Simulation of P systems with active membranes on CUDA

José M. Cecilia, José M. García, Ginés D. Guerrero, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado and Mario J. Pérez-Jiménez

Abstract

P systems or Membrane Systems provide a high-level computational modelling framework that combines the structure and dynamic aspects of biological systems in a relevant and understandable way. They are inherently parallel and non-deterministic computing devices. In this article, we discuss the motivation, design principles and key of the implementation of a simulator for the class of recognizer P systems with active membranes running on a (GPU). We compare our parallel simulator for GPUs to the simulator developed for a single central processing unit (CPU), showing that GPUs are better suited than CPUs to simulate P systems due to their highly parallel nature.

Keywords: *natural computing; membrane computing; P Systems; parallel computing; GPU; CUDA*

INTRODUCTION

Membrane Computing is an emergent branch of Natural Computing introduced by G. Pãun a decade ago [1]. This new model of computation starts from the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this model are called P systems [2]. In essence, a P system consists of a cell-like membrane structure, in which multisets of objects can be placed, i.e. sets of objects with multiplicities associated with the elements. They have several *syntactic* ingredients (Figure 1): a *membrane structure* consisting of a hierarchical arrangement of membranes embedded in a *skin* membrane, and delimiting *regions* or compartments where multisets of *objects* and sets of evolution *rules* are placed.

P systems also have two main *semantic* ingredients: their inherent *parallelism* and *non-determinism*. The objects inside the membranes can evolve according to given rules in a synchronous (in the sense that a global clock is assumed), parallel, and non-deterministic way. Computation of P system is a (finite or infinite) sequence of instantaneous transitions between *configurations,* as shown in Figure 1.

Corresponding author. José M. Cecilia, DITEC, Facultad de Informática, Universidad de Murcia, Campus de Espinardo 30100, Murcia, Spain. Tel: +34 868 88 7656; Fax: +34 868 88 4151; E-mail: chema@ditec.um.es

José M. Cecilia is a member of the Parallel Computer Architecture Research Group at the University of Murcia. His current research interests are in evaluating the newest generation of accelerators for general purpose applications, and also the use of GPUs for general purpose applications in different areas.

José M. García is the Head of the Research Group on Parallel Computer Architecture and a professor in the Department of Computer Engineering at the University of Murcia. His current research interests are high-performance coherence protocols for Chip Multiprocessors (CMPs) and shared-memory multiprocessor systems, high-speed interconnection networks and the use of GPUs for general-purpose applications. He is a member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation.

Ginés D. Guerrero is a member of the Parallel Computer Architecture Research Group at the University of Murcia. His main research interest is in designing general purpose applications for the graphics processing unit.

Miguel A. Martínez-del-Amor is a member of the Research Group on Natural Computing at the University of Seville. His main research interest is to join membrane computing and High Performance Computing using efficient computer simulators.

Ignacio Pérez-Hurtado is an Associate professor in the Department of Computer Science and Artificial Intelligence at the University of Seville, and a member of the Research Group on Natural Computing. His main research interests are computer simulation and models for biological processes within membrane computing.

Mario J. Pérez-Jiménez is the Head of the Research Group on Natural Computing and a professor in the Department of Computer Science and Artificial Intelligence at the University of Seville. His main research fields are computational complexity theory, natural computing, membrane computing, bioinformatics, and computational modelling for systems biology and ecosystems.

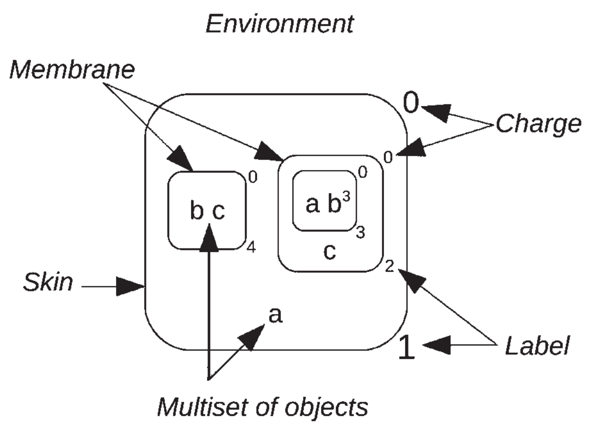


Figure 1: Structure and main components of a P system.

Computation always starts with an *initial configuration* of the system, where the input data is encoded. The transition from one configuration to the next is performed by applying rules to the objects inside the regions (the rules to be used and the objects to evolve are chosen randomly). Whenever it is not possible to apply many rules to the existing objects and membranes of a given configuration, the computation halts (then, the configuration is called a *halting computation*). The result of a halting computation of the system is encoded by the multiset associated with a specific output membrane (or alternatively the *environment*) in the last configuration.

It is noteworthy that here we have double parallelism, one at the level of each region (the rules are used in parallel way), and other at the level of the system (all regions evolve concurrently). This parallelism and non-determinism can be used to solve computationally hard problems; however, we must point out two considerations. On one hand, we have to deal with non-determinism in such a way that the solutions obtained from these devices must be algorithmic solutions in classic sense. On the other hand, the drastic decrease of the execution time from an exponential to a polynomial one is not achieved free, but by the use of an exponential workspace (in the form of membranes and objects), though this space is created in polynomial (often linear) time.

