Adaptative parallel simulators for bioinspired computing models

Miguel Á. Martínez-del-Amor, Ignacio Pérez-Hurtado, David Orellana-Martín, Mario J. Pérez-Jiménez

Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Seville, Spain

Keywords:
Bioinspired computing
Membrane computing
P system
Programming language
Parallel computing
GPU computing

A B S T R A C T

In the Membrane Computing area, P systems are unconventional devices of computation inspired by the structure and processes taking place in living cells. Main successful P system applications lie in computability and computational complexity theories, as well as in biological modelling. Given that models become too complex to deal with, simulators for P systems are essential tools and their efficiency is critical. In order to handle the diverse situations that may arise during the computation, these simulators have to take into account that worst-case scenarios can happen, even though they rarely occur. As a result, there is a significant loss of performance. In this paper, the concept of adaptative simulation for P systems is introduced to palliate this problem. This is achieved by passing high-level information provided directly by P system model designers to the simulator, helping it to better adapt to the target model. For this purpose, an existing simulator for an ecosystem modelling framework, named Population Dynamics P systems, is extended to include the information of modules, that are usually employed to define ecosystem models. Moreover, the standard description language for P systems, P-Lingua, has been re-engineered in its version 5. It now includes a new syntactical item, called feature, to express this kind of high-level semantic information. Experiments show that this simple adaptative simulator supporting modules as features doubles the performance when running on GPUs and on multicore processors.

1. Introduction

Membrane Computing is a paradigm of computation inspired by the behaviour and the structure of living cells, introduced by Gh. Păun in 1998 [1]. This paradigm has led to several kinds of massively-parallel computing devices known as membrane systems or, simply, P systems. Applications of P systems range from contributions within computability and computational complexity theories (e.g. seeking new frontiers in the P vs NP problem [2–4]), to computational modelling in life sciences such as Systems Biology [5,6] (e.g. bacterium quorum sensing [7]) and Population Dynamics [8,9] (e.g. butterfly *Pieris oleracea* in eastern North America [10]).

Although many variants of such systems have been defined so far, the main common ingredients are a compartmentalised structure given by *membranes* or *cells* and a multiset of *objects* within each region. P systems evolve from one state to the next one by a pre-defined set of *rules* in a transition step. A sequence of such states or *configurations* is a *computation* of the system. The key feature in their applications is the massive and double parallelism nature: rules are executed in a maximally-parallel manner within each membrane, while all membranes evolve in parallel at the system level. In order to keep control of the computation taking place, there is a *global clock* that synchronises the execution of rules at the P system level.

Simulating P systems is of huge importance to develop validation, verification and virtual experimentation tools [11,12]. For instance, *Population Dynamics P systems* (PDP systems, for short) were conceived and successfully employed for ecosystem modelling [13], and their efficient simulation is the key for fast model designing process [9], parameter calibration [14] and experimentation [15].

Many simulation tools have been developed since the paradigm was born [11,16]. The general scheme of a simulator design is structured in three modules: definition of the P system to be simulated, simulation of one or more computations, and collection of output data from the model computation. One of the most generic approaches and widely employed is *P-Lingua* [17,18], a programming language designed specifically for the definition of P systems. The P-Lingua framework includes a Java library (called *pLinguaCore*) that allows reading P system descriptions specified in a plain-text file. The framework can either simulate a computation of the described P system or alternatively export the description to other formats that can serve as input to external simulators.

The natural way to accelerate these simulations is to exploit their massive parallelism by leveraging *High Performance Computing* technologies [19]. Implementing P system parallelism in modern parallel processors is not straightforward, mainly due to its non-deterministic and synchronous nature, but it has been shown that *Graphics Processing Units (GPUs)* can be employed successfully for this task. For example, some variants of P systems that have been simulated on GPUs are: P systems with active membranes [20], a specific family of P systems with active membranes solving SAT [21,22], kernel P systems on PSO [23], and spiking neural P systems through a matrix representation [24]. In this sense, a project called *PMCGPU* [25] includes some P system simulators implemented on GPUs. Moreover, distributed systems [26], multicore processors [27] and FPGAs [28] have been explored as alternatives to speedup the simulation.

There are two main approaches when designing P system simulators: *specific* and *generic* approaches [29]. The former is conceived to simulate only certain families of P systems of the same type (looking for best efficiency, even hard-coding information of the P systems, e.g. [30]). The latter aims at simulating any P system of a certain variant (looking for flexibility, allocating large arrays to store all defined objects in the system even though finding all of them at the same time in a configuration is improbable, e.g. [20]). Generic simulators are required in certain scenarios, such as in ecosystem modelling, so that the model designers would use just one simulator for all their models [31]. However, this comes at the expense of performance degradation, because the simulators are developed to support many worst case-scenarios that normally do not take place in models. For example, generic simulators work over the whole set of rules of the target P system, performing a blind search for those that are applicable. In this case, the worst case scenario being assumed is that all the rules can be applicable at every step. Attempts to overcome this issue have been considered. For example, in [22], rules having more interactions are grouped in order to avoid communication between GPU threads. Nonetheless, this is specific to the variant of P systems with active membranes and the paper does not clarify how the global clock synchronisation is satisfied.

