The GPU on the simulation of cellular computing models

José M. Cecilia, José M. García, Ginés D. Guerrero,
Miguel A. Martínez-del-Amor, Mario J. Pérez-Jiménez,
Manuel Ujaldón

Abstract *Membrane Computing* is a discipline aiming to abstract formal computing models, called *membrane systems* or *P systems*, from the structure and functioning of the living cells as well as from the cooperation of cells in tissues, organs, and other higher order structures. This framework provides polynomial time solutions to NP-complete problems by trading space for time, and whose efficient simulation poses challenges in three different aspects: an intrinsic massively parallelism of P systems, an exponential computational workspace, and a non-intensive floating point nature. In this paper, we analyze the simulation of a family of recognizer P systems with active membranes that solves the Satisfiability problem in linear time on different instances of Graphics Processing Units (GPUs). For an efficient handling of the exponential workspace created by the P systems computation, we enable different data policies to increase memory bandwidth and exploit data locality through tiling and dynamic queues. Parallelism inherent to the target P system is also managed to demonstrate that GPUs offer a valid alternative for high-performance computing at a considerably lower cost. Furthermore, scalability is demonstrated on the way to the largest problem size we were able to run, and considering the new hardware generation from Nvidia, Fermi, for a total speed-up exceeding four orders of magnitude when running our simulations on the Tesla S2050 server.

Keywords Manycore · GPUs · P systems · SAT problem · High performance computing

J. M. Cecilia · J. M. García · G. D. Guerrero
Computer Engineering and Technology Department,
University of Murcia, 30100 Murcia, Spain
e-mail: chema@ditec.um.es

J. M. García
e-mail: jmgarcia@ditec.um.es

G. D. Guerrero
e-mail: gines.guerrero@ditec.um.es

M. A. Martínez-del-Amor · M. J. Pérez-Jiménez
Computer Science and Artificial Intelligence Department,
University of Seville, 41012 Seville, Spain
e-mail: mdelamor@us.es

M. J. Pérez-Jiménez
e-mail: marper@us.es

M. Ujaldón (✉)
Computer Architecture Department, University of Malaga,
29071 Malaga, Spain
e-mail: ujaldon@uma.es

1 Introduction

Evolutionary computation is a branch of Natural Computing aiming to abstract computing models, called *evolutionary algorithms*, from the processes of (natural) selection and perturbation. Computations start from an initial population of individuals (randomly generated) and proceed according to rules of selection and other operators, such as recombination and mutation. A *fitness function* evaluates each individual measuring its fitness in the environment. Selection acts in the population focusing on high-fitness individuals and eliminating those with worst fitness from a probabilistic viewpoint. Reproduction (recombination and mutation) produces new individuals by random variation of the individuals in the current population. Finally, the survival step decides which individuals survive in the environment. Evolutionary algorithms constitute a wide set of problems trying to replicate the behavior of mother nature through extensive simulations.

These simulations may take a very long time to find the solutions, mainly due to the numerous fitness evaluations and complicated evolutionary operations involved, and the situation worsens on typical large-scale populations. This is where parallel computing naturally emerges to speed-up simulations and provide practical implementations for a feasible search of a single, unified and parametrized solution.

The use of powerful supercomputers has been proposed over the past years to tackle certain instances of this class of methods, among which we may cite ant colony (Stutzle 1998), particle swarm (Mussi and Cagnoni 2009) and even genetic algorithms (Pospichal and Jaros 2009). Following this trend of good alliance between applications and hardware, we contribute with novelties on both sides:

- At hardware level, we propose the use of commodity graphics hardware (GPUs) as a low-cost and emerging parallel architecture to accelerate the simulations.
- At application level, we focus on Membrane Computing, an emergent research area which abstracts computing ideas (like data structures, operations or computing models, among others) from the structure and behavior of single cells, ultimately grouped into complexes of cells.

A wide variety of Membrane Computing models, called *membrane systems* or *P systems*, have been proposed and studied. They are distributed, parallel and non-deterministic computing devices, and some models have been successfully used for designing polynomial time solutions to NP-complete problems by trading space for time. Specifically, these models were inspired by the capability of cells to produce an exponential number of new membranes in linear time, through *mitosis* (membrane division) and/or *autopoiesis* (membrane creation) processes. Major challenges on P systems simulations are (1) a dynamic handling of memory space and (2) an exponential workspace growing as our code increases the number of variables involved to run the simulation.

Currently, we lack of a feasible biological implementation, either in vivo or in vitro, of P systems. The only way to analyze and execute these devices is on silicon-based architectures which are limited by the physical laws. Although some simulators and software applications have been derived (García-Quismondo et al. 2010; Díaz et al. 2009), most of them were developed for sequential architectures using languages such as Java, CLIPS, Prolog or C, where performance is hardly compromised.

The rest of this paper is organized as follows. Section 2 introduces Membrane Computing and describes the behavior of this biologically inspired way of computation, focusing on computational devices called P systems to solve the Satisfiability (SAT) problem. This behavior is later simulated on graphics architectures, which are described in Sect. 4 together with the programming paradigm and the benchmark we create to run experiments. Section 3 describes the parallelism which can be extracted from a P system simulation with active membranes, and once this is learnt, we demonstrate in Sect. 5 how GPUs can accommodate two levels of parallelism in its computational model. Section 6 analyzes different data policies to increase the memory bandwidth, and also to take advantage of the data locality on GPUs by providing a blocking/tiling algorithm and also by managing dynamic queues. We also get a glimpse of the memory limitations to simulate larger datasets and benchmarks by adding more GPUs to the system. Section 7 summarizes our contribution and Sect. 8 provides some directions for future work.

2 Related work

2.1 Our road towards evolutionary algorithms

Our departure point was stencil-based computations (Cecilia et al. 2010c), an active area of research where a number of optimizations have been proposed on two dimensional time evolution problems (Datta et al. 2008; Krishnamoorthy et al. 2010). Two major assumptions meet on stencil computations: (1) constant-sized neighborhood dependencies that remain static throughout the simulation, and (2) availability of data at neighbor locations at each time step.

Neither of these two remain true in a more complex class of evolutionary algorithms like ant colony (Li et al. 2009) or Membrane Computing, where memory access patterns span dynamically the whole computational domain to produce spatial and temporal data dependencies. When those features arise, communication–computation tradeoff strategies get complicated, and more sophisticated approaches are required for an efficient implementation on many-core GPUs, where independent calculations have to predominate in order to exploit fine-grained parallelism.

2.2 Membrane Computing and P systems

Gh. Păun introduced Membrane Computing in 1998 (Păun 2000), and since then, this bio-inspired computing paradigm has attracted research activities within Natural Computing. The model starts with the assumption that processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this model are called P systems, which consist of a cell-like membrane structure, where compartments allocate multisets of objects, that is, sets of objects with multiplicities associated to the elements.

P systems have several syntactic elements (see Fig. 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 (corresponding to chemical substances) and sets of evolution rules (corresponding to reactions) are placed. The region outside the skin membrane is referred to as environment. Every membrane has associated an unchangeable label, and depending on the P system model, it may also contain a charge or polarization that can be modified during the computation. Besides, P systems possess two valuable features: inherent massive parallelism and non-determinism.

A *configuration* of a P system is an instantaneous description of the system which can be described by the structure of membranes at this moment, and the multisets of objects associated with each compartment/region in the system. We evolve from a configuration to a next one by applying the rules of the system to the objects inside the regions in a *maximal parallel* manner, that is, the rules should be used in parallel to the maximum degree attainable (we assign objects to rules, choosing the rules and the objects assigned to each rule in a non-deterministic manner, but in such a way that after the assignation no further rule can be applied to the remaining objects). When defining transitions in a P system, different maximal multisets of rules can be chosen in order to perform a maximally parallel transition step, hence the evolution of the system has branching in a non-deterministic manner.

A *computation* of a P system is a (finite or infinite) sequence of instantaneous transitions between configurations. The computation starts with an initial configuration of the system, where the input data of a given problem is encoded. The transition from a configuration to the next one is performed by applying rules to the objects inside the regions. A computation which reaches a configuration (called *halting configuration*) where no rule is applicable to the existing objects, is a *halting computation*. Only halting computations give a result and the result is encoded by the multiset of objects in a specified *output membrane* or in the environment of the system.

