Simulating a P system based efficient solution to SAT by using GPUs

José M. Cecilia [b,**], José M. García [b], Ginés D. Guerrero [b], Miguel A. Martínez–del–Amor [a,*], Ignacio Pérez–Hurtado [a,***], Mario J. Pérez–Jiménez [a]

[a] *Research Group on Natural Computing, Dpt. of Computer Science and Artificial Intelligence, University of Seville, Avda. Reina Mercedes s/n, 41012 Sevilla, Spain*
[b] *Grupo de Arquitectura y Computación Paralela, Dpto. Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus de Espinardo, 30100 Murcia, Spain*

A B S T R A C T

P systems are inherently parallel and non-deterministic theoretical computing devices defined inside the field of Membrane Computing. Many P system simulators have been presented in this area, but they are inefficient since they cannot handle the parallelism of these devices. Nowadays, we are witnessing the consolidation of the GPUs as a parallel framework to compute general purpose applications. In this paper, we analyse GPUs as an alternative parallel architecture to improve the performance in the simulation of P systems, and we illustrate it by using the case study of a family of P systems that provides an efficient and uniform solution to the SAT problem. Firstly, we develop a simulator that fully simulates the computation of the P system, demonstrating that GPUs are well suited to simulate them. Then, we adapt this simulator to the GPU architecture idiosyncrasies, improving the performance of the previous simulator.

Keywords:
SAT
P systems
Membrane Computing
GPUs

1. Introduction

Membrane Computing is a computing paradigm inspired on living cells, introduced by Păun [1]. The main idea of this new model of computation is to consider biochemical processes taking place inside living cells from a computational point of view. Devices of this model are called P systems. They have several *syntactic* ingredients: 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*.

Up to now, it has not been possible to have implementations neither *in vivo* nor *in vitro* of P systems. The only way to analyse and execute these devices is through simulators. Therefore, P systems simulators are tools that help the researchers to extract results from a model without the need of having a real implementation.

Since the model of P systems was presented, many simulators and software applications have been produced [2,3]. The majority of these simulators has been developed under sequential architectures using languages such as Java, CLIPS, Prolog or C. However, all of sequential P systems simulators are inefficient on time of execution. They serialise the natural parallelism of P systems, and therefore, the performance is dramatically decreased.

We are witnessing the consolidation of the parallel architectures in the newest generation of processors. The last generation of CMP (Chip MultiProcessor) processors from both Intel and AMD contains up to 8 cores per die. Moreover, these

[*] Corresponding author. Tel.: +34954557947; fax: +34954556251.
[**] Corresponding author.
[***] Corresponding author.
E-mail addresses: mdelamor@us.es (M. Martínez–del–Amor), chema@ditec.um.es (J.M. Cecilia), jmgarcia@ditec.um.es (J.M. García), gines.guerrero@ditec.um.es (G.D. Guerrero), perezh@us.es (I. Pérez–Hurtado), marper@us.es (M.J. Pérez–Jiménez).

processors are still organised in clusters of computers, which are extremely expensive and only available for organisations that have enough resources to buy and maintain them. However, other parallel architectures are being consolidated as an alternative computational model. Among these emergent parallel architectures, the newest version of programmable GPUs provide a compelling alternative to the traditional parallel environments such as cluster of computers, delivering extremely high floating point performance and also a massively parallel framework for scientific applications which fit their architectural idiosyncrasies.

GPUs can support several thousand of concurrent threads providing a massively parallel environment. Current NVIDIA Corporation's GPUs, for example, contain up to 240 scalar processing elements per chip [4], they are programmed using C and CUDA [5,6], and they a have low cost compared with a cluster of computers.

So far, many simulators and software for P systems have been designed for clusters of computers [7], for reconfigurable hardware as FPGA [8,9], and for specific circuits [10]. There is also another attempt to design simulators on GPUs, e.g. [11]. All of these efforts have demonstrated that a parallel architecture is better positioned in performance than traditional CPUs to simulate P systems, due to the inherently parallel nature of them, and specifically GPUs obtain very good preliminary results simulating P systems.

In this paper, we analyse the behaviour of GPUs simulating a P system that belongs to the family of P systems with active membranes and solves the **NP**-complete problem SAT. We analyse the bottlenecks presented in this simulation, and demonstrate that several parameters can be considered when designing P systems based solutions in order to be simulated efficiently on GPUs.

The rest of the paper is structured as follows: In Section 2 a family of P systems with active membranes solving SAT is defined. Section 3 introduces the Compute Unified Device Architecture (CUDA) and some concepts of programming on GPUs are specified. In Section 4 we explain the design aspects of the simulators. Finally, in Section 5 we show some results and several comparisons between the developed simulators. The paper ends with some conclusions and ideas for future work in Section 6.