In order to solve decision problems (abstract problems that require a *yes* or *no* answer), we consider *recognizer P systems*; i.e. P systems such that: (i) the alphabet of objects contains two distinguished elements: *yes* and *no*; (ii) all computations halt; and (iii) if C is a computation of the system, then either object *yes* or object *no* (but not both) must be sent to the output region of the system in the last step of the computation and never in any previous step.

Although most researches in P systems concentrates on the computational power and efficiency of the devices involved, lately they have been used to model biological phenomena within the framework of Computational Systems Biology. P systems provide a modelling approach to biological systems fulfilling the requirements of a good modelling framework: relevance, understandability, extensibility and computational/mathematical tractability. In this case, P systems are not used as a computing paradigm, but rather as formalism for describing the behaviour of the system to be modelled. Several P system models have been proposed to describe oscillatory systems [3], signal transduction [4, 5], gene regulation control [6], quorum sensing [7–9] and metapopulations [10]. These models differ from each other in type of rewriting rules, membrane structure and the strategy applied to run the rules in the compartments defined by membranes. Some of these models using *metabolic algorithm* [11], *dynamical probabilistic P systems* [10] and *(multicompartmental) Gillespie Algorithm* [8] were applied in various case studies. Furthermore, (*probabilistic*) P systems have also been successfully applied as a tool for macroscopic level processes, as the computational modelling of real ecosystems [12].

P systems based models are more useful than other classical modelling approaches, such as Ordinary Differential Equations, because P systems can be used when classical approaches fail to specify and simulate biological phenomena, for instance, when chemical concentrations do not vary continuously over time in a deterministic way [8].

In order to validate a P system based model experimentally, it is necessary to have simulators able to be executed on electronic computers. They would help researchers to compute, analyse and extract results from a model [13]. These simulators have to be as efficient as possible to handle instances of large size. This is one of the main problems with current simulators for P systems.

Software applications for Membrane Computing normally implement sequential (or parallel with relatively few threads) simulation algorithms adapted to common central processing unit (CPU) architectures [13], so they lack the possibility of exploiting the massively parallel nature that P systems present by their definition.

This parallel computation model leads us to look for a highly parallel computational technology where a parallel simulator can run efficiently. The newest generations of graphics processor units (GPUs) are massively parallel processors which can support several thousand concurrent threads. To date, many general purpose applications have been ported to these platforms obtaining good speedups compared to their corresponding sequential versions [14–18]. Current Nvidia GPUs, for example, contain up to 240 scalar processing elements per chip [19], and they are programmed using C programming language extensions called CUDA (Compute Unified Device Architecture) [16, 19, 20].

In this article, we try to highlight the necessity to use a parallel architecture which improves the efficiency of P systems simulators designed to model biological processes. For this purpose, we present a parallel simulator for the class of recognizer P systems with active membranes using CUDA, due to the fact that in this theoretical model, the creation of an exponential number of membranes and objects takes place in a natural way.

The simulator receives as input a P system which is defined and translated into a binary file using the P-Lingua programming language [21]. The simulation algorithm is divided into two main stages: *Selection stage* and *Execution stage*. Both phases are implemented on the GPU, so the entire simulation executes all the computations in different membranes in a parallel way.

We test the simulator with a P system which exploits the intrinsic parallelism of P systems and demonstrate that GPUs are better suited than CPUs to simulate P systems as long as the problem size increases.

The article also describes the model of recognizer P systems with active membranes, and introduces the Compute Unified Device Architecture (CUDA) and some concepts of programming on GPUs, (both conceptual aspects and technical details). Finally, we conclude the article highlighting the main ideas presented, and also some directions for future work.

P SYSTEMS WITH ACTIVE MEMBRANES

Biological membranes are not completely passive. Passing of a chemical compound through a membrane is often by direct interaction with the membrane itself. During this interaction, the chemical compound which passes through the membrane as well as the membrane itself can be modified. These ideas were captured in [1] considering P systems with active membranes where the central role in the computation is played by the membranes. Each membrane is supposed to have an electrical polarization (we will say charge), one of the three possible: positive, negative and neutral. This kind of P systems with a probabilistic semantic has been used to model real ecosystems based on scavenger birds in the Catalan Pyrenees [12].

A P system with active membranes is a tuple of the form $\prod = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$, where $m \geq 1$ is the initial degree of the system; O is the alphabet of objects; H is a finite set of labels for membranes; μ is a membrane structure (a rooted tree), consisting of m membranes injectively labelled with elements of H; $\omega_1, \ldots, \omega_m$ are strings over O, describing the multisets of objects placed in the m regions of μ; and R is a finite set of rules, where each rule is of one of the following forms:

(i) $[a \to v]_h^\alpha$ where $h \in H$, $\alpha \in \{+, -, 0\}$ *(electrical charges)*, $a \in O$ and v is a string over O. They are associated with membranes and depending on the label and the charge of the membranes, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them *(evolution rules)*.

(ii) $a[\,]_h^\alpha \to [b]_h^\beta$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$. An object is introduced in the membrane, possibly modified during this process; the polarization of the membrane can also be modified, but not its label (*send-in communication rules*).