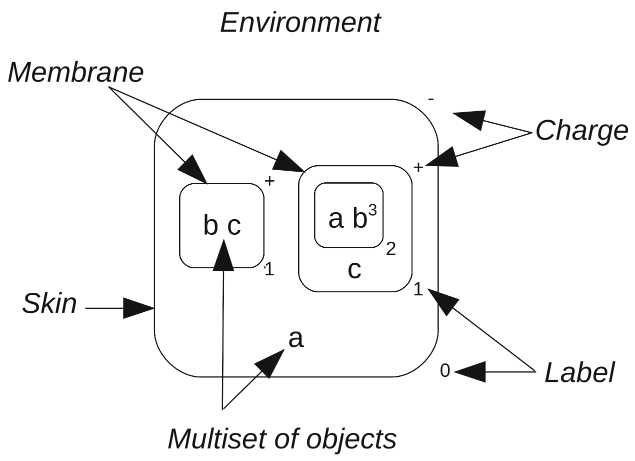


Fig. 1 Structure and syntactic elements of a P system

Note that P systems exhibit two levels of parallelism: one at region level (rules are applied in parallel), and another one at system level (regions evolve concurrently). The objects inside the membranes evolve according to given rules in a synchronous, parallel, and non-deterministic way.

The two-level parallelism and non-determinism can be used to solve NP-complete problems in polynomial time, reducing this from an exponential time, but at the expense of using an exponential workspace of membranes and objects which is created in linear time.

Computational complexity theory usually deals with *decision problems* which require a `yes` or `no` answer. Of course, many abstract problems are not decision problems but, for instance, each optimization problem can be transformed into a roughly equivalent decision problem. There is a natural matching between decision problems and languages, in such a way that *solving* a *decision problem* is equivalent to *recognize* the *language* associated with it. That is the rationale for us to consider a class of P system, called *recognizer*, in order to solve decision problems within the framework of Membrane Computing. This kind of P systems has a working alphabet with two distinguished objects, `yes` and `no`, which halt all computations in the system, and if $\mathcal{C}$ is one of those computations, then either object `yes` or object `no` (but not both) must have been released into the environment and the result encoded in the last step of the computation.

In this computing paradigm, decision problems are solved by using families of recognizer confluent P systems (Pérez-Jiménez et al. 2003) where all possible computations with the same initial configuration must give the same answer. Therefore, this kind of P systems capture the true algorithmic concept in the sense that the result of one computation suffices to determine the answer of the overall system.

Up to date, there have not been in vivo nor in vitro implementations of P systems, and researchers have focused on simulators developed in silico whose initial versions were targeted to sequential platforms (Díaz et al. 2009; García-Quismondo et al. 2010). From this departure point, the main challenge for the simulations of P systems in general is to find the right platform to exploit massively the parallelism inherent to the definition of P systems.

In this respect, several efforts have been done implementing this massively parallelism on parallel architectures. For instance, Alonso et al. (2008) proposed a circuit implementation for the class of transition P systems;

Nguyen et al. (2010) proposed an implementation of transition P systems in FPGAs, providing several levels of parallelism, one at rule level and other at region level, releasing a software framework for Membrane Computing called Reconfig-P, and a generic simulator on GPUs for a family of recognizer P system with active membranes was presented in Cecilia et al. (2010b), showing that the double level of parallelism exposed by GPUs represents a valid alternative to simulate P systems.

2.3 P systems with active membranes

Many different models of P systems have arisen from the inspiration of different behaviors from living cells. In this sense, *P systems with active membranes* are one of the first models introduced by Păun (2002), and they have been proved to be complete from a computational viewpoint, equivalent in this respect to Turing machines.

P systems with active membranes have rules which directly involve the membranes where the objects evolve and at the same time make membranes to progress: *evolution* and *communication* rules are associated with objects and membranes, which can also be *dissolved* or multiplied by *division*. A new feature of these membrane systems is *polarization*, they have one of three possible *electrical charges*: $+$ (positive), $-$ (negative), 0 (neutral). Since the number of membranes can grow exponentially, polynomial time solutions to **NP**-complete problems in Membrane Computing are achieved by trading space (number of membranes and objects) for time (transitions in the computation). This is inspired by biological facts, namely, the capability of cells to produce new membranes via events like *mitosis*.

Now we briefly provide a description of P systems with active membranes (see Păun 2009 for additional information). These systems are of the form $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R)$, where $q \geq 1$ is the initial degree of the system; Γ is the working alphabet of *objects*, H is a finite set of *labels* for membranes; μ is a membrane structure (a rooted tree), consisting of q membranes injectively labeled with elements of H and every membrane has associated electrical charges from the set $\{+,-,0\}$, $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over Γ, describing the *multisets of objects* placed in the q regions of μ; and R is a finite set of *rules*, where each rule adopts one of the following forms:

(a) $[a \rightarrow v]_h^\alpha$ where $h \in H, \alpha \in \{+,-,0\}, a \in \Gamma$ and v is a string over Γ describing a multiset of objects associated with membranes and depending on the label and the charge of the membranes (*evolution rules*).

(b) $a[\,]_h^\alpha \rightarrow [b]_h^\beta$ where $h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in \Gamma$ (*send-in communication rules*). An object is introduced in the membrane, possibly modified, and the initial charge α is changed to β.

(c) $[a]_h^\alpha \rightarrow [\,]_h^\beta b$ where $h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in \Gamma$ (*send-out communication rules*). An object is sent out of the membrane, possibly modified, and the initial charge α is changed to β.

(d) $[a]_h^\alpha \rightarrow b$ where $h \in H, \alpha \in \{+,-,0\}, a, b \in \Gamma$ (*dissolution rules*). A membrane with an specific charge is dissolved in reaction with a (possibly modified) object.

(e) $[a]_h^\alpha \rightarrow [b]_h^\beta [c]_h^\gamma$ where $h \in H, \alpha, \beta, \gamma \in \{+,-,0\}, a, b, c \in \Gamma$ (*division rules*). A membrane is divided into two membranes. The objects inside the membrane are replicated, except for a, that may be modified in each membrane.

Rules are applied according to the following principles:

- All the elements which are not affected by operations to be applied remain unchanged.
- Rules associated with label h are used for all membranes with this label, regardless the membrane is initial or generated by division during the computation.
- Rules from (a) to (e) are used as usual in the framework of Membrane Computing, i.e. maximizing parallelism. At each step, each object in a membrane can only be used by at most one rule (non-deterministically chosen), but any object which can evolve by a rule must do it (given the constraints indicated below).
- Rules (b)–(e) cannot be applied simultaneously in a membrane in a single computational step.
- An object a in a membrane labeled with h and with charge α can trigger a division, yielding two membranes with label h, one of them having charge β and the other one having charge γ. Note that existing contents prior to the division, except for object a, can be the subject of rules in parallel with the division. In this case, we consider that two processes take place in a single step: "first", the contents are affected by the rules applied to them, and, "after that", the results are replicated into the two new membranes.
- If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external one. The skin is never dissolved neither divided.

Finally, we say that $\Pi = (\Sigma, \Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R, \text{in})$ is a *P system with active membranes and input* if it verifies the following conditions:

1. The alphabet Σ, strictly contained in Γ, is the *input alphabet* of Π.
2. $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R)$ is a P system with active membranes.

3. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over $\Gamma \setminus \Sigma$, describing the multisets of objects placed in the q regions of μ.
4. in $\in H$ is the label of the input membrane of Π.

Given a multiset m over Σ, the *initial configuration of* Π *with input* m is the tuple $(\mathcal{M}_1, \ldots, \mathcal{M}_{\text{in}} + m, \ldots, \mathcal{M}_q)$, where $\mathcal{M}_{\text{in}} + m$ means the union of the multisets $\mathcal{M}_{\text{in}}$ and m. Therefore, it is possible to distinguish, in this initial configuration, the objects in membrane *in* that come from the input m of the system, since $\mathcal{M}_{\text{in}}$ represents a multiset over $\Gamma \setminus \Sigma$ and m is a multiset over Σ.

2.4 The SAT problem

Propositional Satisfiability problem (SAT, for short) was the first known **NP**-complete problem, as proven by Cook (1971). Let us recall that a boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation. The SAT problem consists of, given a boolean formula in CNF, determine whether or not it is *satisfiable*, that is, whether there exists an assignment of truth values to its variables on which it evaluates true. This is of paramount importance in many computer science areas, including theory, algorithmic, artificial intelligence, hardware design, electronic design automation, and verification.

The time spent by all known deterministic algorithms solving the SAT problem is exponential depending on the size of the input. With the help of membrane systems, we were able to find the solution at linear time, but trading space for time, that is, by making use of an exponential workspace built in linear time (Pérez-Jiménez et al. 2003). Now we describe the required P-system, which is the focus of our computational simulation.