2. A family of P Systems solving SAT in linear time

Most research in P systems concentrates on the computational power and efficiency of the devices involved. In this context, the inherent parallelism and non-determinism of P systems have been used as tools to solve computationally hard problems in a feasible time. Polynomial time solutions to **NP**-complete problems by means of P systems are achieved by trading time (number of computation steps of the device) for space (number of membranes and objects). This is inspired by the capability of cells to produce an exponential number of new membranes in polynomial time. There are many ways a living cell can produce new membranes: *mitosis* (cell division), *autopoiesis* (membrane creation), *gemmation*, etc. Following these inspirations a number of different models of P systems has arisen, and many of them proved to be computationally complete.

P systems with active membranes is one of the most studied in Membrane Computing [12]. They are formed by a membrane structure, where a label and a polarization (*positive*, *negative* or *neutral*) is associated to each membrane. In this model, every elementary membrane is able to divide itself by replicating its content into a new membrane. In order to solve decision problems (abstract problems that require a *yes* or *non* answer), we consider *recognizer P systems* [13], that is, P systems such that: (a) the working alphabet contains two distinguished elements *yes* and *no*; (b) all computations halt; and (c) if C is a computation of the system, then either object *yes* or object *no* (but not both) must have been sent out of the system (i.e. to the environment), and only at the last step of the computation.

Many examples have been proposed in the framework of P systems with active membranes (with polarizations). In [13], a (uniform) family of recognizer P systems with active membranes solving SAT in linear time is described. The SAT problem is the following: given a Boolean formula in conjunctive normal form (CNF), to determine whether or not there exists a truth assignment to its variables on which it evaluates true.

Let us consider a propositional formula $\varphi = C_1 \wedge \cdots \wedge C_m$ in CNF with $Var(\varphi) = \{x_1, \ldots, x_n\}$, consisting of m *clauses* $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}$, $1 \leqslant i \leqslant m$, where $y_{i,i'} \in \{x_j, \neg x_j : 1 \leqslant j \leqslant n\}$ are the literals of φ. Without loss of generality, we may assume that no clause contains two occurrences of some x_j or two occurrences of some $\neg x_j$ (the formula is not redundant at the level of clauses), or both x_j and $\neg x_j$ (otherwise such a clause is trivially satisfiable, hence can be removed).

We codify φ, which is an instance of SAT with size parameters n and m, by the multiset

$$cod(\varphi) = \bigcup_{i=1}^{m} \{x_{i,j} : x_j \in C_i\} \cup \{\bar{x}_{i,j} : \neg x_j \in C_i\}$$

and we represent by $s(\varphi) = \frac{(n+m)\cdot(n+m+1)}{2} + n$ (denoted by $\langle n, m\rangle$) the length of the formula φ (in a reasonable encoding scheme). The instance φ will be processed by the P system with active membranes $\Pi(s(\varphi))$ with input $cod(\varphi)$.

For each $m, n \in \mathbf{N}$ we consider the P system with active membranes of degree 2: $\Pi(\langle m, n\rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, R, 2)$ defined as follows:

- The input alphabet is $\Sigma = \{x_{i,j}, \bar{x}_{i,j} : 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n\}$.
- The working alphabet is $\Gamma = \Sigma \cup \{c_k : 1 \leqslant k \leqslant m+2\} \cup \{d_k : 1 \leqslant k \leqslant 3n+2m+3\} \cup \{r_{i,k} : 0 \leqslant i \leqslant m, 1 \leqslant k \leqslant 2n\} \cup \{e, t\} \cup \{Yes, No\}$. The set of labels is $\{1, 2\}$.

- The initial membrane structure is $\mu = [\,[\,]_2\,]_1$.
- The initial multisets associated with the membranes are $\mathcal{M}_1 = \emptyset$ and $\mathcal{M}_2 = \{d_1\}$.
- The input membrane is the membrane labeled by 2.
- The set of rules, R, consists of:

(a) $\{[d_k]_2^0 \to [d_k]_2^+[d_k]_2^- : 1 \leqslant k \leqslant n\}$.
(b) $\{[x_{i,1} \to r_{i,1}]_2^+, [\,\overline{x}_{i,1} \to r_{i,1}]_2^- : 1 \leqslant i \leqslant m\}$.
$\{[x_{i,1} \to \lambda]_2^-, [\overline{x}_{i,1} \to \lambda]_2^+ : 1 \leqslant i \leqslant m\}$.
(c) $\{[x_{i,j} \to x_{i,j-1}]_2^+, [x_{i,j} \to x_{i,j-1}]_2^- : 1 \leqslant i \leqslant m,\ 2 \leqslant j \leqslant n\}$.
$\{[\,\overline{x}_{i,j} \to \overline{x}_{i,j-1}]_2^+, [\,\overline{x}_{i,j} \to \overline{x}_{i,j-1}]_2^- : 1 \leqslant i \leqslant m,\ 2 \leqslant j \leqslant n\}$.
(d) $\{[d_k]_2^+ \to [\,]_2^0 d_k,\ [d_k]_2^- \to [\,]_2^0 d_k : 1 \leqslant k \leqslant n\}$.
$\{d_k[\,]_2^0 \to [d_{k+1}]_2^0 : 1 \leqslant k \leqslant n-1\}$.
(e) $\{[r_{i,k} \to r_{i,k+1}]_2^0 : 1 \leqslant i \leqslant m,\ 1 \leqslant k \leqslant 2n-1\}$.
(f) $\{[d_k \to d_{k+1}]_1^0 : n \leqslant k \leqslant 3n-3\}$; $[d_{3n-2} \to d_{3n-1}e]_1^0$.
(g) $e[\,]_2^0 \to [c_1]_2^+$; $[d_{3n-1} \to d_{3n}]_1^0$.
(h) $\{[d_k \to d_{k+1}]_1^0 : 3n \leqslant k \leqslant 3n+2m+2\}$.
(i) $[r_{1,2n}]_2^+ \to [\,]_2^- r_{1,2n}$.
(j) $\{[r_{i,2n} \to r_{i-1,2n}]_2^- : 1 \leqslant i \leqslant m\}$.
(k) $r_{1,2n}[\,]_2^- \to [r_{0,2n}]_2^+$.
(l) $\{[c_k \to c_{k+1}]_2^- : 1 \leqslant k \leqslant m\}$.
(m) $[c_{m+1}]_2^+ \to [\,]_2^+ c_{m+1}$.
(n) $[c_{m+1} \to c_{m+2}t]_1^0$.
(o) $[t\,]_1^0 \to [\,]_1^+ t$.
(p) $[c_{m+2}]_1^+ \to [\,]_1^- Yes$.
(q) $[d_{3n+2m+3}]_1^0 \to [\,]_1^+ No$.

The execution of the P system $\Pi(s(\varphi))$ with input $cod(\varphi)$ can be structured in four consecutive stages:

- In *generation stage*, all possible truth assignments to the variables are generated by using division rules, and they are encoded in the internal membranes. Simultaneously, in such membranes, clauses being true by the encoded truth assignment are checked. Only rules from (a) to (e) are executed, and the whole stage takes $3n-1$ computation steps.
- *Synchronization stage* has the goal of unifying the second subindexes of the objects $r_{i,k}$, to make them equal to $2n$. Rules from (e) to (g) are executed, and the stage needs $2n$ steps to be completed.
- *Check-out stage* has the goal to determine how many (and which) clauses are *true* in every internal membrane (that is, by the assignment *represented* by it). This stage spends 2 steps per clause (a total number of $2m$ steps in the stage), and rules from (h) to (l) are executed.
- *Output stage* searches for internal membranes encoding a solution (that is, containing the object c_{m+1}), what is calculated in each one by the previous stage. If such membranes are found, the object *Yes* is sent out to the environment. Otherwise, the object *No* is sent. Only 4 steps are needed by this stage, and rules from (m) to (q) are executed.

3. Compute Unified Device Architecture (CUDA) parallel programming model

The CUDA programming model developed by NVIDIA allows the programmers write scalable parallel programs for GPUs using a straightforward extension of the C language. Moreover, CUDA is designed for writing highly scalable parallel code that can run across tens of thousands of concurrent threads and hundreds of processor cores. This is basically true because current GPUs can physically contain up to 240 processor cores and 30,720 thread contexts. Therefore, the CUDA programming model is oriented to develop parallel programs that transparently and efficiently scale across different levels of parallelism that GPUs naturally present.

A CUDA program is divided into two main parts (see Fig. 1): The `host part` and the `device part`. The former is the part of the CUDA program which runs on the CPU, being executed by one or more sequential CPU threads. The latter is executed on the GPU and it is called *kernel* in the CUDA programming model. This part can be composed by one or more kernels that are suitable for execution on the GPU. A kernel executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads in two ways showing the two levels of parallelism inside the kernel (see Fig. 1).

Firstly, the programmer declares the number of threads that compose the thread block. The threads of the same block can communicate to each other through fast on-chip memory and they are allowed to synchronize with each other via barriers. Moreover, the programmer declares the number of thread blocks that forms the Grid of blocks that executes a kernel. Threads from different blocks only can share data through a slow off-chip global memory which is visible for every thread in the grid, and the only way to synchronize all the blocks is ending the kernel.

Finally, the CUDA programming model requires that all thread blocks in the same kernel have to be independent, which means that the final result should be the same, independent of the order of the blocks. More details of the CUDA programming model can be found in [5,6].