(iii) $[a]_h^\alpha \to [\,]_h^\beta b$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$. An object is sent out the membrane, possibly modified during this process; the polarization of the membrane can also be modified, but not its label *(send-out communication rules)*.

(iv) $[a]_h^\alpha \to b$ where $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in O$. In reaction to an object, a membrane can be dissolved, while the object specified in the rule can be modified (*dissolution rules*).

Rules are applied in a maximal parallel way. In one step, each object in a membrane can only be used by at most one rule, but any object which can evolve by a rule must do it. However, rules (ii) to (iv) cannot be applied simultaneously in a membrane in one

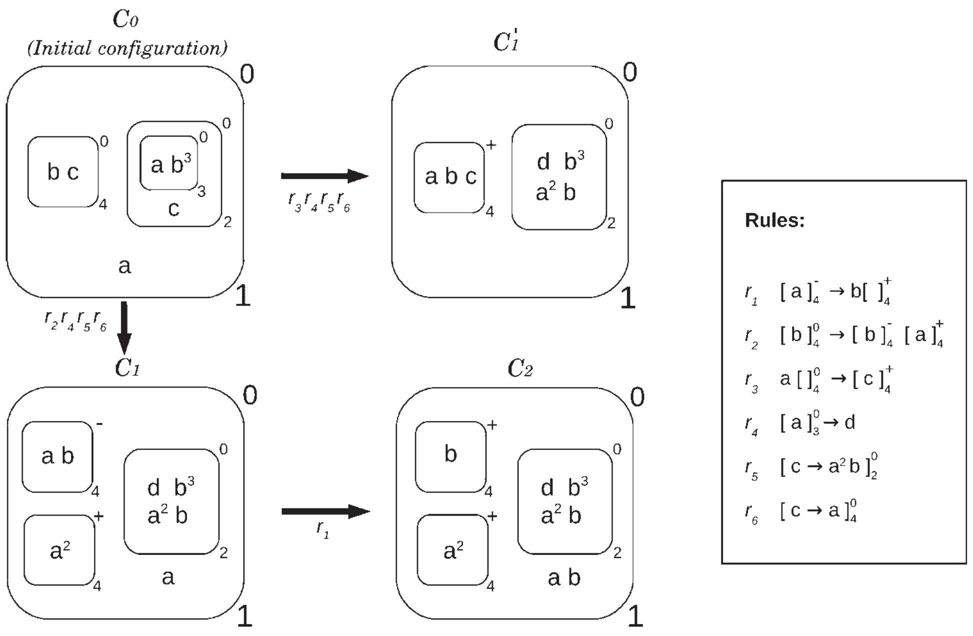


Figure 2: Example of a P system computation.

computation step. Moreover, rules associated with label *h* are used for all membranes with this label, and all the objects which are not involved in any of the operations to be applied remain unchanged. Finally, the skin is never dissolved.

One of the most important roles of cells is reproduction, and this is achieved through the division of a cell into two identical copies. The biological term for this process is called *mitosis*, and, in fact, it consists of a sequence of several phases. By division we can obtain 2^n cells in *n* steps, which look very attractive from a computational efficiency point of view. Bearing in mind that many reactions which take place in a cell are related to membranes, rules for membrane division are considered. This kind of P system is able to efficiently solve computationally hard problems making use of an exponential workspace created in a natural way by division rules. Hence, the simulation of these P systems using conventional software is a good challenge.

This idea can be formalised through the following rule:

(v) $[a]^{\alpha}_h \to [b]^{\beta}_h [c]^{\gamma}_h$ where $h \in H$, $\alpha, \beta, \gamma \in \{+,-,0\}$, $a, b, c \in O$. In reaction to an object, the membrane is divided into two membranes with the same label, but possibly different polarizations; the object specified in the rule is replaced in the two new membranes by possibly new objects *(division rules).*

Rules (ii) to (v) cannot be applied simultaneously in a membrane in one computation step. The skin is never divided.

Figure 2 shows an example of a simple P system and its computation, in order to help the understanding of the dynamics of the model (see chapter 7 in [26]).

PARALLEL COMPUTING ON GPUS

Graphics coprocessors were designed to process voluminous and repetitive calculations and render smooth and realistic-looking images on computer screens. They ease the computational burden on the CPU by handling the calculations and other simple, highly repetitive operations necessary for rendering the lines, polygons, and surfaces of full-motion graphics scenes, for example.

With the growing demand for more realistic computer games (the major force driving GPU evolution) a GPU can deliver hundreds of billion of operations per second (some GPUs more than a teraflop, or a trillion operations per second).

In 2002, Mark Harris, now a computer researcher with Nvidia, coined the term GPGPU [17] for 'general purpose computation on GPUs'.

In mid-2007, Nvidia consolidated this trend and introduced the Compute Unified Device Architecture (CUDA). The CUDA is composed as both, hardware and software architecture for issuing and managing computations on their most recent GPU families (G80 family onward), making it operate as a truly generic parallel computing device.