Let us consider a boolean formula $\varphi = C_1 \wedge \ldots \wedge C_m$ in CNF with $\text{Var}(\varphi) = \{x_1, \ldots, x_n\}$, consisting of m *clauses* $C_i = z_{i,1} \vee \ldots \vee z_{i,k_i}$, $1 \le i \le m$, where $z_{i,i'} \in \{x_j, \neg x_j : 1 \le j \le n\}$ 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 clause level), or both x_j and $\neg x_j$ (otherwise such a clause is trivially satisfiable, hence it can be removed).

We codify φ, which is an instance of SAT with size parameters n and m, by the following multiset (which is actually a set)

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

and the length of the formula φ (in a reasonable encoding scheme) is represented by $s(\varphi) = \frac{(n+m)\cdot(n+m+1)}{2} + n$ (denoted by $\langle n, m \rangle$: it is well known that the application s is a bijection from $\mathbf{N}^2$ onto $\mathbf{N}$). The instance φ will be processed by the P system with active membranes $\Pi(s(\varphi))$ with input cod(φ) (Pérez-Jiménez et al. 2003).

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

- The input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} : 1 \le i \le m, 1 \le j \le n\}$.
- The working alphabet is $\Gamma = \Sigma \cup \{c_k : 1 \le k \le m + 2\} \cup \{d_k : 1 \le k \le 3n + 2m + 3\} \cup \{r_{i,k} : 0 \le i \le m, 1 \le k \le 2n\} \cup \{e, t\} \cup \{\text{Yes}, \text{No}\}$.
- The set of labels is $H = \{1,2\}$.
- The initial membrane structure is $\mu = [\ [\ ]_2\ ]_1$ (a rooted tree where the root is the membrane labeled by 1 which has a son: the membrane labeled by 2).
- The initial multisets associated with the membranes are the following $\mathcal{M}_1 = \emptyset$ and $\mathcal{M}_2 = \{d_1\}$.
- The input membrane is the membrane labeled by 2.
- The set of rules, R, consists of:

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

The symbols c_k and d_k are counters. The presence of a symbol $r_{i,k}$ in a membrane represents that the truth assignment encoded by such membrane makes true the ith clause. The subscript k is used for a synchronization process. Finally, e and t are auxiliary symbols.

The execution of the P system $\Pi(s(\varphi))$ with input $cod(\varphi)$ can be structured in four consecutive stages (see Pérez-Jiménez et al. 2003 for details):

1. *Generation* Membranes are structured within a rooted tree with a single branch. The root node is the *skin membrane*, and the second node is called *internal membrane*. All possible truth assignments to the variables are generated by using division rules, and they are encoded in the internal membranes by executing step by step the set of P system rules. In this way, 2^n internal membranes are created such that each one encodes a truth assignment to the variables of the formula. This stage ends whenever object d_n appears in the skin membrane. Only rules from (a) to (e) are executed at this stage, and the whole stage takes $3n - 1$ transition steps.
2. *Synchronization* The objects encoding a true clause (a partial solution to the CNF formula) are unified in the membrane, making the second subscript of $r_{i,j}$ to be $2n$. Rules from (e) to (g) are executed, and the stage requires $2n$ steps to be completed.
3. *Check out* The goal here is to determine how many (and which) clauses are *true* in every internal membrane (that is, by the assignment that represents). This stage ends when object d_{3n+2m} appears in the skin membrane. It spends $2m$ steps, and rules from (h) to (l) are executed.
4. *Output* Internal membranes encoding a solution send an object to the skin. If the skin has such object from some membrane, the object *Yes* is sent to the environment. Otherwise, the object *No* is sent. Only four steps are needed by this stage, and rules from (m) to (q) are executed.

2.5 P system simulation algorithm

The P system simulation algorithm to solve the SAT problem is based on the P system computation described above. For instance, Algorithm 1 summarizes the sequential code based on previous stages. First, *Generation* and *Synchronization* are the stages creating an exponential workspace of membranes in a synchronous way, and also unifying the objects that codify a partial solution. Both stages are executed in the same function, which is referred to as *Generation* from now on. Note that each membrane runs in parallel at each iteration of *Generation*, but a global synchronization is required by different iterations.

Once the workspace is created, the *Check out* and *Output* stages are performed. First, they determine the clauses being true in every internal membrane, and then they check whether there is a solution for the SAT problem. Hereafter, we combine these two stages into a joint *CheckOut* function.

Algorithm 1 The sequential pseudocode of the P system simulation algorithm for the SAT problem with n variables.

```
Require: $n \geq 0$
  {Start Generation and Synchronization stages}
  repeat
    Generation
  until $n$
  {Start Check out and Output stages}
  CheckOut
```

The specific simulation of the family of P systems that solves SAT for a single GPU is analyzed in Cecilia et al. (2010a), where problems to carry out the theoretical simulation of P systems on GPUs are depicted, and some heuristics to accelerate its computation are provided.

3 The parallel simulator for the P system solving the SAT problem

The family of P systems solving the SAT problem (in short, *SAT P system*) gathers all computational features of the recognizer P systems with active membranes (Păun 2002). Among them, we highlight the theoretical double level of parallelism and non-determinism that makes P systems a computational tool to solve NP-complete problems in polynomial time.

The first level of parallelism for the SAT P system is found among membranes, that is, by executing rules inside each membrane in parallel along the computation (see Fig. 2). The second level of parallelism is found within each membrane (see Fig. 3). That way, the first level is coarse-grained and can be characterized by an inter-task parallelism and exploited by the number of processors available in a parallel system, whereas the second level of parallelism is fine-grained and intra-task to be exploited by the number of cores within each processor, either on multi- or many-core architectures.

Membrane parallelism is shown in Fig. 2 for the execution of the *Generation* function for the SAT P system in a sequential as well as a parallel architecture with four Compute Elements (CE). In a parallel architecture, a set of membranes is initially created by the master process, whose size is equal to the number of CEs available during the execution. Then, a membrane is sent to each CE by the master processor. This step is called *Parallel Preprocessing (PP)*, and it is developed just before the *Generation* starts the computation on each CE. This CE is represented by a processor (die) which can later be eventually decomposed into multi- or many-cores when exploiting intra-task parallelism.

Fig. 2 Sequential and parallel membranes generation on four Compute Elements (CE). The *Parallel Preprocessing (PP)* is required to set up the parallel execution prior to its starting on each *CE*. This CE is represented by a processor (die) which can later be eventually decomposed into multi- or many-cores depending on target architecture