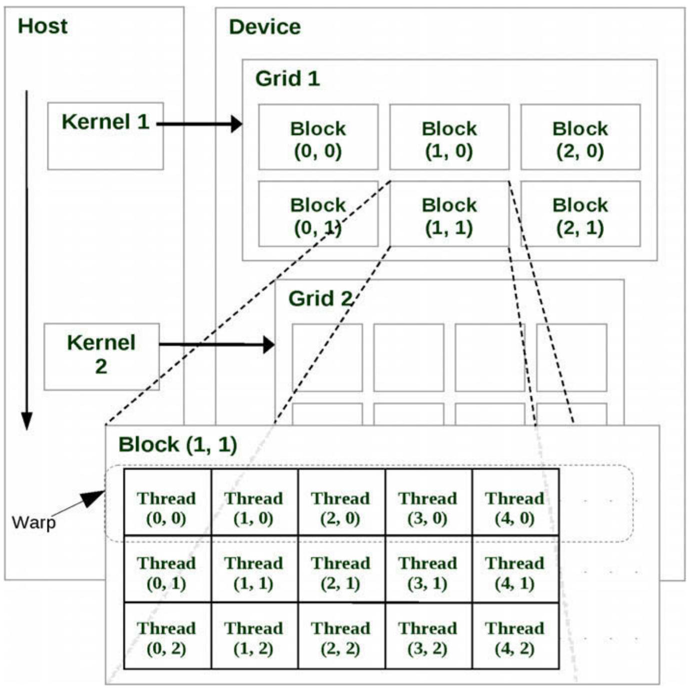

Fig. 1. The CUDA programming model.

Fig. 2 shows the skeleton of a parallel programming with CUDA with a simple example that adds two vectors in parallel. Given vectors **Ad** and **Bd** containing *dimgrid***dimblock* integer numbers, it performs $Ad \leftarrow Ad + Bd$. It contains a very simple implementation of a vector addition in parallel showing both host and device sides of CUDA programming model. The GPU memory needed by the kernel is allocated in the host program. Once the GPU memory is allocated, the host program moves the data from the main memory (CPU memory or host memory) to the GPU memory (global or device memory). The CPU program also launches the kernel and defines the number of threads and blocks that performs the computation on the GPU. This is done with an extension function call syntax AddVectors `<<<dimGrid, dimBlock>>>(...)` used to launch the kernel **AddVectors()** in parallel across *dimGrid* blocks of *dimBlock* threads each. Finally, the host program retrieves the results from the GPU memory to the host memory and frees the memory used in the GPU.

The parallel kernel assigns one element to one thread, performing the addition of each element in parallel. The `__global__` key word indicates that the function is a kernel and, therefore, it will be executed on the GPU. Inside the kernel, first of all, each thread determines which element it should process depending of three main parameters: its thread block integer index (**blockIdx.x**), index within its block (**threadIdx.x**), and the total number of threads per block (**blockDim.x**).

The kernel showed in Fig. 2 is a common parallelization pattern, where a serial loop with independent iterations can be executed in parallel across many threads [14]. The CUDA programmer writes a scalar program (kernel) which specifies the behaviour of a single thread. Each thread computes an iteration of the serial loop identifying the data to compute through indexes previously explained.

The GPU is well suited for certain kind of computations. Among these computations, the data parallelism programs are the best positioned to be accelerated on these architectures. The AddVectors kernel is a simple example where parallel work is decomposed to match result data elements. However, other kind of computations can be accelerated on the GPU. While different thread blocks or different threads of a kernel can potentially execute entirely different code, such task parallelism does not generally scale as well as data parallelism. Moreover, data-parallel kernels typically expose substantially more fine-grained parallelism than task-parallel kernels and, therefore, generally can take best advantage of the GPU architecture [14].

4. Design of the simulator

In this section we briefly describe the simulator for the family of recognizer P systems described in Section 2. Firstly, we explain the previous work that we have done to prepare the development of the parallel simulator on the GPU. Then, we introduce the simulator design that fully simulates the P system computation to solve the SAT problem. Finally, we describe the adapted simulator to the GPU in order to accelerate the simulation.

Host side (CPU)

```
void main()
{
  int size = N * sizeof (float);
  float *A, *B, *Ad, *Bd;
  int dimGrid=numBlocks;
  int dimBlock=numThreads;

  //Allocate memory on the CPU
  A=(float *)malloc(size);
  B=(float *)malloc(size);
  //Initialize CPU memory
  initialize (A,B)
  //Allocate memory on the GPU
  cudaMalloc((void **)&Ad,size);
  cudaMalloc((void **)&Ad,size);
  //Copy data from CPU to GPU
  cudaMemcpy(Ad, A, size ,
       cudaMemcpyHostToDevice);
  cudaMemcpy(Bd, B, size ,
       cudaMemcpyHostToDevice);

  //Launch parallel AddVectors kernel
  //Using dimGrid Blocks of
  //dimBlock thread each
  AddVectors<<<dimGrid,dimBlock>>>(Ad,Bd);

  //Copy data back from GPU to CPU
  cudaMemcpy(A,Ad,size ,
       cudaMemcpyDeviceToHost);

  //Free Device memory
  cudaFree (Ad);
  cudaFree (Bd);
  //Free Host memory
  free (A);
  free (B);
}
```

Device side (GPU)

```
__global__ void AddVectors(float *Ad,float
      *Bd)
{

  uint threadId = blockIdx.x * blockDim.x
       + threadIdx.x;

  Ad[threadId]= Ad[threadId]+Bd[threadId];

}
```

Fig. 2. Parallel programming with CUDA. Code to compute $a \to a + b$.