From the hardware point of view, the GPU device is a scalable processor array consisting of a set of SIMT (Single Instruction Multiple Threads) multiprocessors (SM), each of them containing several stream processors (SPs) as it is showed in Figure 3. Different memory spaces are available in the GPU. The global memory (also called device memory) is the only space accessible by all multiprocessors, acting as the main device memory with the largest capacity in the GPU. Each multiprocessor has its own private memory space called shared memory. The shared memory is smaller and also lower access latency than global memory. Finally, there are other addressing spaces for specific purpose: texture and constant memory [19,20,22,23,25].

Figure 3 shows the Nvidia GPU called Tesla C1060 which has been used for this work. It contains 30 multiprocessors, and each of them consists of 8 processors (240 processors in total). As for memory, Tesla C1060 contains 4 GB of global memory and 16 kB of shared memory per SM. Table 1 shows the Tesla C1060 features.

A parallel program in the CUDA programming model is similar to a program in another sequential language (like Fortran or C), but it has two different parts or codes: a sequential code (*host* code) executed by the CPU, and a parallel code (device code or kernel) executed by the GPU. The host code is mainly responsible of transferring data between main memory and global memory, and also setting the kernel parameters, such as the number of blocks per grid and the number of threads per block, as well as invoking the device code.

The device code is grouped into one or more program routines called kernels, named from the host code as if they were procedures or objects in C/C++. A kernel is a piece of code programmed in a SPMD (Single Program, Multiple Data) style, i.e. the same code is executed over different data

Table 1: Major hardware and software features of Tesla C1060

Tesla parameters	Value
SM	30
SP/SM	8
32-bit registers/SM	16 384
Shared memory/SM	16 KB
Threads/SM	1024
Threads/Block	512
Threads/Warp	32
Device memory	4 GB

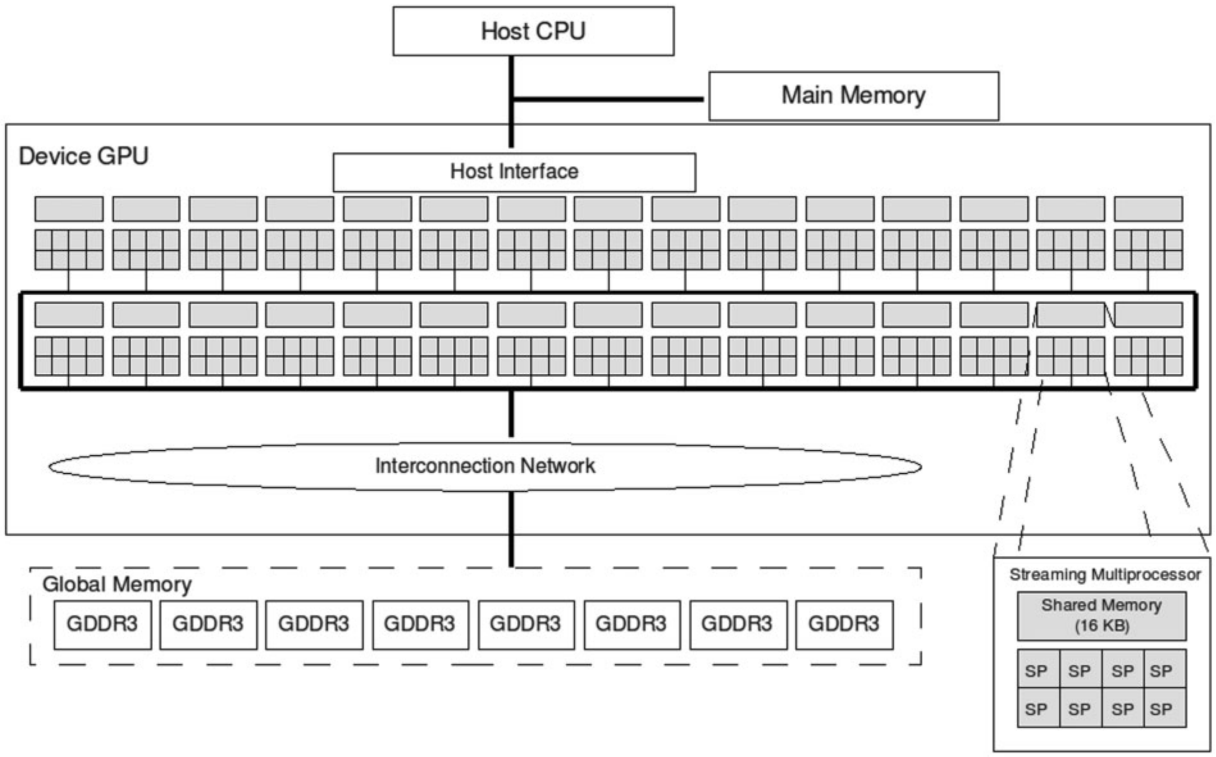


Figure 3: Tesla C1060 GPU with 240 SPs (Streaming Processors) organized in 30 SMs (Streaming Multiprocessors).