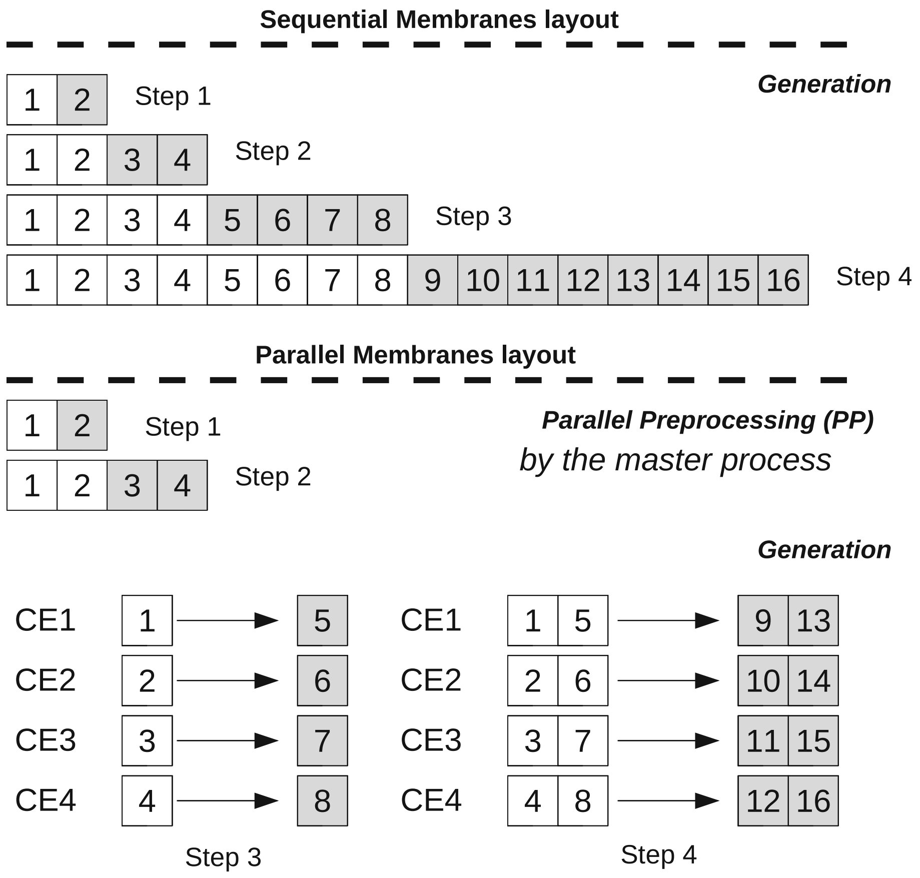
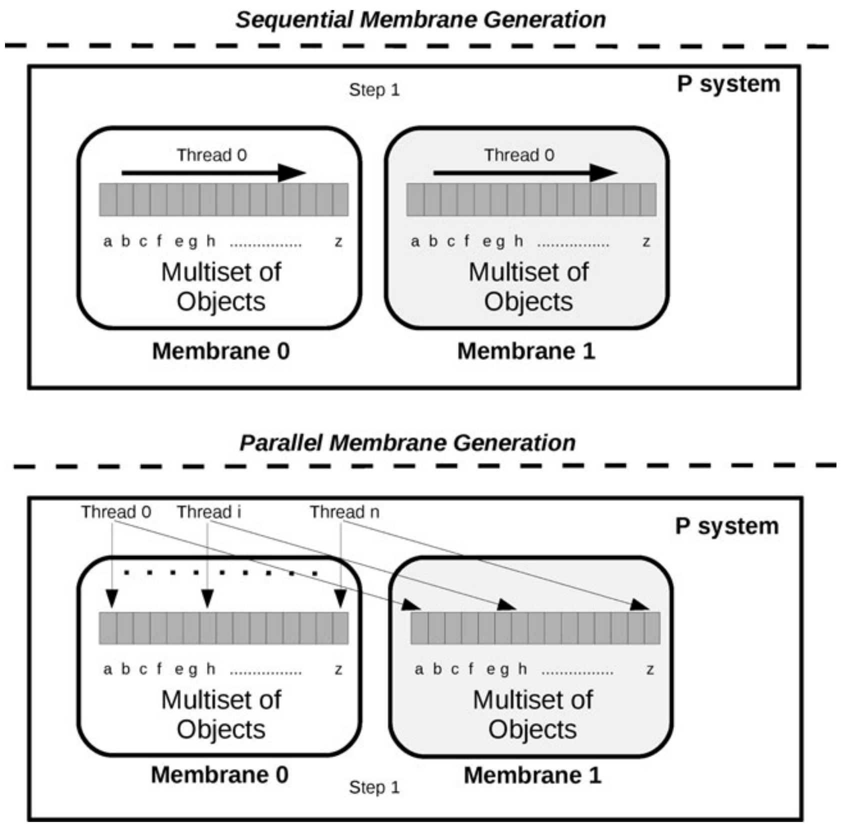


Fig. 3 Sequential and parallel execution when creating the exponential workload shows the second level of parallelism in P systems, that internal to membranes. P system rules are applied for the SAT problem running on hardware cores, and many-core GPUs translates this into massive parallelism using hundreds of cores



Furthermore, Fig. 2 shows that it is known which membrane generates each one and also in which computational step. For instance, membrane two is always generated by membrane one in the first computational step, membrane three is always generated by membrane one in the second step, and so on. Finally, each node sends the partial response back to the master in order to produce the final result of the P system.

Figure 3 shows the second level of parallelism in P systems, that internal to membranes. Once the initial data has arrived to the CE after the *Parallel Preprocessing* step, it starts the computation according to Algorithm 1, and applying the P system rules for the SAT problem depicted in Pérez-Jiménez et al. (2003). Then, resources on each CE can be exploited at its peak to cooperate for speeding up the computation of the *Generation* and *CheckOut* functions. This resources are essentially hardware cores, but fortunately GPUs are many-core which can handle this level of parallelism at large scale using hundreds of streaming processors (see Table 1).

4 Experimental setup

4.1 Hardware features

Major features for our hardware equipment are summarized in Tables 1 and 2. Intel-based host machines provide service to two different GPU systems based on Nvidia Teslas, a C1060 model manufactured in mid 2008 and delivered as a graphics card plugged into a PCI-express 2 socket (see Fig. 4a), and the more recent S2050 released in November 2010 and based on the Fermi architecture (NVIDIA 2010). In addition to a larger number of streaming processors (see Fig. 5), Fermi improves the GPU capabilities with additional features like enhanced single-precision floating-point accuracy and double-precision floating-point performance, new general-purpose L1 and L2 caches, faster context switching, a unified 64-bit virtual address space and a brand new instruction set.

The Tesla S2050 Computing System is mounted on a 1U rack chassis and endowed with four M2050 Fermi GPUs on a 2 $\times$ 2 setup accessible through a couple of PCI-express Gen2 external cables which are plugged into the motherboard of the host computer via a Host Interface Card (HIC).

4.2 CUDA programming model

All GPU platforms previously outlined can be programmed using the Compute Unified Device Architecture (CUDA) programming model which makes the GPU to operate as a highly parallel computing device. Each 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). Different memory spaces are available in each GPU on the system. The global memory (also called device or video memory) is the only space accessible by all multiprocessors. It is the largest and the slowest memory space and it is private to each GPU on the system. Moreover, each multiprocessor has its own private memory space called shared memory. The shared memory is smaller and also lower access latency than global memory (NVIDIA 2008).

Table 1 Hardware features for the Teslas C1060 and M2050 GPUs

GPU element	Feature	Tesla C1060	Tesla M2050
Streaming processors (GPU cores)	Cores per multiprocessor	8	32
	Number of multiprocessors	30	14
	Total number of cores	240	448
	Clock frequency	1,296 MHz	1,147 MHz
Maximum number of threads	Per multiprocessor	1,024	1,536
	Per block	512	1,024
	Per warp	32	32
SRAM memory available per multiprocessor	32-bit registers	16 K	32 K
	Shared memory	16 KB	16 or 48 KB
	L1 cache	No	48 or 16 KB
	Total SRAM (shared + L1)	16 KB	64 KB
Global (video) memory	Size	4 GB	3 GB
	Speed	2 $\times$ 800 MHz	2 $\times$ 1,500 MHz
	Width	512 bits	384 bits
	Bandwidth	102 GB/s	144 GB/s
	Technology	GDDR3 DRAM	GDDR5 DRAM

Table 2 Summary of features for the host architectures used during our experimental survey. The Tesla GPUs are described separately in Table 1

Hardware platform	4 Intel Xeon E5530 CPU (plus 4 Tesla C1060 GPUs)	1 Intel Q9440 CPU (plus 4 Tesla M2050 GPUs)
Cores per CPU	4	4
CPU cores and speed	16 @ 2.4 GHz	4 @ 2.66 GHz
Main memory (DRAM)	16 GB (+16 GB video)	8 GB (+12 GB video)
CUDA compiler	nvcc Nvidia 2.3	nvcc Nvidia 3.1

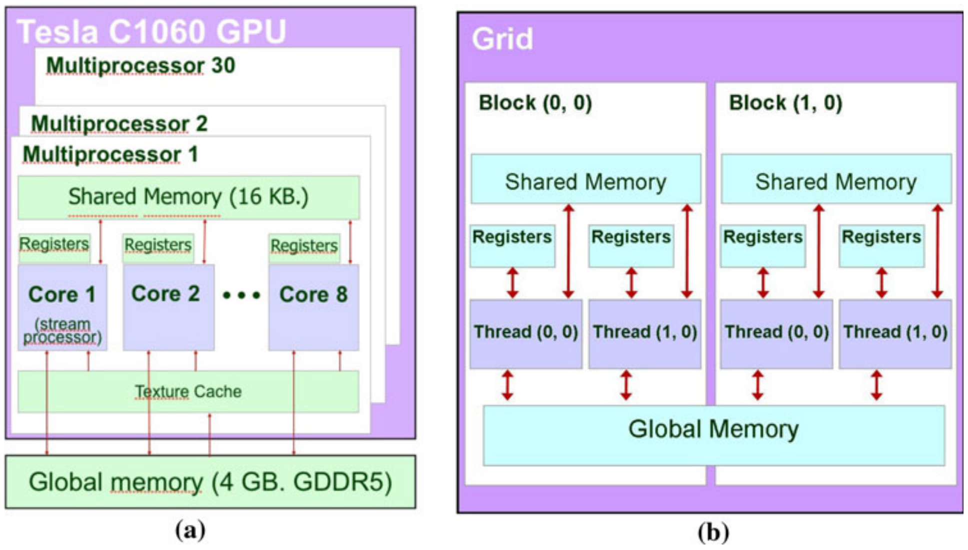


Fig. 4 **a** The CUDA hardware interface for the Tesla C1060 GPU. We can see how 240 stream processors are arranged into 30 multiprocessors (z dimension), each composed of 8 cores (x dimension). The memory hierarchy extends on dimension y to accommodate registers, share memory, texture cache and, finally, global memory. **b** The CUDA programming model. Threads are mapped into cores using a grid composed of blocks which can only communicate through global memory