4.1. Design of the baseline simulator: sequential simulator

As previously mentioned in Section 3, the CUDA programming model is based on C/C++ language. Therefore, the first recommended step when developing applications in CUDA is to start from a baseline algorithm written in C++, where some parts can be susceptible to be parallelized on the GPU. The sequential simulator design is based on the four main stages of the P system execution, as it is depicted in Section 2: Generation, Synchronization, Check-out, and Output. All of these stages are sequentially executed in this simulator, reproducing the behaviour of the P system.

Firstly, the Generation stage is executed, generating 2^n membranes by dividing each one in n steps, where n is the number of variables that composes the CNF formula of the SAT problem. After that, the simulator executes the Synchronization stage which evolves the objects following the rules previously explained. The Check-out stage determines the membranes that codify a solution (where all the clauses are true) of the SAT problem, and finally the Output stage sends out the answer to the environment.

The input of this simulator is a DIMACS CNF file, where the CNF formula is encoded. The output is read in the environment of the P system, where the result is stored. It is important to remark that the semantics of the P system is reproduced by the simulation algorithm, so the present simulator is specific for this solution.

4.2. Design of GPU simulator using CUDA: parallel simulator

The objective of this parallel simulator is to fully simulate the behaviour of the P system computation, doing this in a parallel way whenever is possible. To do that, we use the baseline design based on the four main stages of the P system computation. The first three stages are developed as CUDA kernels in this simulator, and the last one (Output stage) is developed on the CPU.

Similarly to the design presented in [11], this simulator assigns a thread block to each membrane as showed in Fig. 3. In this way, the parallelism among membranes in the P system is simulated. Moreover, each thread is assigned to each object of the input multiset, which is a literal of the CNF formula of the input SAT problem (with the exception of object d_1). This mapping is common to all the defined kernels.

Algorithm 1 shows the pseudocode for this version of the simulator. The Generation stage is simulated by using three kernels which computes the rules previously explained in Section 2. This is an iterative process of n steps where the kernels are

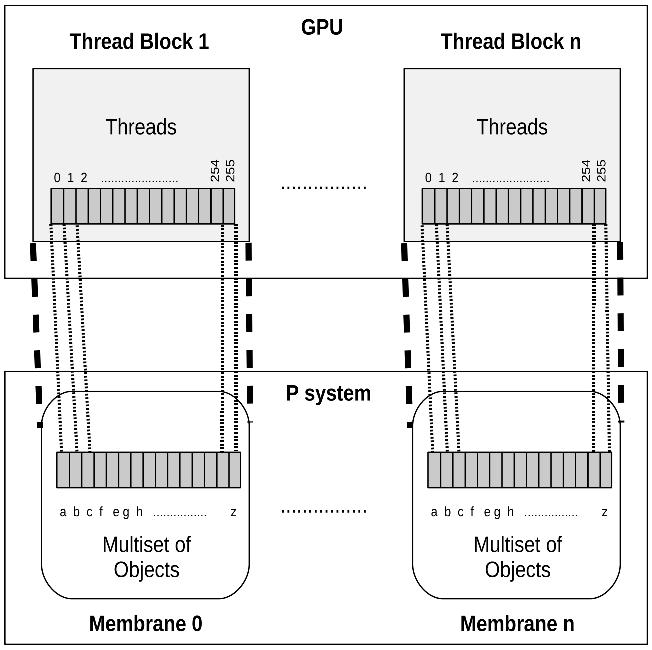

Fig. 3. General design of the parallel simulator.

Algorithm 1. Parallel simulator. Fully reproducing the SAT P system

```
Require: $numMembranes \geqslant 0$
  {Start Generation stage}
  repeat
    $Division\_kernel <<< Blocks, Threads >>> (numMembranes)$
    $numMembranes \Leftarrow numMembranes \times 2$
    $AdjustBlocks(Blocks, numMembranes)$
    $Send\_out\_kernel <<< Blocks, Threads >>> (numMembranes)$
    $Send\_in\_kernel <<< Blocks, Threads >>> (numMembranes)$
    $d \Leftarrow d + 1$
  until $d < numVariables$
  {Starts Synchronization and Check Out stage}
  $Syn\_Check\_kernel <<< Blocks, Threads >>> (numMembranes)$
  {Starts Output stage on CPU}
  $Output(numMembranes)$
```

called n times. Each time, the simulator adjusts the number of thread blocks before calling the kernel, since new membranes are created.