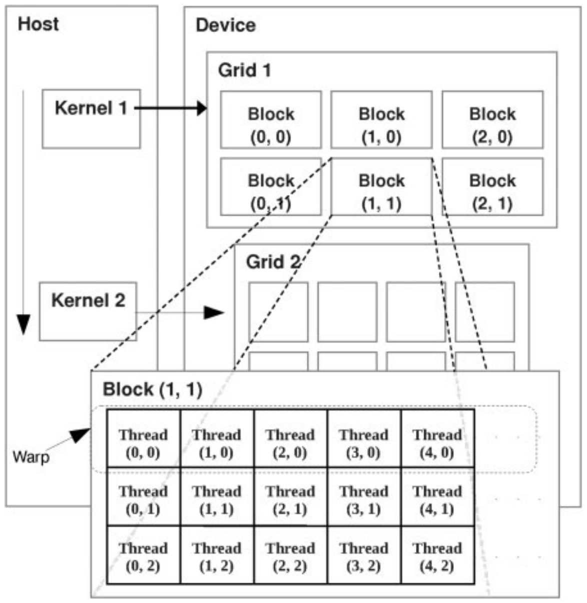


Figure 4: CUDA execution model.

by different threads on different SP cores. The kernel computation is performed by all these threads running in parallel.

Figure 4 shows the CUDA execution model [19] which is based on a hierarchy of abstraction layers: grids, blocks, warps and threads. The thread is the basic execution unit that is mapped to a single SP. A block is a batch of threads which can cooperate together because they are assigned to the same multiprocessor, and therefore they share all the resources included in this multiprocessor, such as register file and shared memory. A grid is composed of several blocks which are equally distributed and scheduled among all multiprocessors, since there are normally more blocks than multiprocessors. Finally, threads included in a block are divided into batches of 32 threads called warps. The warp is the scheduled unit, so the threads of the same block are scheduled in a given multiprocessor warp by warp.

The programmer declares the number of blocks, the number of threads per block and their distribution, which can be declared in one, two or three dimensions (see values on Table 1). Each block, and also each thread, has its own and unique identifier (thread id and block id). These allow the programmer to select different data and code depending on the thread id and block id.

Memory accesses and synchronization scheme are other important aspects in the CUDA programming model. The latency of access to each memory included in the GPU can be reduced if the memory access follows the correct pattern [16]. Global synchronization is not provided at the device side, only threads in a block can wait for each other. Hence, block synchronization mechanisms must be explicitly implemented by the host through consecutive kernel invocations.

SIMULATING P SYSTEMS WITH ACTIVE MEMBRANES

The simulator we have developed is based on the sequential simulator for P systems with active membranes developed in P-Lingua [24]. In this design, the simulation process is divided into two stages: *selection stage* and *execution stage*. The selection stage consists of the search for the rules to be executed in each membrane in a given configuration. The rules selected are executed at the execution stage.

At the end of the execution stage, the simulation process restarts the selection stage in an iterative way until a halting configuration is reached. This stop condition is 2-fold: a certain number of iterations or a final configuration is reached. On one hand, at the beginning of the simulation, we define the maximum number of iterations. On the other hand, a halting configuration is obtained when there are no more rules to select at selection stage. As previously explained, the halting configuration is always reached since it is a simulator for recognizer P systems.

The input data for the selection stage consists of a description of the membranes with their multisets (strings over the working alphabet O, labels associated with the membrane in H, etc.) and the set of rules R to be selected. The output data of this stage is the set of selected rules per membrane which will be executed on the execution stage.

The execution stage applies the rules previously selected on the selection stage. During the execution stage, membranes can vary by including new objects, dissolving membranes, dividing membranes, etc, obtaining a new configuration of the simulated P system. This new configuration will be the input data for the selection stage of the next iteration.

PARALLEL SIMULATOR OF P SYSTEMS ON THE GPU

Figure 5 shows the basic design of the simulator that we have implemented on the GPU.

We have developed five kernels to implement the selection and execution stages. The first kernel