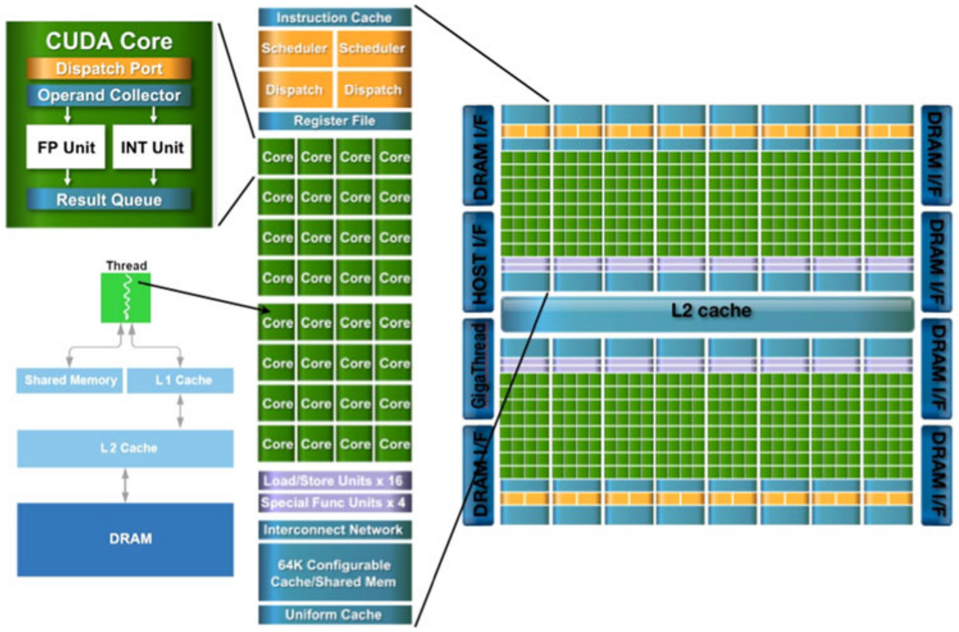


Fig. 5 An outline of the Fermi architecture used within the Tesla M2050 GPU. We have 16 multiprocessors, each composed of 32 cores. An L1 cache (up to 48 KB) is introduced in parallel to the shared memory, and even an L2 cache emerges with 768 KB. More resources are devoted to floating-point arithmetic too

The CUDA programming model is based on a hierarchy of abstraction layers (see Fig. 4b): 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. 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 to arrange parallelism given the program constraints (i.e., data and control dependencies).

4.3 Our benchmark

Data policies and simulation performance are evaluated on GPUs under a set of benchmarks generated by the WinSAT program (Qasem 2009), which basically builds data structures for the simulator to run under a pre-established set up. WinSAT can generate random SAT problems in DIMACS CNF format file by configuring three parameters: the number of variables (n), the number of clauses (m and the number of literals per clause (k).

The number of membranes in our P system depends on the number of CNF variables, n (membranes $= 2^n$). We vary this parameter from $n = 13$ variables (2^{13} membranes) to $n = 21$ variables (2^{21} membranes), whereas the number of literals ($l = m \times k$) is kept constant in $l = 256$. Memory requirements for each benchmark can be calculated according to Eq. 1. For example, the largest case corresponds to $n = 21$, which requires 2 GB (note that $n = 22$ would require 4 GB to run, and the Tesla M2050 is endowed with 3 GB of video memory on each of its four GPUs).

$$\text{Size} = 2^n(\text{membranes}) \times l(\text{objects}) \times 4(\text{unit})\,\text{bytes}. \qquad (1)$$

5 The GPU implementation for the simulator

Our P system simulator for the SAT problem organizes data depending on the features of the GPU architecture. In short, a CUDA thread block is set for each membrane and a CUDA thread per object (or set of objects) in the initial multiset.

The first attempt for the SAT P system simulation on GPUs, the *Generation* stage, is encoded as a CUDA kernel, and it starts right after the *Parallel Preprocessing* step. Once membranes have been generated, the *CheckOut* stage starts its execution in a different kernel. Each thread block loads a membrane from global memory, and then each thread checks the rules associated with this stage. Finally, each block returns whether its associated membrane makes true the CNF formula or not. For these stages, all threads within a CUDA thread block cooperate with coalesced access to device memory (threads of the same warp access the same memory segment either for reading or writing—for additional explanations on coalescing and warps, we refer the reader to NVIDIA 2008).

Blocking can be exploited on GPUs, taking advantage of the on-chip shared memory by using tiles and dynamic queues with the aim of increasing the bandwidth to device memory (see Fig. 6). Tiles decompose the computational domain into a number of independent chunks whose size fits within the shared memory, and they are implemented using the concept of CUDA blocks (see Fig. 4b). This way, the whole data structure can benefit from this high-speed and low-latency memory even though it represents just a tiny fraction of the algorithm requirements. On the other hard, dynamic queues allow to establish the number of queued elements at real-time, thus increasing or decreasing the number of queues handled by each block. Our simulation requires to allocate memory dynamically as the data set grows exponentially on every new iteration, and once this memory is generated, we have to check for this overwhelming amount of memory not to exceed our hardware limits. Here, dynamic queues are used for the generation phase and tiling is used during the checking phase.

The simulation has to perform a *Block Preprocessing* (BP) step before starting the *Generation* stage itself, which is implemented through a CUDA kernel where a set of membranes are partially created, placing them apart from each other at a block size distance. An additional kernel is created at the end of the simulation to perform the *Generation* locally to each block, followed by the *CheckOut* stage. Each thread on a thread block cooperates for an efficient load from global memory to shared memory of the initial membrane generated by the *Block Preprocessing* step (represented by black squares in Fig. 6). Then, the *Generation* stage interacts with shared memory, saving expensive loads/writes from/to global memory which are around 400 times slower. Finally, the *CheckOut* stage is performed over the data stored in shared memory after a block-level synchronization. This checks whether a clause makes true the CNF formula, and writes its result into device memory.

Figure 7 shows the data policy used by the simulation of the P system for the SAT problem on a GPU-based platform. This simulator arranges data according to the "best practices" existing at this moment for CUDA enabled devices with CUDA Compute Capabilities (CCC) 1.3 (NVIDIA 2008). Nevertheless, those guidelines are mainly focused on arithmetic intensive applications on a single GPU. It remains to be seen whether they are valid on architectures like GPU-based clusters with a much higher degree of parallelism.

Within a GPU-based cluster, GPUs cannot interact with each other, and a CPU process has to be created to monitor each GPU independently. Note that the C1060 does not