When the exponential workspace is created, the Synchronization and Check-out stages are executed following the rules showed in Section 2. Both stages are performed in the same kernel, and so, in parallel to each membrane. Global synchronization is not needed because there is no communication among the internal membranes at this stage. Finally, the Output stage is developed on the CPU, checking the conditions and launching the result of the computation.

4.3. Adapting the simulator to the GPU architecture: hybrid simulator

Although the parallel simulator fully reproduces the P system computation showed in [13], perhaps another P system design can obtain better performance whenever it is simulated by GPUs. In this sense, the hybrid simulator[1] uses some heuristics along the simulation to adapt the P system computation to the GPU architecture idiosyncrasy.

GPUs are basically a graphics accelerator which are designed to mainly accelerate graphics applications. Graphics applications presents huge data parallelism, that is, developing the same computation over different set of data. Then, the

[1] It is a *hybrid simulator* because it does not perform exactly the same computational steps as the theoretic P system.

Algorithm 2. Hybrid simulator. Adapting the P system computation

```
Require: $numMembranes \geqslant 0$
  {Starts Generation stage}
  repeat
    $Generation\_kernel <<< Blocks, Threads >>> (numMembranes)$
    $numMembranes \Leftarrow numMembranes \times 2$
    $AdjustBlocks(Blocks, numMembranes)$
    $d \Leftarrow d + 1$
  until $d < N$
  {Starts Synchronization, Check Out and Output stage}
  $Syn\_Check\_kernel <<< Blocks, Threads >>> (numMembranes)$
```

communication and synchronization requirements among processing elements should be drastically limited in order to enhance performance [4].

The objective of this simulator is to reduce the communication and synchronization overheads presented in the previous simulator by using several heuristics as showed in the pseudocode 2. For instance, some objects that control the timing of the theoretical computation, depicted in Section 2, are replaced by statical variables instead of dynamic variables. Doing this, this simulator can join CUDA kernels, and therefore, reduce the synchronization overhead produced by launching kernels onto the GPU, since only one kernel can be launched at the same time on the GPU.

The Generation stage is basically the same than the parallel simulator, creating the exponential workspace in the system and reproducing the same steps, but now all of them are included in the same kernel, reducing the synchronization overhead.

Furthermore, the kernel that represents the Check-out stage differs substantially, including a fastest way to produce the output. In this case, this kernel presents more data parallelism whenever it checks the clauses. For this purpose, each thread checks whether its corresponding object encode a true clause. If so, a shared variable, one per thread block and clause, is true. At the end of the kernel, if all these variables are set to true, the answer to the CPU is affirmative (a solution has been found). Otherwise, the answer for this thread block is negative, which means that there is no solution in the membrane, and therefore, the solution depends on the rest of membranes. This approach reduces the data movement through the PCI Express bus which is expensive in terms of performance, and it also loads more computational workload onto the GPU.

5. Performance analysis

In this section, we analyse the performance of the three simulators presented above: the sequential simulator developed in C++ (from now, *simulator 1*), the parallel simulator on CUDA (*simulator 2*) and the hybrid simulator on CUDA (*simulator 3*). The GPU used for the experiments is a NVIDIA GPU Tesla C1060 which has 240 execution cores and 4 GB of device memory,

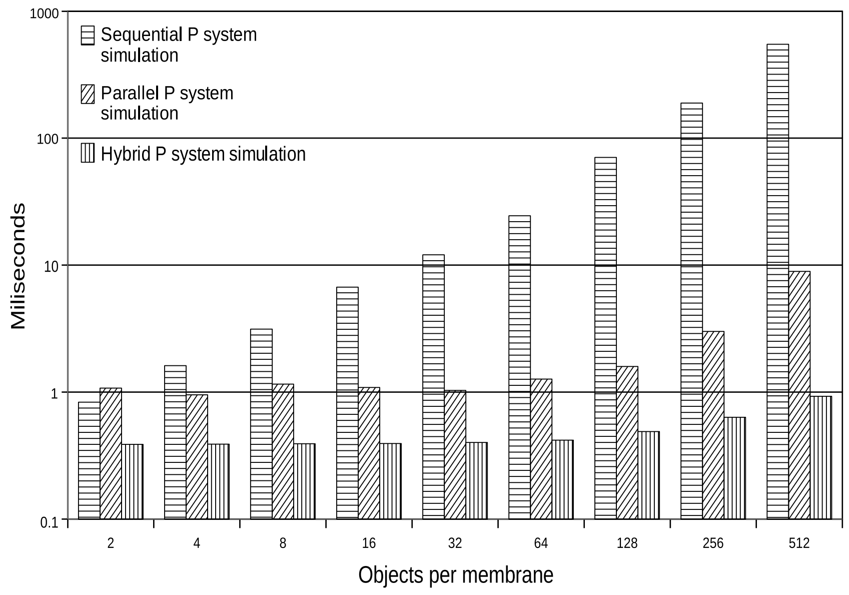

Fig. 4. Simulation performance for sequential, parallel and hybrid simulator: test 1 (2048 membranes).