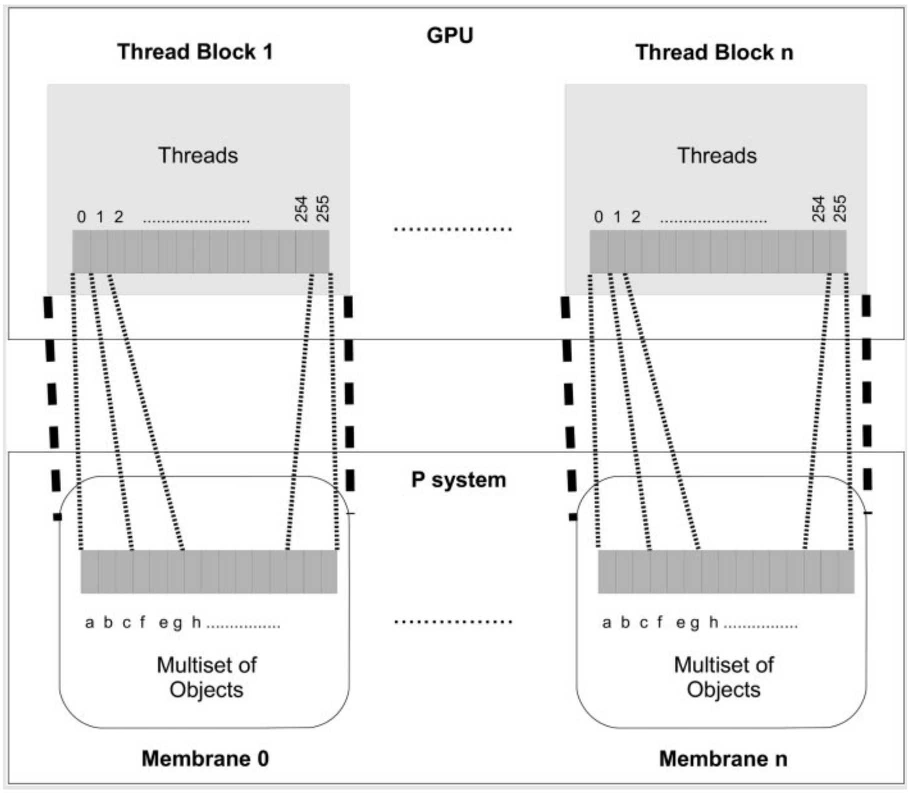


Figure 5: Structure of our parallel simulator on the GPU.

implements the selection stage and also the execution stage for evolution rules. The other four kernels implement the other execution rules (dissolution, division, send-out and send-in rules).

The selection kernel starts with the selection stage, which maps each membrane to a block, where each thread represents an object of the alphabet *O*. Each block runs in parallel looking for the set of rules that have to be selected for its membrane, and each individual thread is responsible for identifying if there are some rules to be executed associated with the object represented on the left hand side. After the selection stage, we also execute in this kernel the evolution rules. These rules are executed inside this kernel for three main reasons: the evolution rules do not imply communication (and therefore, synchronization) among membranes; they are executed in a maximal way, and this decision allows us to use less global memory because it is not necessary to store the selected evolution rules for the execution stage.

The rest of the rules to be applied are executed in four different kernels, one kernel per each kind of rule (dissolution, division, send-out, send-in).

Algorithm 1 shows the pseudo-code of the simulator. First of all, we move the data needed for the computation to the GPU. Then, the code calls the selection kernel which returns the selected rules for the current configuration of the P system. Among the possible selected rules there will be different kinds of rules to be executed, therefore, we identify the type of those rules in order to launch only the kernels which are needed to complete the execution stage. As we explained before, we iterate on this process until the maximum number of steps is reached or the system returns an answer. Finally, we copy back the result data to CPU.

Our simulator presents two restrictions, due to some peculiarities in the CUDA programming model: it can handle only two levels of membrane hierarchy for simplicity (the skin and the rest of elementary membranes), and the number of objects in the alphabet must be divisible by a number smaller than 512 (the maximum number of threads per thread block), in order to distribute the objects among the threads equally.

DESIGNING A CASE OF STUDY FOR P SYSTEMS

In order to evaluate the performance of the simulator, we have designed a family of P systems, named

Algorithm 1 Parallel simulator of P systems on the GPU

```
1: $CopyDataFromCPUtoGPU(membranes)$
2: $CopyDataFromCPUtoGPU(rules)$
3: while NOT $maxStep$ AND NOT $finalConfiguration$ do
4:    $kernelSelection(rules, membranes, selectedRules)$
5:    if $DISSOLUTION \in selectedRules$ then
6:       $kernelDissolution(rules, membranes, selectedRules)$
7:    else if $DIVISION \in selectedRules$ then
8:       $kernelDivision(rules, membranes, selectedRules)$
9:    else if $SEND\_OUT \in selectedRules$ then
10:      $kernelSendOut(rules, membranes, selectedRules)$
11:   else if $SEND\_IN \in selectedRules$ then
12:      $kernelSendIn(rules, membranes, selectedRules)$
13:   end if
14: end while
15: $CopyDataFromGPUtoCPU(membranes)$
```

Algorithm 1: Parallel simulator of P systems on the GPU.

test P system, where it is easy to vary the number of membranes as well as the number of objects. This test P system also fits the behaviour of the GPU since only evolution and division rules are defined (without communication and dissolution rules), and every object in every membrane will evolve according to a given rule. The defined P system is of the following form $\prod = (O, H, \mu, \omega_1, \omega_2, R)$, where: $O = \{d, o_i / 0 \leq i < n\}$, $H = \{1,2\}$, $\mu = [[\,]_2]_1$, $\omega_1 = \emptyset$, $\omega_2 = O$, $R =$

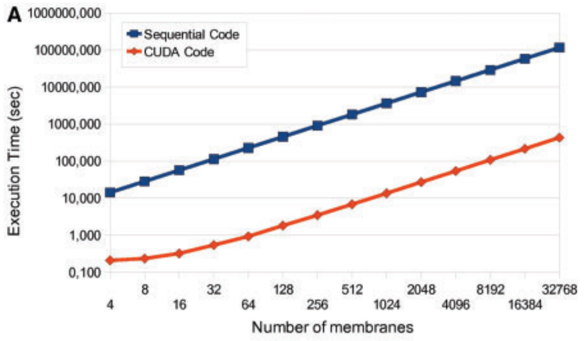
(i) Evolution rules: $[o_i \rightarrow o_i]_2^0, 0 \leq i < n$
(ii) Division rule: $[d]_2^0 \rightarrow [d]_2^0 [d]_2^0$

Thus, the test P system allows us to take control of the number of objects in the system by modifying the n parameter. Furthermore, the number of rules changes along with the number of objects, and the number of membranes in every step of the computation is equal to 2^s, where s is the step number. Lastly, the number of evolution rules selected and executed per membrane in every step is invariable, since they are defined one per object and all the objects of the alphabet are presented in every membrane labelled with 2.

