of programs for a predefined number of steps. This number of steps, as well as the model to simulate, are specified as inputs to the simulator. In the case that the model simulated defines a number of steps, then this number prevails over the one given as input. The simulation of a model is performed by executing the following steps:

1. Normalize the repartition coefficients, as described in subsection 10.5.2.
2. For each simulation step, perform the following operations:

   a. Assign a program to each thread. This is done by using the indexes of the threads in the *CUDA* programming model.
   b. Each thread checks if its program is to be applied, as described in subsection 10.5.3.
   c. If its program is to be executed, each thread calculates its production function, as described in subsection 10.5.4.2.
   d. If its program is to be executed, each thread clears the values of those variables which depend on the production function of the program (that is, consumes its values), as described in subsection 10.5.5.
   e. Assign a pair *coefficient–variable* from each repartition protocol to each thread. This is also done by using the indexes of the threads in the *CUDA* programming model.
   f. If its repartition protocol's program is to be executed, each thread distributes the result of the corresponding production function according to the associated pair, as described in subsection 10.5.6.

5.8 Remarks on the Simulator

This simulator will be published under open source license. It can be used for simulating complex distributed processes modelled with ENPSs. Therefore, several robot behaviors can be simulated in parallel (for example, a robot could avoid obstacles, follow another robot or look for a target at the same time). The synchronization at the same time between several behaviors of one robot is done by the help of the enzyme variables which can be used as stop conditions [37]. Apart from simulating several behaviors for only one robot in parallel, the simulator could be used to simulate interaction and cooperation between several robots in complex distributed robotic systems.

6 Simulator Performance

6.1 Simulator Workflow

In order to ease the simulation of ENPS models, the simulator takes an input file describing an ENPS in XML format. The XML format used is the one accepted by SNUPS [36], a previously existent sequential simulator for ENPSs. This way, the reusability of the models is improved, as the same file can be used with independence to the selected simulator, be it SNUPS and on the GPU-based one introduced, without any change in the XML file format. Hence, there is no need to change the file format, in the case that the same ENPS is to be simulated on both simulators.

Thus, in order to simulate an ENPS, one needs to encode it on the same XML format as it is required on SNUPS. Once this P system is encoded, the resulting file can be parsed by the GPU simulator. After the parsing process, the simulation is performed. Eventually, the information is displayed on the command prompt. Figure 10.6.1 shows a graphical representation of this process.

6.2 Performance Comparison

All parallel parts of the algorithm are executed with a degree of parallelism at least equal to the number of programs of the simulated model. The degree of parallelism can be even greater when the repartition protocol stage is applied. Hence, a theoretical acceleration of at least the number of programs of the model could be reached, if compared to the runtime of sequential simulators. In real terms, the simulator was tested by using an ENPS model of obstacle avoidance [37] as an example. These models were simulated by using SNUPS [36]. Then, the resulting runtimes were compared with the GPU simulator runtimes, in order to get an approximate speed-up. In the specific case of the obstacle avoidance model, the total number of programs is 41 [37]. Hence, an acceleration of at least 41 is theoretically expected in this case, if compared to sequential ENPSs simulators [36].

The novelty of ENPSs as a computing model [36] accounts for the need to generate *ad-hoc* case studies for the simulator. That is, it is not possible to find an

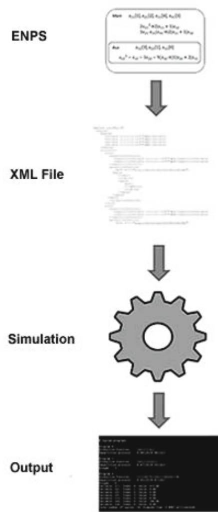

Fig. 8 Workflow of the simulator

extensive collection of case studies in the literature. Thus, the authors needed to generate them in order to measure the experimental performance of the GPU simulator proposed. In practice, the simulator performance has been tested by using an obstacle avoidance model [37]. Taking this model as a starting point, some case studies have been generated. All of them share the same programs, variables and membrane structure of the obstacle avoidance model proposed in [37]. Thus, these models consist of 9 membranes, 41 programs and 29 variables each [37]. The only differences between these case studies consist of the initial values of the variables associated to the membranes.

Model number	SNUPS	GPU	Acceleration
1	36.3702	6.7286	5.4053
2	14.9084	6.6304	2.2484
3	14.9040	7.7268	1.9288
4	26.3204	6.8255	3.8561
5	15.2276	6.4188	2.3723
6	18.9548	6.5659	2.8868
7	30.7377	6.7206	4.5736
8	27.0497	7.6020	3.5582
9	15.7529	6.8335	2.3052
10	30.1695	6.6364	4.5460

Fig. 9 Comparison of execution times for a sequential ENPSs simulator (SNUPS) and the GPU ENPSs simulator proposed

For this purpose, 10 randomly generated models were executed. Each model was executed for 100 steps. Figure 10.9 displays the execution times for these runs. This table compares the execution times for the same models run on SNUPS [36] and the GPU simulator. The execution times are given in milliseconds. For each model, the acceleration is given as the result of the division $\frac{SNUPS\ runtime}{GPU\ runtime}$.