In general, when designing models of P systems, designers usually think first on a general scheme for their solution in form of a workflow of *modules* that can be executed sequentially or in parallel [9]. In each module, only a subset of rules can be applied. This kind of high-level information has always been skipped in generic simulators. In this work, the idea of bridging the gap between the model designer and the simulator developer is explored through the concept of adaptative simulator. It will enable designers to specify certain high-level semantic features (such as modules) that will permit simulators to better adapt to the model. This opens the way to discard worst case-scenarios that will never take place; for example, avoiding checking every single rule defined in the model at every transition step. It is noteworthy that this approach lies in the middle ground between generic and specific approaches: an adaptative simulator can handle any P system of the supported variant but receiving enriched information along with the P system description.

In short, the main contributions of the paper are the following: (1) introducing the concept of *adaptative simulator* in the world of Membrane Computing; (2) testing this concept by developing an *adaptative simulator for PDP systems* as a case study, which takes advantage of the modules defined usually in the algorithmic scheme of ecosystem models; (3) extending the new *P-Lingua (version 5)* with a new way to specify semantic information called *features*; (4) definition of a specific syntax to define modules; (5) extending the PDP system simulators within PMCGPU project to support features by reading the new files generated with P-Lingua 5 and processing this module information; (6) developing the adaptative multicore (OpenMP) simulator so that the number of internal loops iterations decrease substantially; (7) developing the adaptative GPU (CUDA) simulator so that internal loops iterations are reduced as well as the internal data structures are compacted, which leads to better memory coalesced accesses and concurrent execution of kernels by using CUDA streams; (8) benchmarking using an extension of the PDP system model for tritrophic interactions[1] over the following parallel hardware: i7 CPU, Xeon CPU, Tesla P100 GPU, Tesla K40 GPU and GTX 1050Ti GPU, including a detailed speedup and profiling analysis.

The rest of the paper is organised as follows. The Introduction section will provide a summary of the concepts required to understand this work, covering PDP systems, P-Lingua and GPU computing. Section 2 describes the base algorithm for PDP systems, called DCBA, and its implementation in both multicore and GPU processors. Section 3 presents the new version of P-Lingua implemented supporting features. Section 4 depicts the extension made to DCBA and the parallel implementations to support modules. Section 5 provides a succinct overview to the model employed for benchmarking, and a detailed analysis of the obtained performance results. Finally, Section 6 ends the paper with conclusions and planned future work.

1.1. Population dynamics P systems

In this section, the formal definition of a PDP system is provided. First, the syntactical ingredients are introduced, and after that, some semantic aspects are discussed.

Definition 1. A Population Dynamics P system of degree (q, m) with $q \geq 1$, $m \geq 1$, and taking $T \geq 1$ time units, is a tuple

$$\Pi = (G, \Gamma, \Sigma, T, R_E, \mu, R, \{f_{r,j} : r \in R, 1 \leq j \leq m\},\ \{\mathcal{M}_{ij} : 1 \leq i \leq q, 1 \leq j \leq m\})$$

where:

- $G = (V, S)$ is a directed graph. Let $V = \{e_1, \dots, e_m\}$ whose elements are called environments, and S is a set of pairs of environments representing the arcs of the graph;
- Γ is the working alphabet and $\Sigma \subsetneq \Gamma$ is an alphabet representing the objects that can be present in the environments;
- T is a natural number that represents the simulation time of the system;
- R_E is a finite set of communication rules between environments of the form

  $$(x)_{e_j} \xrightarrow{p} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$$

  where $x, y_1, \dots, y_h \in \Sigma$, $(e_j, e_{j_l}) \in S$ $(1 \leq l \leq h)$ and p is a computable function from $\{1, \dots, T\}$ to $[0, 1]$. By default, and if p is not specified for a rule, then it is the constant function 1. h may be different in each rule. These functions verify the following:

  - For each $e_j \in V$ and $x \in \Sigma$, the sum of functions associated with the rules whose left-hand side is $(x)_{e_j}$ is exactly 1.

- μ is a membrane structure consisting of q membranes injectively labelled by $1, \dots, q$. The skin membrane is labelled by 1. A electrical charge from the set $EC = \{0, +, -\}$ is also associated with each membrane.

[1] An ecosystem with three trophic levels.

- R is a finite set of evolution rules of the form

  $u[\, v\,]_i^{\alpha} \rightarrow u'[\, v'\,]_i^{\alpha'}$

  where $u, v, u', v' \in M_f(\Gamma)$ (the set of all finite multisets over Γ), i $(1 \leq i \leq q)$, $u + v \neq \emptyset$ and $\alpha, \alpha' \in \{0, +, -\}$. The following restriction must hold:

  – If $(x)_{e_j}$ is the left-hand side of a rule from R_E, then none of the rules of R has a left-hand side of the form $u[v]_1^{\alpha}$, for any $u, v \in M_f(\Gamma)$ and $\alpha \in \{0, +, -\}$, having $x \in u$.