EXPERIMENTAL RESULTS

Figure 6 presents the results we have obtained for the simulator between a sequential version developed in the C++ language and our simulator developed in CUDA. Notice that in both graphs the Y-axis is represented in an exponential form.

For our tests, we use two benchmarks based on the test P system explained in the previous section.



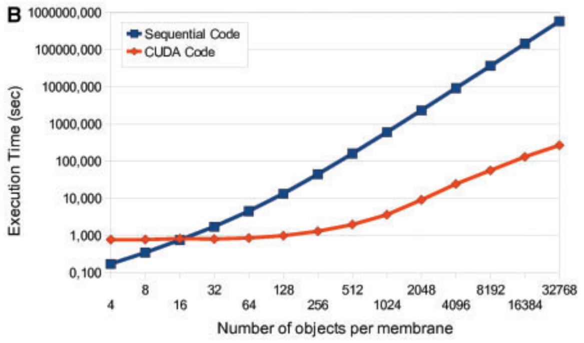


Figure 6: Test P system results for both sequential and CUDA simulator. (**A**) By number of membranes and (**B**) by number of objects.

These benchmarks cover both ways of parallelism that P systems naturally have by its definition. The first one tests the parallelism between membranes, increasing the number of membranes exponentially, and the second one tests the parallelism between objects increasing the number of objects within each membrane exponentially.

Figure 6a shows the results for the benchmark which increases the number of membranes exponentially, having a fixed number of objects per membrane (2560 objects). The CPU simulator increases its time exponentially from the beginning (with four membranes) until reaching the final configuration (with 32768 membranes). Our CUDA simulator, which assigns 256 threads per block (each thread handles 10 elements per membrane), also increases its execution time in a near exponential way, but the performance difference is about two orders of magnitude (100×), and this difference enlarges with the number of membranes (because the resources of the GPU are fully utilized).

Figure 6b shows the behaviour of both simulators executing the benchmark which increases the number of objects per membrane. In this case, the number of membranes is fixed to 1024, which implies to have enough blocks to distribute the work among multiprocessors. Our simulation starts with only few objects per membrane, which implies just few threads per block in the CUDA code. Figure 6b shows that the sequential code initially obtains better performance than the CUDA code until the simulations reach 32 elements per membrane. Less than 32 elements per membrane implies less than 32-threads per blocks in the CUDA code which does not fill a Warp; hence GPU resources are badly used.

The sequential code increases its simulation time along with the number of objects since just one thread has to deal with all the objects in each membrane.

The simulation time remains flat using the CUDA code until reaching a 256-object configuration. The simulation time increases a little bit faster from this configuration onwards because the following configurations have more objects per membranes than threads per block (it uses 256-thread blocks). Therefore, objects in a membrane are equally distributed across all the threads in a block: 512-object per membrane implies two objects per thread; 1024-object per membrane implies 4 objects per thread, and so on. Otherwise, it implies to have an overloaded thread which reduces the performance of our simulator, and leads us to conclude that it is better to have lightweight threads.

Overall, we have obtained an impressive reduction in the simulation time, reaching for our bigger tested configuration (32 768 objects per membrane) an improvement of three orders of magnitude (1000×) in the execution time between the sequential simulator and our CUDA simulator.

CONCLUSIONS AND FUTURE WORK

In this article, we have described the design of a simulator for the class of recognizer P systems with active membranes on the GPU. Our experimental results show that GPUs are good platforms to simulate membrane systems due to the double parallel nature that they present. The first level of parallelism is presented by the objects inside the membranes which fits with the parallelism among threads exposed on GPUs, and the second one is presented between membranes which we represent with the thread blocks on the CUDA programming model.

Using the power and parallelism provided by GPUs to simulate P systems with active membranes is a new concept in the development of applications for membrane computing. We believe that GPU features help researches to accelerate their simulations by using a cheap and scalable parallel architecture.

In forthcoming versions, we are planning to adapt our simulator to simulate specific problems at maximum performance. We are also working to obtain full simulation of P systems with active membranes, removing the limitations mentioned above. Furthermore, we would like to include the possibility to simulate other kinds of P systems in our simulator, such as probabilistic and stochastic P system models, which could be used to computationally model biological systems within the framework of systems biology.

The latest GPUs provide even more massively parallel programming environment, so we will attempt to scale our simulator to obtain better performance and also provide more memory space for our simulations.

Key Points

- P systems are an alternative approach to model biological phenomena in the field of computational systems biology based on the functioning of living cells. However, one needs efficient simulators in order to experimentally validate the models.
- GPUs are being established as a massively parallel processor where programmers can accelerate scientific applications.
- GPUs are good alternative to conventional CPUs to simulate membrane systems due to the double parallel nature that GPUs and P systems present.
- The advent of the accelerators in high performance computing offers fresh avenues for developing new and efficient simulators for P systems and systems biology.