7 Conclusions

In this paper, a GPU-based simulator for ENPSs. ENPSs describe a parallel computing model with applications in artificial intelligence. This simulator might be suitable for large scale models which can be applied within the field of robotics.

The massively parallel environment provided by the GPUs is suitable for ENPSs simulations. Following this line of work, it would be interesting to simulate these models by means of GPU clusters or other parallel architectures (such as FPGAs or computer clusters). These systems might be applied to model the behavior of massive robot swarms and complex sensor networks.

ENPSs can be used to model different behaviours, such as *follow the leader*, *obstacle avoidance* and *wall following* (cf. Chap. 9 of this book). The resulting simulators could be compared in terms of execution time and performance. This comparison could help experts select the most suitable simulator for the task in hand, be it *wall following*, *obstacle avoidance*, etc.

Another interesting challenge concerning the parallel simulation of ENPS models has to do about exploring the possibility of simulating several robot behaviors in parallel on GPUs. That is, simulating situations in which robots need to achieve more than one objective at the same time. These simulations could help to recreate scenarios in which robots need to perform multi-objective tasks.

Another important open problem concers the integration of the simulator into user-oriented software platforms. This integration will ease the use of the simulator by Membrane Computing experts, thus improving the human-computer interaction experience. Some examples of end-user software frameworks for simulating P systems are SNUPS[36] and P-Lingua[13].

Another important point with which to deal has to do with a more exhaustive evaluation of the performance of the simulator. Whilst the shallow performance evaluation included in this paper shows an average speed-up factor of 3x if compared to the *Java* simulator SNUPS, in order to assess the real speed–up factor to be reached by the simulator it is necessary to develop larger models and compare their runtimes not only with *Java* or other virtual machine-based programming languages, but also with languages on a lower level of abstraction, such as C or Fortran.

Nevertheless, the current models have such a small number of programs that these low-level simulators could yield better runtimes than the GPU simulator, as they are free from the overhead regarding the distribution of tasks among the GPU threads. In other words, the GPU simulator is expected to yield a better performance only when the number of programs is considerably high, that is, about thousands of programs per model. In order to asses the performance in these cases, it is necessary to extend

the models currently found in the literature up to new models with thousands of programs. On these models, the GPU simulator is expected to yield lower execution times not only compared to SNUPS execution times, but also to the times obtained by C and Fortran simulators.

Acknowledgements. Manuel García-Quismondo, Mario J. Pérez-Jiménez and Luis F. Macías Ramos are supported by project TIN 2009-13192 from "Ministerio de Ciencia e Innovación" of Spain, co-financed by FEDER funds. Manuel García-Quismondo and Mario J. Pérez-Jiménez are also supported by "Proyecto de Excelencia con Investigador de Reconocida Valía P08-TIC-04200" from Junta de Andalucía. Manuel García-Quismondo is also supported by the National FPU Grant Programme from the Spanish Ministry of Education. Mario J. Pérez-Jiménez is also supported by project TIN2008-04487-E from the Spanish Ministry of Science and Education through the Complementary Action TIN2008-0448-E/TIN.

References

1. Antonelli, G., Chiaverini, S., Fusco, G.: A calibration method for odometry of mobile robots based on the least-square technique: Theory and experimental validation. IEEE Transactions on Robotics 21(5), 994–1004 (2005)
2. Cabarle, F.G.C., Adorna, H., Martínez, M.A.: A Spiking Neural P System Simulator Based on CUDA. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) CMC 2011. LNCS, vol. 7184, pp. 87–103. Springer, Heidelberg (2012)
3. Cardona, M., Colomer, M.A., Pérez-Jiménez, M.J., Sanuy, D., Margalida, A.: Modeling Ecosystems Using P Systems: The Bearded Vulture, a Case Study. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC9 2008. LNCS, vol. 5391, pp. 137–156. Springer, Heidelberg (2009)
4. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with Active Membranes on CUDA. Briefings in Bioinformatics 11(3), 313–322 (2010)
5. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. The Journal of Logic and Algebraic Programming 79, 317–325 (2010)
6. Ceterchi, R., Mutyam, M., Păun, G., Subramanian, K.G.: Array-rewriting P systems. Natural Computing 2(3), 229–249 (2003)
7. Christinal, H.A., Díaz-Perni, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Thresholding of 2D Images with Cell-like P Systems. Romanian Journal of Information Science and Technology (ROMJIST) 13(2), 131–140 (2010)
8. Christinal, H.A., Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Tissue-like P Systems Without Environment. In: Martínez del Amor, M.A., Păun, G., Pérez-Hurtado, I., Riscos, A. (eds.) Proceedings of the Eighth Brainstorming Week on Membrane Computing, pp. 53–64. Fenix Editora (2010)
9. Colomer, M.A., Lavín, S., Marco, I., Margalida, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D., Serrano, E., Valencia-Cabrera, L.: Modeling Population Growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by Using P-Systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) CMC 2010. LNCS, vol. 6501, pp. 144–159. Springer, Heidelberg (2010)
10. Conforth, M., Meng, Y.: An Artificial Neural Network Based Learning Method for Mobile Robot Localization. Robotics Automation and Control 6 (2008)