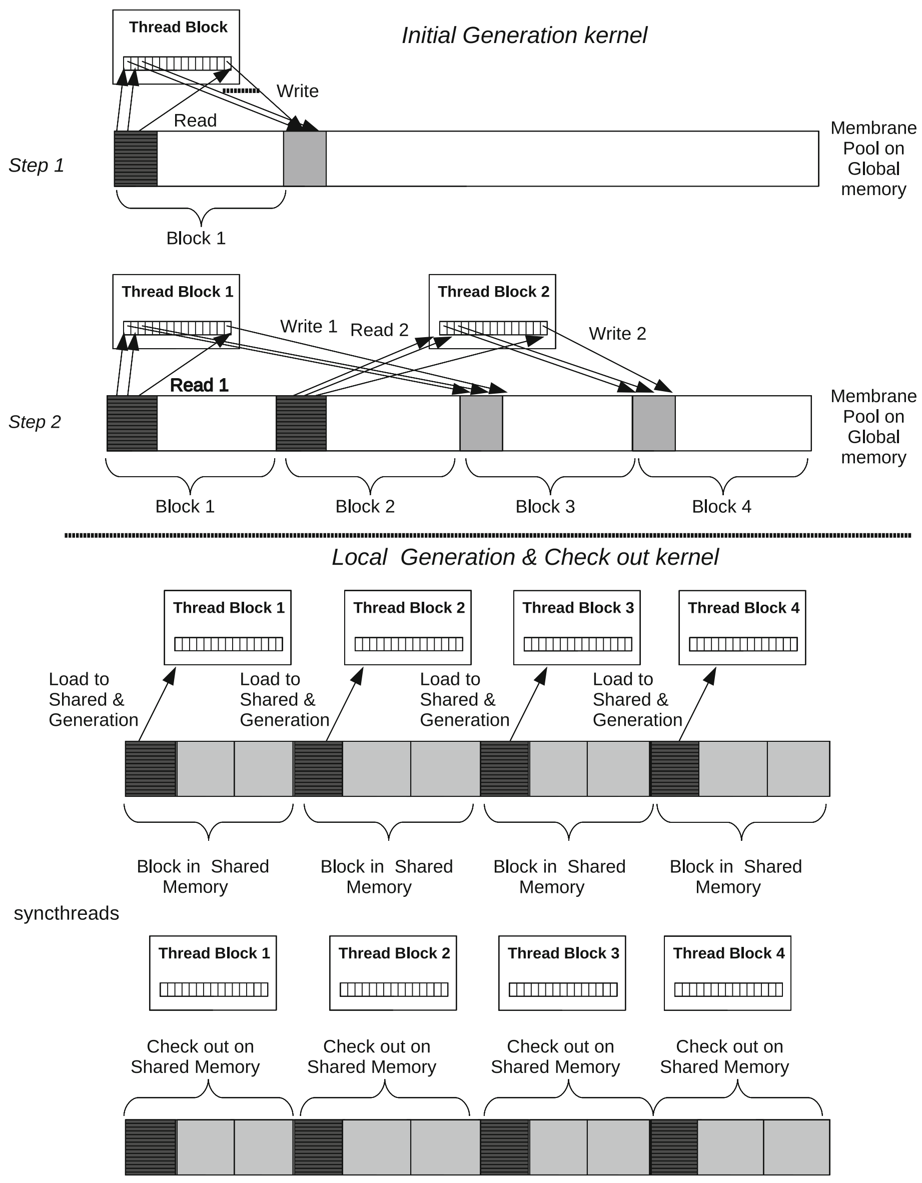


Fig. 6 P system simulation on a single GPU using dynamic queues and tiling. On the *upper side*, dynamic queues and tile sizes are established (Block$_i$ bytes) by placing the initial data into separate global memory spaces. On the *lower side*, the local generation of the dynamic queues is performed in Shared Memory in order to proceed with the Check Out using tiling

force us to use parallelism at CPU core level, as we have exactly four CPUs, which can individually host each of the required processes. This way, we do not need to use multithread capabilities on CPU cores to handle multiple GPUs. Within our S2050 system, however, *pthreads* have to be used to enable the four GPUs available. Note that *pthreads* may also be useful to exploit multi-core parallelism when the simulation runs entirely on a single CPU, but the CPU times that we have measured as a mere reference for the GPU speed do not include such enhancement.

Figure 7 shows how the master thread creates four CPU threads (CPU context) to invoke the execution on each GPU and manage its resources (i.e allocate device memory, move data to/from the GPU, and so on). Resources created on each CPU thread are not accessible by any other thread, and there is no explicit initialization function for the runtime API (NVIDIA 2008), which makes hard to measure time in a reliable manner, particularly on multi-GPU environments.

For the GPU case, the master process performs the *Parallel Preprocessing* step as usual, generating as many membranes as GPUs are involved in the simulation, and performing the assignment.

At a starting point, the simulation barely exploits GPU resources because the computation begins with a single CUDA thread block (which represents the membrane generated by the *Parallel Preprocessing* step). However, the number of CUDA thread blocks grows exponentially in the *Generation* stage along with the number of membranes, and GPU resources are fully utilized at early stages of the simulation. Another alternative consists of creating a larger

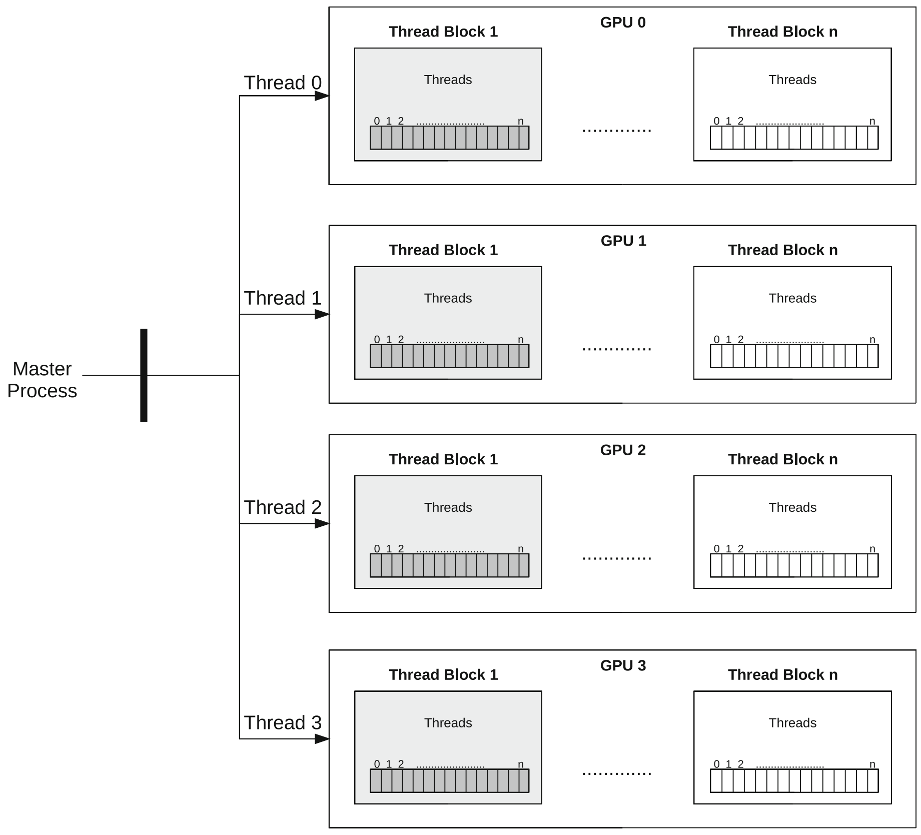


Fig. 7 Data policy on a set of four GPUs. The Master Process creates four CPU threads for dealing with each GPU context. The dynamic generation of thread blocks on each GPU is shown, where the initial membrane is highlighted as a gray block (as input of each GPU context)

set of initial membranes in the *Parallel Preprocessing* step to fulfill that GPU resources are occupied right from the beginning, but we have tested that this initial low usage of GPU resources has a negligible impact, even on tiny benchmarks.

6 Performance evaluation

This section evaluates our P systems implementations on GPUs under different aspects: Comparison versus a CPU counterpart, improvement degree attained through tiling, speed-up obtained when upgrading the GPU generation, and scalability when porting the code to a multi-GPU system or a cluster of GPUs. We now address each of these issues separately.

6.1 GPU versus CPU

Table 3 presents execution times on a high-end CPU and the two high-end Tesla GPUs already introduced for the set of simulations included within our benchmark. It is worth mentioning that for a fair comparison we have selected hardware platforms with a similar cost (investment ranges between 1.500 and 2.000 euros for each single processor).

We see that GPUs are far ahead in performance: Around three orders of magnitude faster, even after considering two additional issues: (1) a feasible implementation which would enable four cores on the CPU using pthreads and (2) the GPU overhead caused by the *initial* and *final* data transfers between CPU and GPU, GPU memory allocation, and CUDA runtime initialization. This overhead is accounted for in the last column of the table, being around 30–35% for the Tesla M2050 case).