Acknowledgements
The authors would like to thank anonymous reviewers for their detailed and constructive comments and suggestions that have helped to improve this article.

FUNDING

Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, and also by the Spanish MEC and European Commission FEDER under grant CSD2006-00046 to J.M.C., J.M.G., G.D.G.; TIN2006–13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and "Proyecto de Excelencia con Investigador de Reconocida Valía" of the Junta de Andalucía under grant P08-TIC04200 to M.A.M., I.P.-H. and M.J.P.-J.

References

1. Păun G. Computing with membranes. *J Comput Syst Sci* 2000;**61**(1):108–43, and Turku Center for Computer Science-TUCS Report, Nr. 208, 1998.
2. The P systems Webpage: http://ppage.psystems.eu (November 2009, date last accessed).
3. Fontana F, Bianco L, Manca V. P systems and the modeling of biochemical oscillations. *Lect Notes Comput Sci* 2006;**3850**: 199–208.
4. Cheruku S, Pãun A, Romero–Campero FJ, *et al*. Simulating FAS-induced apoptosis by using P systems. *Prog Nat Sci* 2007;**17**(4):424–31.
5. Pérez-Jiménez MJ, Romero-Campero FJ. P systems, a new computational modelling tool for systems biology. Transactions on computational systems biology VI. *Lect Notes Bioinform* 2006;**4220**:176–97.
6. Pérez-Jiménez MJ, Romero-Campero FJ. Modelling gene expression control using P systems: The lac operon, a case study. *Biosystems* 2008;**91**(3):438–57.
7. Krasnogor N, Gheorghe M, Terrazas G, *et al*. An appealling computational mechanism drawn from bacterial quorum sensing. *Bull EATCS* 2005;**85**:135–48.
8. Pérez–Jiménez MJ, Romero-Campero FJ. A model of the Quorum Sensing system in Vibrio fischeri using P systems. *Artificial Life* 2008;**14**(1):95–109.
9. Terrazas G, Krasnogor N, Gheorghe M, *et al*. An environment aware P-System model of quorum sensing. *Lect Notes Comput Sci* 2005;**3526**:473–85.
10. Pescini D, Besozzi D, Mauri G, *et al*. Dynamical probabilistic P systems. *Int J Found Comput Sci* 2006;**17**(1):183–95.
11. Bianco L, Fontana F, Manca V. P Systems with reaction maps. *Int J Foundations Comput Sci* 2006;**17**(1):27–48.
12. Cardona M, Colomer MA, Pérez–Jiménez MJ, *et al*. Modeling ecosystems using P systems: the bearded vulture, a case study. *Lect Notes Comput Sci* 2009;**5391**:137–56.
13. Gutiérrez-Naranjo MA, Pérez-Jiménez MJ, Riscos-Núñez A. Available membrane computing software. In: Ciobanu G, Gh Păun, Pérez-Jiménez MJ (eds). *Applications of Membrane Computing, Natural Computing Series*. Berlin, Heidelberg: Springer-Verlag, 2006;411–436.
14. Ruiz A, Ujaldon M, Andrades JA, Becerra J, Huang K, Pan T, Saltz JH, *et al*. The GPU on biomedical image processing for color and phenotype analysis, BIBE, IEEE 2007;1124–8.
15. Satish N, Harris M, and Garland M. Designing efficient sorting algorithms for manycore GPUs, NVIDIA Corporation No. NVR-2008-001, September 2008.
16. NVIDIA CUDA Programming Guide 2.0 2008: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (November 2009, date last accessed).
17. GPGPU organization. World Wide Web electronic publication: http://www.gpgpu.org (November 2009, date last accessed).
18. NVIDIA CUDA. World Wide Web electronic publication: http://www.nvidia.com/cuda (November 2009, date last accessed).
19. Lindholm E, Nickolls J, Oberman S, *et al*. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 2008;**28**(2):39–55.
20. Owens JD, Houston M, Luebke D, *et al*. Gpu computing. *Proc IEEE* 2008;**96**(5):879–99.
21. García–Quismondo M, Gutiérrez–Escudero R, Martínez–del–Amor MA, *et al*. P-Lingua 2.0: A software framework for cell-like P systems. *Int J Comput Commun Control* 2009; **4**(3):234–43.
22. Nickolls A, Buck I, Garland M, *et al*. Scalable parallel programming with CUDA. *Queue* 2008;**6**(2):40–53.
23. Owens JD, Luebke D, Govindaraju N, *et al*. A survey of general-purpose computation on graphics hardware. *Comput Graph Forum* 2007;**26**(1):80–113.
24. Díaz–Pernil D, Pérez–Hurtado I, Pérez–Jiménez MJ, *et al*. A P-Lingua programming environment for Membrane Computing. *Lect Notes Comput Sci* 2009;**5391**: 187–203.
25. Garland M, Grand SL, Nickolls J, *et al*. Parallel computing experiences with CUDA. *IEEE Micro* 2008; **28**(4):13–27.
26. Păun G. *Membrane Computing, An Introduction*. Berlin: Springer-Verlag, 2002.