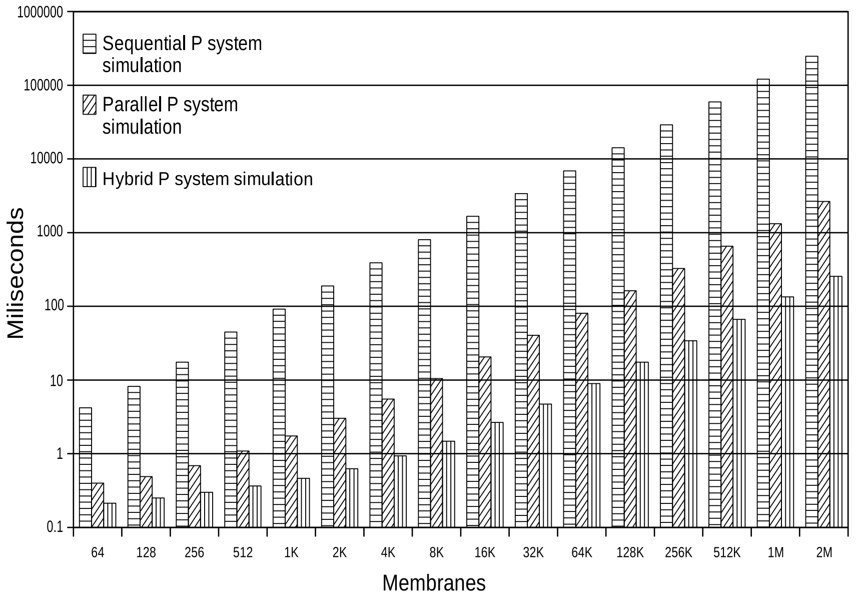

Fig. 5. Simulation performance for sequential, parallel and hybrid simulator: test 2 (256 Objects/Membrane).

plugged in a computer server with a Intel Core2 Quad CPU and 8 GB of RAM, and using a 32-bit Ubuntu Server as Operating System.

We have developed two benchmarks (called *test 1* and *test 2*, respectively) to analyse the performance behaviour of our simulators in two ways: increasing the number of threads per thread block, and increasing the number of thread blocks per grid. Both benchmarks have been generated by WinSAT program [15]. WinSAT is able to generate random SAT problems in DIMACS CNF format file by configuring several parameters: the number of variables (n), the number of clauses (m) and the number of literals per clause (we fix k for our experiments). As mentioned in Section 4, the number of threads per block is associated to the number objects in the input multiset, which is the same as the number of literals of the CNF formula φ ($m*k$). The number of thread blocks (2^n) is equal to the number of membranes in the system, which depends on the number of variables in the CNF formula (n).

Fig. 4 shows the experimental performance of the simulators (in a log scale) for test 1, The benchmark test 1 increases exponentially the number of literals in the CNF formula (and so, the number of objects in the P system and threads per block in the GPU) until reaching a configuration with 512 literals. It also has fixed the number of thread blocks (and membranes) to 2048 ($n = 11$). When the number of threads per block is low, the performance of GPU codes is not substantial compared with the sequential code. That is, the data parallelism is low, and we cannot take advantage of the resources available on the GPU. However, as long as the number of threads per block increases, the data parallelism of the application also increases, and therefore, the performance of our GPU codes improves notably compared to the sequential code, obtaining up to 63x of speed-up between simulators 1 and 2. Furthermore, the simulator 3 accelerates the simulation on the GPU, being this up to 9.63 times faster than simulator 2. Hence, the hybrid simulator is better adapted to the GPU architecture than the parallel simulator of the P system, because it presents more data parallelism in its computation as it is described in Section 4.

Fig. 5 shows the experimental performance of the simulators (in a log scale) for test 2. The benchmark test 2 increases the number of variables in the CNF formula (and so, the number of membranes in the P system and the number of blocks in the GPU in an exponential manner) until reaching a configuration with 2^{11} membranes. The number of simulated membranes is constrained by the available memory of the system. The number of literals in the formula is fixed to 256, which means 256 threads per block.