- For each $r \in R$ and for each j $(1 \leq j \leq m)$, $f_{r,j} : \{1, \ldots, T\} \longrightarrow [0, 1]$ is computable. These functions verify the following:

  – For each $u, v \in M_f(\Gamma)$, i $(1 \leq i \leq q)$, $\alpha, \alpha' \in \{0, +, -\}$ and j $(1 \leq j \leq m)$ the sum of functions associated with j and the set of rules whose left-hand side is $u[v]_i^{\alpha}$ and whose right-hand side has polarisation α', is the constant function 1.

- For each j, $(1 \leq j \leq m)$, $\mathcal{M}_{1j}, \ldots, \mathcal{M}_{qj}$ are finite multisets over Γ, describing the objects initially placed within the regions in environment e_j (also known as initial configuration).

In other words, a PDP system consists of m environments $e_1, \ldots, e_m$ linked by the arcs from a directed graph G. Each environment e_j contains a P system, $\Pi_j = (\Gamma, \mu, R_{\Pi_j}, \mathcal{M}_{0j}, \ldots, \mathcal{M}_{q,j})$, of degree q (i.e. number of membranes) where every rule $r \in R$ has associated a computable function $f_{r,j}$ (specific for environment j) forming R_{Π_j}. Let us remark that all Π_j have the same *skeleton*; that is, membrane structure μ and set of rules from R. Specifically, rules from R are also called skeleton rules.

A *computation* is a sequence of *configurations* through transition steps, at maximum T. A *configuration* of the system at an instant t is a tuple of multisets of objects present in the m environments and at each region of each Π_j, together with the polarisation of each membrane in each P system (0, + or −). At the initial configuration of the system we assume that all environments are empty and all membranes have a neutral polarisation. We also assume that a global clock exists, marking the time for the whole system; that is, all membranes and the application of all rules (from R_E and from all R_{Π_j}) are synchronised in all environments.

The P system can pass from one configuration to the next one by using the rules from $\bigcup_{j=1}^{m} R_{\Pi_j} \cup R_E$ as follows: at each transition step, the rules to be applied are selected according to the *probabilities* assigned to them, and all applicable rules are simultaneously applied in a maximal way (i.e. no more rules can be further applicable). This is done by consuming the left-hand side of the rules, and generating the right-hand side afterwards. For rules in R_{Π_j}, the charge of the (active) membrane can be changed. In this sense, the consistency of charges must be maintained: in order to simultaneously apply several rules from R_{Π_j} to the same membrane, all rules must have the same electrical charge on their right-hand side.

For more information about the syntax of the models, refer to [9,31]. The semantics discussed in this section are the basics of the functioning of the framework. However, specific details will depend upon the simulation algorithm that will reproduce the computation [32,33].

1.2. P-Lingua and pLinguaCore version 4

P-Lingua [17,18] is a software framework for Membrane Computing which includes a definition language for P systems (also called P-Lingua). A file defining a P system model with P-Lingua language in plain text is also known as *P-Lingua file*, and has the *.pli* extension. Several parsing tools and simulators have been developed within the framework in a series of versions from 1 to 4. The main tool is the Java library pLinguaCore, containing three distinguished components:

- A parser for reading input P-Lingua files and checking constraints related to the corresponding variant. In order to achieve this, the first line of a P-Lingua file should include a P system model type declaration by using a unique identifier. There are several predefined P system models that can be used, each one with its own identifier, for instance `transition`, `membrane_division`, `tissue_psystems`, and `probabilistic`. The analysis of semantic ingredients, such as rule patterns, is hard-coded for each variant. Several versions of pLinguaCore were released to cover different types of models.
- For each P system model type, the pLinguaCore library includes one or more built-in generic simulators, each one implementing a different simulation algorithm. For instance, Population Dynamic P systems [9] (`probabilistic` identifier in P-Lingua) can be simulated inside the library by applying three different algorithms: `BBB`, `DNDP`, or `DCBA` [32,34] (discussed in Section 2.1).
- Alternatively, pLinguaCore library is able to transform input P-Lingua files to other formats such as XML or binary in order to feed external simulators. The generated files for the given P systems are free of syntactic and some semantic errors since the transformation is done after the parser analysis. Several external simulators use this tool, like in the *PMCGPU* project (Parallel simulators for membrane computing on GPU) [19,25], where two GPU simulators read binary files generated by pLinguaCore.

The main advantage of P-Lingua framework is its ease of use. The syntax is close to the scientific notation employed in Membrane Computing. Thus, researchers can write and debug P systems in a familiar way using modules, parameters, variables, iterators and other programming ingredients. For instance, several real ecosystem models have been written, debugged and simulated using the P-Lingua framework [9,15,31].