6.2 Improvements with tiling

Table 3 also shows us the benefit of using the tiling technique and dynamic queues: up to 2× speed-up factor versus the non-tiling counterpart on GPUs, being more rewarded on the newer Tesla M2050. This is because tiling becomes more effective on caches, and the M2050 enables a 16 KB L1 cache, a larger L2 cache, and provides three times more room for the shared memory to allocate more membranes. GPUs improve significantly the device memory bandwidth through shared memory usage, which is explicitly used by the CUDA programmer. This way, one can control the number of accesses and the way to access on memory bounded applications like ours. Even though the small size of the shared memory decreases GPU occupancy, the benefit of reducing the number of accesses

Table 3 Execution times (ms) on different hardware platforms (CPU vs. GPU) and enabling tiling on the GPU

Number of membranes	CPU Xeon E5530	GPU (wo. tiling)		GPU (w. tiling)		GPU (w. overhead)	
		Tesla C1060	Tesla M2050	Tesla C1060	Tesla M2050	Tesla C1060	Tesla M2050
2^{13}	800.47	0.82	0.62	0.64	0.37	1.17	1.46
2^{14}	1,659.92	1.55	1.20	1.15	0.66	1.68	1.91
2^{15}	3,382.49	2.90	2.30	2.17	1.24	2.67	2.80
2^{16}	6,888.05	5.65	4.37	4.23	2.37	4.66	4.56
2^{17}	14,211.80	11.16	8.71	8.29	4.65	8.73	8.02
2^{18}	28,995.10	22.06	17.15	16.46	9.19	16.52	15.01
2^{19}	59,521.80	44.69	33.16	32.79	18.27	33.01	28.94
2^{20}	121,199.67	88.48	69.03	65.51	36.65	66.12	57.04
2^{21}	247,467.00	171.04	127.85	130.96	73.23	131.43	113.30

We vary the number of membranes and keep constant the number of literals, $l = 256$, and membranes per CUDA block, 8. Communication and initialization times (runtime overhead) are included in the last column for the tiling case

to device memory is much higher and this cost is widely amortized.

Another factor that favors the GPU is the problem instance size. Considering the slowest GPU time, speed-up is $976.18\times$ when the simulation covers 2^{13} membranes and reaches up to $1{,}446.83\times$ when we extend it to 2^{21} membranes. The reason behind that lies in the data bandwidth, which is much higher on GPUs. On small data sets, memory latency plays its role, but when the data set grows exponentially like in our benchmark, bandwidth is what really matters.

Table 4 shows the breakdown of the total execution time for a single GPU executing the benchmark with $n = 21$ variables and using a tiling version. These numbers also evaluate the impact of the data block size, which is limited by the on-chip shared memory space (16 KB for Tesla C1060 and 48 KB for Tesla M2050). Considering those constraints, we were able to measure performance for 2, 4 and 8 membranes per block on the Tesla C1060, and for 2, 4, 8, 16 and 32 membranes per block on the Tesla M2050. Note that the number of global memory accesses and the number of iterations in the *Block Preprocessing* kernel intrinsically depends on the block size. In particular, eight membranes per block require half of the memory accesses and computations (that is, iterations) as compared to the four membranes per block configuration. Similarly, four membranes cut down to a half those required by the two membranes per block case.

Likewise, memory accesses in the *Generation* and *CheckOut* stages are reduced in a similar proportion as long as the block size increases. However, the GPU resources occupancy worsens for the eight membranes per block case, because the shared memory usage per block prevents from allocating more than one block per GPU multiprocessor. As a result, the overall improvement is barely 14% versus the four membranes per block configuration on the Tesla M2050, and then worsens if we continue increasing this parameter (which cannot grow any more on the Tesla C1060 due to shared memory constraints).

6.3 On a set of GPUs

Table 5 shows the performance for the tiling version of the GPU simulator with eight membranes per block, and varying the problem size. We vary here the number of GPUs to study the scalability on a graphics multiprocessor. *Parallel Preprocessing* time spent to arrange the execution on multiple GPUs is ignored, though this time is negligible as the simulation creates just four membranes on a four GPUs configuration.

The multi-GPU environment shines with a linear speed-up along with the number of GPUs. This result is expected as the computational workload is evenly distributed on GPUs. Furthermore, there is more room on each GPU memory space, so higher workloads may be executed. Nevertheless, a P systems simulation creates an exponential workspace to obtain polynomial time solutions for NP-complete problems, so the simulation composed of $n = 22$ variables consumes 3.2 GB and already exceeds our Tesla M2050 capabilities.

6.4 Overall performance

Table 6 summarizes the performance for all software implementations and hardware enhancements exploited through parallel strategies deployed along this paper. For the smallest benchmark, GPU performance achieves an impressive speed-up factor which exceeds three orders of magnitude, and this factor even growing with the problem size. Acceleration reaches its peak for the highest number of membranes that can fit into video memory given our hardware constraints (that is, the $n = 21$ variables case).

Table 4 Execution times (ms) on a single GPU depending on hardware platform and number of membranes per CUDA block for the particular case of a P system composed of 2^{21} membranes

GPU	Block size (in membranes)	2	4	8	16	32
Tesla C1060	Par. and Block Preprocessing	83.59	41.56	20.68	n.a.	n.a.
	Generation and Check Out	113.12	104.02	103.61	n.a.	n.a.
	Total execution time	196.71	145.58	124.29	n.a.	n.a.
Tesla M2050	Par. & Block Preprocessing	32.51	16.17	8.10	4.13	2.02
	Generation and Check Out	85.71	75.90	65.12	65.63	101.24
	Total execution time	118.22	92.07	73.22	69.76	103.26

Communication and initialization times (runtime overhead) are not accounted for (n.a. means "not available" due to shared memory constraints)

Table 5 Execution times (ms) for our P systems simulation on different number of GPUs. We vary the number of membranes and keep constant the number of literals, $l = 256$, and membranes per CUDA block, 8

Number of membranes	Number of GPUs	GPU (implem. without tiling)		GPU (implem. with tiling)	
		Tesla C1060	Tesla M2050	Tesla C1060	Tesla M2050
2^{13}	1	0.82	0.62	0.64	0.37
	2	0.47	0.36	0.37	0.23
	4	0.29	0.24	0.24	0.17
2^{14}	1	1.55	1.20	1.15	0.66
	2	0.85	0.65	0.64	0.38
	4	0.48	0.46	0.38	0.23
2^{15}	1	2.90	2.30	2.17	1.24
	2	1.52	1.20	1.10	0.67
	4	0.83	0.67	0.64	0.39
2^{16}	1	5.65	4.37	4.23	2.37
	2	2.89	2.24	2.53	1.25
	4	1.89	1.32	1.75	0.69
2^{17}	1	11.16	8.71	8.29	4.65
	2	5.65	4.43	4.20	2.38
	4	3.52	2.26	2.89	1.26
2^{18}	1	22.06	17.15	16.46	9.19
	2	11.12	8.63	9.39	4.66
	4	5.64	4.41	4.10	2.41
2^{19}	1	44.69	33.16	32.79	18.27
	2	22.43	16.78	16.47	9.22
	4	11.30	8.41	8.04	4.86
2^{20}	1	88.48	89.03	65.51	36.65
	2	44.34	34.63	32.81	18.32
	4	22.25	17.39	15.92	9.37
2^{21}	1	171.04	127.85	130.96	73.23
	2	85.60	63.99	65.52	36.65
	4	42.95	32.07	32.30	18.89

In general, all speed-ups increase with the problem size, being more remarkable the scalability shown by the multiprocessor version: 3.87× when moving to four GPUs means that we are barely 3% below the optimal line. This outstanding behavior can find a good rationale in the overall amount of cache available for running the code, which is multiplied by a factor of four in a memory-bound algorithm like ours. On the down side, one might expect more from the new Tesla M2050 GPU: 448 cores running at 1.147 GHz deliver 513 GFLOPS, while those 240 cores

Table 6 Summary for the execution times (ms) and speed-up attained by the set of implementations outlined in this paper

Code version	Number of membranes (problem size)				
	2^{13}	2^{15}	2^{17}	2^{19}	2^{21}
1. CPU baseline simulator	800.47	3,382.49	14,211.80	59,521.80	247,467.00
2. Running on Tesla C1060 GPU	0.82	2.90	11.16	44.69	171.04
3. Running on Tesla M2050 GPU	0.62	2.30	8.71	33.16	127.85
4. With tiling on Tesla M2050	0.37	1.24	4.65	18.27	73.23
5. With tiling on 4 Tesla M2050	0.17	0.39	1.26	4.86	18.89
Departure GPU speed-up (2 vs. 1)	976.18×	1,166.37×	1,273.45×	1,331.88×	1,446.83×
Speed-up on M2050 GPU (3 vs. 2)	1.32×	1.26×	1.28×	1.34×	1.33×
Speed-up with tiling (4 vs. 3)	1.67×	1.85×	1.87×	1.81×	1.74×
Speed-up on 4 GPUs (5 vs. 4)	2.17×	3.17×	3.69×	3.75×	3.87×
Overall speed-up factor (5 vs. 1)	4,708×	8,673×	11,279×	12,247×	13,100×

of the Tesla C1060 running at 1.296 deliver 311 GFLOPS. Analytically, we have a 1.65× speed-up factor in raw processing power; in practice, however, our improvement fluctuates around 1.30× , confirming the theory that (1) our simulations are not that demanding on arithmetic intensity and (2) the bottleneck lies more on memory accesses.