The behaviour of the GPU simulators, as shown in Fig. 5, is similar in both. This is because the execution time in the GPU codes increases exponentially depending on the number of blocks running at the same time. Once all the GPU resources have been fully occupied, the execution time increases linearly with the number of blocks. In this case, we report up to 94x of speed-up between simulators 1 and 2. However, Fig. 5 shows the speed-up becomes a constant number of 10x when the number of membranes is greater than 128k. This is the number of blocks that fills the pipeline of the GPU in this case, having the hybrid simulator better overall performance than the parallel one.

6. Conclusions and future work

In this paper, we have designed and analysed three simulators for a family of P systems with active membranes that provides an efficient and uniform solution to the SAT problem [13]. The first simulator (simulator 1) performs sequentially on the CPU the computation of this P system. The other two simulators (simulators 2 and 3) have been developed on the GPU using CUDA. Simulator 2 fully simulates the computation of the P system as simulator 1. Simulator 3 adapts the previous simulator to the GPU architecture idiosyncrasies, increasing the data parallelism of the application in some parts, taking advantage of all the parallel resources available on the GPU.

Doing this, we report up to 94x of speed-up between the simulators 1 and 2, and up to 10x between both GPU codes. Therefore, in this work we show two different results. On one hand, we demonstrate that GPUs are well suited to simulate P system due to the highly parallelism that they present in its architecture. Although the GPU is not a cellular machine, its features help the researches to accelerate their simulations. On the other hand, if the P systems based solutions are redesigned to be adapted to the GPU programming model, the performance of the simulations can be also improved.

Nevertheless, the simulation of this kind of P systems that creates an exponential workspace to achieve polynomial time is memory bounded. This bottleneck limits the size of the **NP**-complete problem instances whose solutions can be successfully simulated. Moreover, the simulation of this exponential workspace is performed in exponential time, since no real parallelism like in P systems is available. However, we can reduce this restriction to obtain better simulation times, using the highly parallelism that the GPU provides.

The massively parallel environment provided by the GPUs is good enough for the simulator, however, we need to go beyond. The clusters of GPUs provides a higher massively parallel environment, so we will attempt to scale to those systems to obtain better performance in our simulated codes. Moreover, the newest generation of GPUs such as FERMI from NVIDIA provides improved GPU architectures to develop general purpose applications and also more memory resources.

Furthermore, it would be interesting to avoid the brute force algorithms in the P system designs, and start to develop heuristics in the design of membrane solutions (for instance, avoiding membrane division as much as possible).

Acknowledgements

The first three authors acknowledge the support of the project from the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under Grant 00001/CS/2007, and also by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04. The last three authors acknowledge the support of the project TIN2009-13192 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds, and the support of the “Proyecto de Excelencia con Investigador de Reconocida Valía” of the Junta de Andalucía under Grant P08-TIC04200.

References

[1] G. Păun, Computing with membranes, Journal of Computer and System Sciences, 61 (2000) 108–143, Turku Center for CS–TUCS Report No 208 (1998).
[2] M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, Available Membrane Computing software, in: Applications of Membrane Computing, 2006, pp. 411–436.
[3] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, An overview of P-lingua 2.0, Membrane Computing: 10th International Workshop, WMC 2009, Lecture Notes in Computer Science 5957 (2010) 264–288.
[4] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.
[5] NVIDIA, NVIDIA CUDA Programming Guide 2.0, 2008.
[6] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008) 40–53.
[7] G. Ciobanu, G. Wenyuan, P systems running on a cluster of computers, Membrane Computing: 4th International Workshop, WMC 2003, Lecture Notes in Computer Science 2933 (2004) 289–328.
[8] V. Nguyen, D. Kearney, G. Gioiosa, An algorithm for non-deterministic object distribution in P systems and its implementation in hardware, Membrane Computing: 9th International Workshop, WMC 2008, Lecture Notes in Computer Science 5391 (2009) 325–354.
[9] V. Nguyen, D. Kearney, G. Gioiosa, A region-oriented hardware implementation for Membrane Computing applications, Membrane Computing: 10th International Workshop, WMC 2009, Lecture Notes in Computer Science 5957 (2010) 385–409.
[10] L. Fernandez, V.J. Martinez, F. Arroyo, L.F. Mingo, A hardware circuit for selecting active rules in transition P systems, SYNASC ’05: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE Computer Society,Washington, DC, USA, 2005, pp. 415–418.
[11] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulation of P systems with active membranes on CUDA, Briefings in Bioinformatics, Parallel and ubiquitous methods and tools in Systems Biology (online version). doi:10.1093/bib/bbp064
[12] G. Păun, Membrane Computing. An Introduction, Springer-Verlag, Berlin, Germany, 2002.
[13] M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini, Complexity classes in models of cellular computing with membranes, Natural Computing: an International Journal 2 (3) (2003) 265–285.
[14] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, Micro, IEEE 28 (4) (2008) 13–27.
[15] M. Qasem, WinSAT website (5 2009). URL http://users.ecs.soton.ac.uk/mqq06r/winsat/