However, one of the main drawbacks of this framework is the poor flexibility to define semantic ingredients such as derivation modes, i.e, the way in which P system computations should be simulated. Indeed, P-Lingua 4 includes only one fixed and hard-coded derivation mode for each variant (with unique identifier). Simulators are also hard-coded and fixed in pLinguaCore, containing one or more simulators for each derivation mode (associated to the P system model type defined in the first line of the P-Lingua file). Moreover, P-Lingua users are not able to tune simulators inside pLinguaCore to get more efficient simulations since all of them implement the same API. Finally, a collateral downside is related to the time and memory consumption to parse very large P systems. All these issues have been tackled in the new version 5 (Section 3).

1.3. GPU computing

In what follows, we summarise the key concepts of GPU computing required for the understanding of this work. Modern *Graphics Processing Units (GPUs)* are massively parallel co-processors for computing acceleration [35]. This is easily achievable by using *CUDA*, a programming model introduced by NVIDIA in 2007. From now, we will focus on CUDA, given that it is the technology employed in this work, and the concepts are similar in other programming models such as OpenCL and ROCm. CUDA abstracts the underlying architecture in GPUs, focusing only on managing *threads* and memory. Moreover, it is a *heterogeneous* model where the GPU (a.k.a. device) and CPU (a.k.a. host) have separated execution and memory spaces. Programmers have to define the code executed by the threads (called *kernel*) and manually manage the access to memory. This has to be efficiently handled by considering the memory hierarchy and coalesced access to data (contiguous threads accessing consecutive positions in memory).

Threads are distributed into a grid where they are arranged into synchronised and cooperating *blocks*. Kernels are assigned to each grid, and as long as there are enough resources in the hardware, it is possible to launch concurrent kernels by assigning them to different CUDA *streams*. A stream represents a flow of execution of kernels, and there is a default stream that cannot run in parallel with others. Inside GPUs of today, one can find from hundreds to thousands of cores distributed in multiprocessors. Moreover, several Gigabytes of *global* memory (large but slow) are built in the cards, and accessible by all threads executed in the device. The host is in charge of sending data to and retrieving data from the device, and calling kernels that are off sourced to there [35]. Specific features of each GPU hardware are normally defined by the compute capability number [36]. Concurrent access by threads to a single datum in GPU memories can be accomplished without race conditions through specific *atomic operations (e.g. addition, multiplication, swapping, etc.).*

2. DCBA

In this section, the target simulation algorithm for PDP systems, named DCBA, is depicted. Firstly, the theoretical basis to understand the algorithm is provided. Second, the parallel design of DCBA is described.

2.1. Algorithm description

There are three main simulation algorithms for PDP systems: *BBB*, *DNDP* and *DCBA* [32,33]. In short, *Binomial Block Based* (BBB) algorithm first groups rules having the same left-hand side (Definition 2) into blocks, shuffle and iterate them, executing rules according to a multinomial random variate according to their probabilities. In contrast, *Direct Non Deterministic distribution with Probabilities* (DNDP) algorithm randomly loops over the rules calculating for each one a binomial random variate according to the probability. DNDP uses a second stage to satisfy maximality by executing, one by one, the remaining rules as much as possible.

Each algorithm aimed at improving the accuracy in which the reality is mapped to the models. Perhaps, the most difficult feature to handle by the simulation algorithms is the competition of objects between rules. In this sense, both DNDP and BBB have a common drawback: the distribution of objects among competing rules leads to high deviations in the results. This is why the latest algorithm, called *Direct distribution based on Consistent Blocks Algorithm* (DCBA) [32], was defined for. The key idea of DCBA is to implement a proportional distribution of objects among rules grouped into consistent blocks (a concept similar to, but not the same than, blocks in BBB), while dealing with consistency and probabilities.

In what follows, the key concepts required for DCBA are going to be described. Then, a brief pseudocode and explanation of the algorithm is provided. For more information, please refer to [31,32]. Firstly, rules in R and R_E can be classified into consistent blocks having the same left-hand side and the same charge in the right-hand side, following the Definitions 2 and 3.

Definition 2. The left and right-hand sides of the rules are defined as follows:

(a) Given a communication rule $r \in R_E$ of the form $(x)_{e_j} \xrightarrow{p} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ where $e_j \in V$ and $x, y_1, \ldots, y_h \in \Sigma$:

- The left-hand side of r is $LHS(r) = (e_j, x)$.
- The right-hand side of r is $RHS(r) = (e_{j_1}, y_1) \cdots (e_{j_h}, y_h)$.

(b) Given a skeleton rule $r \in R$ of the form $u[v]_i^{\alpha} \to u'[v']_i^{\alpha'}$ where $1 \le i \le q$, $\alpha, \alpha' \in \{0, +, -\}$ and $u, v, u', v' \in M_f(\Gamma)$:

- The left-hand side of r is $LHS(r) = (i, \alpha, u, v)$. The charge of $LHS(r)$ is $charge(LHS(r)) = \alpha$.
- The right-hand side of r is $RHS(r) = (i, \alpha', u', v')$. The charge of $RHS(r)$ is $charge(RHS(r)) = \alpha'$.