11. Dong, J., Liu, B., Peng, K., Yin, Y.: Robot Obstacle Avoidance based on an Improved Ant Colony Algorithm. In: Zhou, S.M., Wang, W. (eds.) Proceedings of WRI Global Congress on Intelligent Systems (GCIS 2009), pp. 103–106. IEEE Computer Society (2009)
12. Du, R., Zhang, X., Chen, C., Guan, X.: Path Planning with Obstacle Avoidance in PEGs: Ant Colony Optimization Method. In: Zhu, P. (ed.) International Conference on Cyber, Physical and Social Computing (CPSCom), pp. 768–773. IEEE Computer Society (2010)
13. García-Quismondo, M., Gutiérrez-Escudero, R., Martínez-del-Amor, M.A., Orejuela-Pinedo, E., Pérez-Hurtado, I.: P-Lingua 2.0: A software framework for cell-like P systems. International Journal of Computers, Communications and Control 4(3), 234–243 (2009)
14. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with CUDA. IEEE Micro 28(4), 13–27 (2008)
15. Gill, M.A.C., Zomaya, A.Y.: Genetic algorithms for robot control. In: IEEE International Conference on Evolutionary Computation, p. 462. IEEE Computer Society (1996)
16. Gerstner, W., Kistler, W.: Spiking Neuron Models. Single Neurons, Populations, Plasticity. Cambridge Univ. Press (2002)
17. Ibarra, O., Pérez-Jimenez, M.J., Yokomori, T.: On spiking neural P systems. Natural Computing 9(2), 475–491 (2010)
18. Ionescu, M., Paun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae 71 (2-3), 279–308 (2006)
19. Ionescu, M., Paun, G., Pérez–Jiménez, M.J., Rodríguez-Patón, A.: Spiking Neural P systems with several types of spikes. International Journal of Computers, Communications & Control 4(4), 648–656 (2011)
20. Ivankjo, E., Komšić, I., Petrović, I.: Simple Off-Line Odometry Calibration of Differential Drive Mobile Robots. In: Proceedings of 16th International Workshop on Robotics in Alpe-Adria-Danube Region - RAAD, pp. 164–169 (2007)
21. Khatib, O.: Real-time obstacle avoidance for manipulators and mobile robots. The International Journal of Robotics Research 5(1), 90–98 (1986)
22. Maass, W.: Computing with spikes. Special Issue on Foundations of Information Processing of TELEMATIK 8(1), 32–36 (2002)
23. Maass, W., Bishop, C. (eds.): Pulsed Neural Networks. MIT Press, Cambridge (1999)
24. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (2008)
25. NVIDIA. NVIDIA CUDA Programming Guide 2.0 (2008)
26. Pan, L., Pérez-Jiménez, M.J.: Computational complexity of tissue-like P systems. Journal of Complexity 26(3), 296–315 (2010)
27. Pan, L., Paun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China. Information Sciences. 54(8), 1596–1607 (2011)
28. Pan, L., Wang, J., Hoogeboom, H.J.: Asynchronous Extended Spiking Neural P Systems with Astrocytes. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) CMC 2011. LNCS, vol. 7184, pp. 243–256. Springer, Heidelberg (2012)
29. Paun, G., Paun, R.: Membrane Computing and Economics: Numerical P Systems. Fundamenta Informaticae 73(1-2), 213–227 (2006)
30. Paun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
31. Paun, G.: Membrane Computing. An Introduction. Springer, Heidelberg (2002)

32. Paun, G.: Computing with Membranes. Turku Centre for Computer Science, Turku, Finland, vol. 208 (1998)
33. Paun, G., Pérez-Jiménez, M.J., Riscos, A.: Tissue P systems with cell division. International Journal of Computers, Communications & Control 3(3), 295–303 (2008)
34. Paun, G., Paun, R.: Membrane computing as a framework for modeling economic processes. In: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005). IEEE Computer Society, Washington DC (2006)
35. Păun, A., Păun, G.: The power of communication: P systems with symport/antiport. New Generation Computing 20(3), 295–305 (2002)
36. Pavel, A., Arsene, O., Buiu, C.: Enzymatic Numerical P Systems - A New Class of Membrane Computing Systems. In: Proceedings 2010 IEEE Fifth International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA 2010), pp. 1331–1336. IEEE Computer Society, Liverpool (2010)
37. Pavel, A., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing (2011) (in press)
38. Romero, F.J., Pérez-Jiménez, M.J.: A model of the Quorum Sensing System in Vibrio Fischeri using P systems. Artificial Life 14(1), 95–109 (2008)
39. Takizawa, H., Koyama, K., Sato, K., Komatsu, K., Kobayashi, H.: CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In: International Parallel and Distributed Processing Symposium (IPDPS 2011), pp. 864–876. IEEE Computer Society, Anchorage (2011)
40. `http://www.gpgpu.org`
41. `http://www.nvidia.com/object/cuda_home_new.html`
42. ISI web page, `http://esi-topics.com/erf/october2003.html`