Considering the largest problem size and amount of parallelism we were able to expose, we reach a minimum execution time of 18.89 ms on four Tesla M2050 GPUs, each endowed with 448 cores for a total of 1,792 GPU streaming cores. This represents an improvement factor of 13,100× with respect to the departure time given by the original simulator, that is, more than four orders of magnitude.

7 Conclusions

Membrane Computing is an emergent branch of natural computing inspired on the behavior of living cells, whose devices are called P systems. They can provide a polynomial time solution for NP-complete problems by trading space for time because their massively parallel and non-deterministic nature. These characteristics of P systems make a challenge their simulation in either known platform. Up to now, the P system simulation has just been developed in sequential environments. This article tackle its parallelization for a family of recognizer P systems with active membranes, SAT problem on different GPU platforms using the CUDA programming model.

Our data placement analysis reveals that tiling increases the bandwidth by taking advantage of data locality. The effect is that performance improves between 60 and 90% depending on the memory architecture and the way to manage it. We also dedicate some efforts to reduce the cost of preprocessing steps required for applying this technique.

GPUs constitute a good platform to simulate P systems for SAT: The two levels of parallelism that P systems exhibit, one at region level and another one at system level, were exploited by our GPU implementation to reach speed-up factors exceeding three orders of magnitude in our baseline code. Taking this as a departure point, the newest generation of many-core GPU architectures, Nvidia Fermi, enhances the GPU with additional memory resources to develop general purpose applications and more sophisticated P systems models. This fact, combined with a good scalability in our Tesla S2050 composed of four Fermi GPUs, provided us an ideal framework to gain an additional order of magnitude, and thus, the largest simulation composed of 2,097,152 membranes and 256 literals reaches a magnificent speed-up factor of 13,100× versus a high-end Intel Xeon CPU.

Alternative models of P systems which could be used to computationally replicate biological systems within the framework of population and systems biology (i.e., probabilistic/stochastic models) are well positioned to be successfully simulated on multi- and many-core systems due to its arithmetic intensity and large number of iterations required to adjust the model. A high-performance implementation of those simulation models looks promising on GPUs and we have provided some guidelines to succeed by using the CUDA hardware architecture and its programming paradigm. Moreover, the combination of cloud computing and heterogeneous systems along with GPUs represent another trend in modern systems to run even larger simulations and benefit from the promising scalability shown throughout our experimental study.

8 Future work

The Membrane Computing paradigm is being recently applied to study the evolution of complex systems, and

P systems serve as a modeling tool for biological phenomena, mainly within the framework of Systems Biology and Population Dynamics (Păun and Romero-Campero 2008; Pérez-Jiménez and Romero-Campero 2006). Major advantages of Membrane Computing as a formalism for describing and simulating the behavior and evolution of biological systems are the *discretization* and *modularity* of their formal models.

In order to exploit those luring features, a P systems based general framework for modeling ecosystems dynamics was presented in Cardona et al. (2010a). This tool has been used to model real ecosystems computationally, being two good exponents the scavenger birds of the Catalan Pyrenees (Spain) (Cardona et al. 2010b) and the zebra mussel in a reservoir at Ribarroja (Spain) (Cardona et al. 2010a). The ultimate goal is to assist ecologists to adopt a priori management strategies for the real system by executing virtual experiments on a simulator developed ad hoc for these P systems based models (Martínez-del-Amor et al. 2010). Given that those simulations are computationally expensive and quite demanding on memory resources, we are developing a simulator on CUDA to speed up the process by following similar techniques to those described along this paper, which we envision as a starting point for a significant number of applications to benefit from our GPU acceleration methods in the near future.

Acknowledgments This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Spanish MICINN under grants TIN2009-13192 and TIN2009-14475-C04, by the European Commission FEDER funds under grant Consolider Ingenio-2010 CSD2006-00046, and by the Junta of Andalucia of Spain under projects P06-TIC02109 and P08-TIC04200. The authors wish to thank anonymous reviewers for the suggestions received to improve the quality of this article.

References

Alonso S, Fernández L, Arroyo F, Gil J (2008) A circuit implementing massive parallelism in transition P systems. Int J Inf Technol Knowl 2(1):35–42

Cardona M, Colomer MA, Margalida A, Palau A, Pérez-Hurtado I, Pérez-Jiménez MJ, Sanuy D (2010a) A computational modeling for real ecosystems based on P systems. Nat Comput. doi: 10.1007/s11047-010-9191-3

Cardona M, Colomer MA, Margalida A, Pérez-Hurtado I, Pérez-Jiménez MJ, Sanuy D (2010b) A P system based model of an ecosystem of some scavenger birds. LNCS 5957:182–195

Cecilia JM, García JM, Guerrero GD, Martínez-del-Amor MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2010a) Simulating a P system based efficient solution to SAT by using GPUs. Int J Log Alg Prog 79(6):317–325

Cecilia JM, García JM, Guerrero GD, Martínez-del-Amor MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2010b) Simulation of P systems with active membranes on CUDA. Brief Bioinform 11(3):313–322

Cecilia JM, García JM, Ujaldón M (2010c) CUDA 2D stencil computation for the Jacobi method. In: Proceedings of the 10th international workshop on state-of-the-art in scientific and parallel computing, Reykjavik, Iceland

Cook SA (1971) The complexity of theorem-proving procedures. In STOC '71: Proceedings of the third annual ACM symposium on theory of computing, New York, USA, pp 151–158

Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson DA, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on State-of-the-art multicore architectures. In: Proceedings ACM/IEEE Supercomputing 2008, pp 1–12

Díaz D, Graciani C, Gutiérrez-Naranjo MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2009) Software for P systems. In Paun Gh, Rozenberg G, Salomaa A (eds) The Oxford handbook of membrane computing. Oxford University Press, Oxford, pp 437–454

García-Quismondo M, Gutiérrez-Escudero R, Pérez-Hurtado I, Pérez-Jiménez MJ, Riscos-Núñez A (2010) An overview of p-lingua 2.0. LNCS 5957:264–288

Krishnamoorthy S, Baskaran MM, Bondhugula U, Ramanujan J, Rountev A, Sadayappan P (2010) Effective automatic parallelization of stencil computation. In: Proceedings 2010 ACM conference on programming languages, design and implementation, pp 235–244

Li J, Hu X, Pang Z, Qian K (2009) A parallel ant colony optimization algorithm based on fine-grained model with GPU acceleration. Int J Innov Comput Inf Control 5(11):3707–3715

Martínez-del-Amor MA, Pérez-Hurtado I, Pérez-Jiménez MJ, Riscos-Núñez A, Colomer MA (2010) A new simulation algorithm for multienvironment probabilistic P systems. In BIC-TA'2010: proceedings 2010 IEEE fifth international conference on bio-inspired computing: theories and applications, vol 1, pp 59–68

Mussi L, Cagnoni S (2009) Particle swarm optimization within the CUDA architecture. In: GECCO conference

Nguyen V, Kearney D, Gioiosa G (2010) An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. Int J Log Alg Prog 79(6):383–439

NVIDIA (2008) CUDA programming guide 2.0

NVIDIA (2010) Next generation CUDA architecture. Code named Fermi. http://www.nvidia.com/object/fermi_architecture.html

Păun G (2000) Computing with membranes. J Comput Sys Sci 61:108–143 (TUCS report no 208)

Păun G (2002) Membrane computing: an introduction. Springer, Berlin

Păun G (2009) Active membranes. In: Păun Gh, Rozenberg G, Salomaa A (eds) The Oxford handbook of membrane computing. Oxford University Press, Oxford, pp 282–301

Păun G, Romero-Campero-FJ (2008) Membrane computing as a modeling framework. LNCS 5016: 168–214

Pérez-Jiménez MJ, Romero-Campero FJ (2006) P systems, a new computational modelling tool for systems biology. LNCS 4220:176–97

Pérez-Jiménez MJ, Romero-Jiménez Á Sancho-Caparrini F (2003) Complexity classes in models of cellular computing with membranes. Nat Comput 2(3):265–285

Pospichal P, Jaros J (2009) GPU-based acceleration of the genetic algorithm. In: GECCO conference

Stutzle T (1998) Parallelization strategies for ant colony optimization. Springer, Berlin

Qasem M (2009) WinSAT. http://users.ecs.soton.ac.uk/mqq06r/winsat