Definition 3. Rules from R and R_E can be classified into consistent blocks as follows: (a) the rule block associated to $(i, \alpha, \alpha', u, v)$ is $B_{i,\alpha,\alpha',u,v} = \{r \in R : LHS(r) = (i, \alpha, u, v) \wedge charge(RHS(r)) = \alpha'\}$; and (b) the rule block associated with (e_j, x) is $B_{e_j,x} = \{r \in R_E : LHS(r) = (e_j, x)\}$.

Definition 4. Two consistent blocks $B^1_{i_1,\alpha_1,\alpha'_1,u_1,v_1}$ and $B^2_{i_2,\alpha_2,\alpha'_2,u_2,v_2}$ compete for objects when the following holds:

(a) The two blocks are mutually consistent. That is, if $i_1 = i_2 \wedge \alpha_1 = \alpha_2$ then $\alpha'_1 = \alpha'_2$.

(b) Their left-hand sides overlap. That is, either of the following conditions hold:

- If $i_1 = i_2$ and $\alpha_1 = \alpha_2$ then $u_1 \cap u_2 \neq \emptyset$ and $v_1 \cap v_2 \neq \emptyset$
- If $i_1 \neq i_2$ but i_1 is the parent membrane of i_2, then $v_1 \cap u_2 \neq \emptyset$, and vice versa.

For the sake of simplicity, the left-hand side of a block B, denoted by $LHS(B)$, is the left-hand side of any rule in the block. Note that a block $B_{i,\alpha,\alpha',u,v}$ determines a consistent set of rules; that is, all skeleton rules in a block are mutually consistent (generating the same charge to the same membrane). Let us recall that the sum of probabilities of the rules belonging to the same consistent block must be equal to 1. In particular, rules with probability equal to 1 form individual blocks. Moreover, rules having exactly the same left-hand side (LHS) belong to the same block, but rules with overlapping (but different) left-hand sides are classified into different blocks. Overlapping LHS leads to object (resource) *competition*. This is a critical aspect to be managed by simulation algorithms (see Definition 4).

DCBA tackles the resource competition issue by performing a proportional distribution of objects among competing blocks. Algorithm 1 describes the main loop of the DCBA, which is structured in two stages: selection and execution. Selection stage consists of three phases: Phase 1 distributes objects to the blocks in a certain proportional way, Phase 2 ensures *maximality* by checking the maximal number of applications of each block, and Phase 3 translates from block to rule applications by calculating random numbers using a multinomial distribution.

`INITIALISATION` procedure constructs a static distribution table $\mathcal{T}_j$ for each environment, where:

- Column labels correspond to every consistent block $B_{i,\alpha,\alpha',u,v}$ and $B_{e_j,x}$, for environment j.
- Row labels are pairs (x, i), for all objects $x \in \Gamma$, and $0 \le i \le q$. i $= 0$ denotes the environment.
- Each entry at row (x, i) and column B is:
  - $\frac{1}{k}$, if $LHS(B) = (e_j, x)$ and $i = 0$

Algorithm 1 DCBA MAIN PROCEDURE

```
Require: A PDP system $\Pi$ of degree $(q, m)$, $T \geq 1$ (time units), $A \geq 1$ (accuracy parameter), and an initial configuration $C_0$.
 1: for $j \leftarrow 1$ to $m$ do
 2:     $(\mathcal{T}_j) \leftarrow$ INITIALISATION $(j, \Pi)$
 3: end for
 4: for $t \leftarrow 0$ to $T-1$ do
 5:     SELECTION:
 6:     for $j \leftarrow 1$ to $m$ do
 7:         $B^j_{sel} \leftarrow \emptyset$, $R^j_{sel} \leftarrow \emptyset$
 8:         $(\mathcal{T}^t_j, C'_t, B^j_{sel}) \leftarrow$ PHASE 1 $(j, \Pi, A, C_t, \mathcal{T}_j)$      ▷ (Distribution of objects)
 9:         $(C'_t, B^j_{sel}) \leftarrow$ PHASE 2 $(j, \Pi, C'_t, B^j_{sel}, \mathcal{T}^t_j)$      ▷ (Maximality)
10:         $(R^j_{sel}) \leftarrow$ PHASE 3 $(j, \Pi, B^j_{sel})$      ▷ (Probabilistic distribution)
11:     end for
12:     EXECUTION:
13:     for $j \leftarrow 1$ to $m$ do
14:         $(C_{t+1}) \leftarrow$ PHASE 4 $(j, \Pi, C'_t, R^j_{sel})$
15:     end for
16: end for
```

- $\frac{1}{k}$, if $LHS(B) = (j, \alpha, u, v)$ and either $i = j$ and $x^k \in v$ or j is parent of i and $x^k \in u$. In other words, if the object in the row label x appears in its associated compartment i with multiplicity[2] k in the left-hand side of the block B of the column label.
- 0, otherwise.

Moreover, two multisets, B^j_{sel} and R^j_{sel}, are also initialised to be empty. They are going to be used to store the selection of blocks and rules, respectively.

Before starting SELECTION, a copy of the current configuration C_t is made, named C'_t. PHASE 1 carries out the distribution of objects among the blocks by using the static tables $\mathcal{T}_j$. These tables are modified by three different filters: FILTER 1 (removing columns of non-applicable blocks b due to mismatch charges in $LHS(b)$ and C'_t), FILTER 2 (discarding the columns with objects in the LHS not appearing in C'_t) and FILTER 3 (discarding empty rows). In order to get a set of mutually consistent blocks, the consistency condition is checked after applying FILTER 1 and FILTER 2. If it fails, the simulation process can either halt with an error message or non-deterministically construct a subset of mutually consistent blocks.

After applying these filters, a *dynamic table* $\mathcal{T}^t_j$ is generated, whose values are normalised as follows: (1) a sum per row is computed, and (2) for each row, the corresponding multiplicity of the object in the multiset C'_t is multiplied by the original value in the dynamic table to the square (i.e. $(\frac{1}{k})^2$), and divided by the sum of the row calculated before in (1).

Finally, the number of applications for each block is calculated by computing the minimum value per column. This number is annotated in B^j_{sel} for the corresponding block associated with the column, and used to consume the objects in the LHS from C'_t. However, this application might not be maximal due to rounding, so in order to be more accurate this phase is repeated A times (called the A parameter).

As mentioned, it might be possible that some blocks are still applicable because of rounding artefacts in the distribution process. Due to the requirements of P systems semantics, a maximality phase (PHASE 2) is applied. It imposes a random order to the remaining columns (blocks) in each dynamic table and by looping over them, the remaining applications are distributed one by one according to the arbitrary order. These last applications are also annotated in B^j_{sel}. Thus, some blocks appearing deep in the random order might not be applied.

In order to complete the SELECTION stage of the algorithm, DCBA has to provide a multiset of rules (rule applications) to the EXECUTION stage. After PHASE 2, only block applications in B^j_{sel} are available. Hence, PHASE 3 calculates rule section. It applies the corresponding probabilistic distribution at the local level of each block following a multinomial random variate $M(N, g_1, \ldots, g_l)$, where N is the number of applications of the block, and $g_1, \ldots, g_l$ are the probabilities associated with the rules $r_1, \ldots, r_l$ within the block at step t, respectively. These random applications are then annotated in R^j_{sel}.

Finally, the EXECUTION stage is applied. For each selected rule r in R^j_{sel}, the $RHS(r)$ is added to C'_t, and $charge(RHS(r))$ is assigned to the corresponding membrane (now safely, thanks to satisfying consistency condition). Finally, C'_t is just the next configuration C_{t+1}. In order to loop through the configurations, in each iteration, DCBA cleans up B^j_{sel} and R^j_{sel}, and deletes the dynamic distribution tables, starting over again with $\mathcal{T}_j$.

2.2. Parallel implementation

A key advantage of DCBA over DNDP and BBB is that the distribution of objects to the blocks is made through a matrix. Thus, it does not require a random order over rules and blocks (at least for phases 1, 3 and 4). This facilitates a parallel implementation. On the contrary, a major downside of DCBA is its low efficiency, which requires to construct a large table with dynamic structure. Moreover, this table is sparse: it is very rare that objects participate in every block's left-hand side. For this reason, an efficient implementation of DCBA should avoid the construction of the whole distribution table. In pLinguaCore, a hash table [32] was employed to make a dense representation. In contrast, parallel implementations used different approaches.

The project ABCD-GPU, developed inside PMCGPU, was conceived to provide parallel implementations to DCBA. First, the distribution table was replaced by a virtual table [37] that translates the operations over the table to operations over the rule blocks information, plus requiring only three new arrays. When DCBA loops over columns, this implementation iterates over the rule blocks. Information of each column can be extracted from the corresponding block's LHS. Loops over rows are performed by accessing again the blocks' LHS, but storing the partial results in a global array. The three new arrays aforementioned are:

- *activationVector*: the information of filtered blocks is stored here as boolean values, having one position for each rule block and environment.

[2] Number of instances of an element in a multiset.

- *addition*: the row sums calculated for normalisation are stored here, having one number per each tuple (object, region, environment). Using floating numbers might lead to imprecise representations. In order to solve this issue, an alternative representation is also employed (from now on, *n/d representation*), where the numerator and denominator are stored in two separated arrays (named *numerator* and *denominator*). Consequently, the divisions and multiplications are made just once, when calculating the column minimums. In rows having many competing blocks, the n/d representation can overflow, so floating numbers are the only solution.
- *MinN*: the minimum numbers calculated per column are stored here.

ABCD-GPU started with a multi-core version [27,37] based on C++ and OpenMP. Three approaches for parallelisation were taken: by environments, by simulations and both environment and simulation. Given that PDP systems have a probabilistic nature, it is required to run several simulations in order to increase the confidence of the results and get accurate average temporal series. These simulations can run independently of each other, and hence, it has been the best way to parallelise the simulator. Moreover, experiments showed that PDP system simulations were memory bandwidth bound. Using a CPU i7 Sandy Bridge (1×4), a peak speedup of 2.5x was obtained against a serial implementation.

In [38], an implementation in CUDA was introduced. This simulator seeks for fine-grain parallelism, since only parallelising by simulations is not enough for a GPU. The following grid configuration was employed: a thread block for each environment in the x axis and for each simulation in the y axis; 256 threads per thread block that loop in tiles over the rule blocks[3] for selection phase kernels or loop over rules for execution phase. Although this design is tight to the simulated model (number of environments, rule blocks) and the user preferences (simulations to run), the number of CUDA threads is large enough. This design is illustrated in Fig. 1.

The implementation was made through 7 different kernels [38,39], 3 for phase 1, 2 for phase 4, and one for each of the remaining phases:

- Kernel for filters (phase 1): filters 1 and 2 are applied, plus consistency checking. Only error warning and halting is supported (see Section 2.1), given that a model presenting inconsistency is found to be rare and hard to control by designers. Filter 3 is not required with the virtual table.
- Kernel for normalisation and column minimum (phase 1): the row sums of the table are calculated and employed to compute the minimum values per column, denoting the applications of the corresponding rule block. When using *accurate* mode, the n/d representation is employed. In order to save computations, and at initialisation, the total sums from $\mathcal{T}_j$ are pre-calculated, and those non-applicable rule blocks subtract just the *numerator* vector accordingly. When *non accurate* mode is used, floating numbers are used per row, so applicable rule blocks add their corresponding values in *addition* vector.
- Kernel for updating (phase 1): once the applications of each rule block is computed, this kernel updates the configuration by subtracting the corresponding LHS, annotates that number to B^j_{sel} and applies filter 2 again.

[3] From now, we will call rule blocks to the consistent blocks formed out of the PDP system, in order to distinguish them from the thread blocks of CUDA programming model.

- Kernel for phase 2: two approaches, static and dynamic, are taken here since this phase is inherently sequential. The static one performs a random loop over the remaining blocks. Randomness is simulated by the GPU scheduler with atomic operations among threads. This results to be sufficient according to experiments [39]. The dynamic approach detects competing blocks using shared memory. After that, the maximal applications are calculated in parallel for each non-competing group of blocks [38].
- Kernel for phase 3: in this phase, a library called CURNG_BINOMIAL [37] was developed in order to generate a binomial random variate. As in pLinguaCore, only constant probabilistic functions $f_{r,j}$ and p_r along time are supported (they may differ only between environments). The number of applications per rule is annotated in R^j_{sel}.
- Kernels for phase 4: rules are visited by threads, and only those with applications in R^j_{sel} are considered. The RHS is generated in the configuration using atomic operations for addition, and the membrane charges are safely updated. Two kernels implement this stage separately for rule blocks belonging to the skeleton of the PDP system and another for environment communication rule blocks.

Concerning the internal data layout of the C++ implementation, the information of the rules and blocks are compacted by separating the LHS, RHS and probability in different arrays. These are accessed by indexes assigned to each rule block and rule. The configuration of the PDP system is implemented as follows: an array of 8-bit integers for the charges of the membranes and an array of 32-bit unsigned integers for the multisets within the regions, storing the multiplicity of each object defined in the alphabet. *activationVector* is bitwise coded: an array of 32-bit words stores in each position the activation flags for 32 rule blocks. When using the n/d representation, two arrays of 32-bit unsigned integers are used (for *numerator* and *denominator*), otherwise an array of 32-bit floats is used for *addition*. *MinN* is an array of 32-bit unsigned integers. When using the GPU, the arrays are stored in global memory given that they get updated in each step. Some parts of the rule information get stored in constant memory given that they remain unchanged during the simulation, showing a slight performance improvement according to our experiments. Further details can be consulted in the source code at PMCGPU website [25].

ABCD-GPU simulator has been tested by using randomly generated PDP systems, and validated using the model of the Bearded Vulture ecosystem in [39,40]. Using a NVIDIA Tesla C1060 GPUs and random PDP systems, an overall speedup of 7x was obtained compared to a single CPU, and 3x against 4-core CPU using the OpenMP implementation [37,38]. Using a Tesla K40 and simulating the Bearded Vulture model, the obtained speedup was 18.1x.

3. P-Lingua version 5.0

In this work, P-Lingua syntax has been extended by including the concept of *feature* in order to pass information directly from the P-Lingua user to the simulator. A new lightweight parser has been developed in C++ 11 using the Standard Template Library (STL) instead of using the Java pLinguaCore. The new software is able to execute a fast and memory-efficient parsing of the P-Lingua files with the new syntax and produce output files in XML/JSON/binary format. The generated binary files are a serialisation of internal data structures describing the P system and can be loaded by simulators directly into memory. This version of P-Lingua is retro-compatible, so files written with versions 1 to 4 can be also used in version 5. The software can be downloaded from https://github.com/RGNC/plingua.

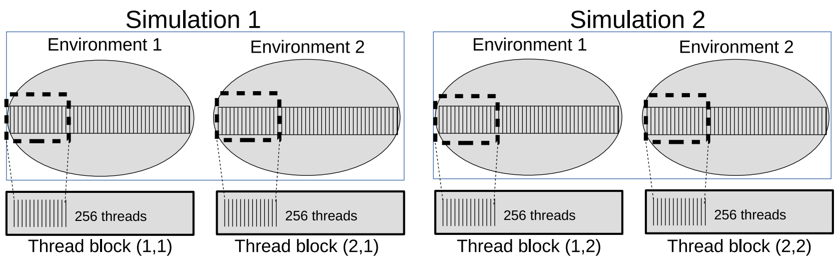


Fig. 1. Example of the design for ABCD-GPU, working over a PDP system with 2 environments and performing 2 simulations. Inside environments, rule blocks are represented by strides. Thread blocks with 256 threads are launched and loop over the rule blocks.

3.1. *Syntax of a P-Lingua file*

In general, a P-Lingua file begins with the definition of the P system model type and contains a set of functions, where each function can define rules, initial multisets or the initial membrane structure. Global and local variables can be defined as in conventional programming languages.

```
@model<probabilistic>
@include "pdp_model.pli"
N = 10; // Number of environments
M = 7 ; // Number of species
```

The lines above specify the model type to be used and two global variables. In this new version of P-Lingua, other files can be imported. A new syntax has been included in order to define variants in P-Lingua files instead of using pre-defined identifiers, in this case, the included file `pdp_model.pli` defines the semantics (e.g. derivation mode) of the `probabilistic` model (for PDP systems). The specific details about the new semantic ingredients in P-Lingua 5 are out of the scope of this work, and can be found in [41].

```
1 def main()
2 {
3  @mu = []'p;
4  @mu(p) += [[[]'1]'0]'{k},{k} :  101 <= k <= (100+N);

5  @ms(1,{j+100}) += X{k} * q{k,j}  : 2<=k<=M,1<=j<=N;
6  @ms(0,{j+100}) += X{1} * q{1,j}  : 1<=j<=N;
7  @ms(0,{j+100}) += R{0}           : 1<=j<=N;

8  /*r1*/ X{1}[]'1 --> +[X{1},G*(h{j})]'1:: m{j} : 1<=j<=3;
9  /*r2*/ [X{i}]'1 --> +[X{i}*(1+d{i})]'1:: (k{i,1}*0.5)
                   : 2<=i<=M;
10  /*r3*/ [X{i}]'1 --> +[X{i}]'1:: 1-(k{i,1}*0.5) : 2<=i<=M;

11  reproduction_rules(M,3);
12 }
```

The code above is an example of a function (with line numbers). A `main` function must be included as starting module. Functions can call other functions and, optionally, they can also return a value. For instance, there is a predefined function for random number generation. In the example, there is a call with two parameters in line 11, and the initial membrane structure has been defined in a parametric way in the first two lines (3 and 4). For Population Dynamics P systems, a virtual skin membrane `p` is used. That is, the structure is a cell-like P system, i.e, a tree structure where the membranes at the second level are the environments. Each membrane has two labels: membrane and environment identifiers. In this example, membranes used for the environments are identified by $(100+i,\, 100+i)$ with i from 1 to the number of environments N.

Lines from 5 to 7 add multisets of objects to the initial multisets defined by a particular pair of labels. Multiplicities are defined by using the multiplication symbol (`*`). Likewise, line 6 includes `q{i,j}` times the object `X{1}` in the initial multiset of membrane 0 (skin) at environment `j+100`, for `j` from 1 to `N`.

Lines from 8 to 9 are examples of rules. Each rule has a left-hand side and a right-hand side separated by an arrow symbol (`-->`). Membranes are defined by square brackets with an optional charge + or − before the left bracket (default is neutral charge) and a membrane label after the right bracket. Iterators can be used to define rules in a parametric way. Finally, probabilities related to rules are included after the symbol `::` and before the iterators' ranges. For instance, line 8 defines three rules using a parameter `j` from 1 to 3, each one having probability `m{j}`.

3.2. *Features*

In this new P-Lingua version, the user can provide additional high-level information to the simulator by using *features* (inspired from directives in common programming languages, as in OpenMP). A feature can be a numeric value or a character string associated to a unique name (character string). The syntax to define a feature is as follows:

```
@feature_name = feature_value;
```

Next, three features are defined as an example, the first one is an integer, and the rest are strings:

```
@number_of_modules = 5;
@sequence = "3, 1, 4, 2, 5";
@names = "high, low"
```

It is also possible to define a feature without value; in this case, a default value of 1 is assigned. Moreover, the user can define two types of features:

- Global features: They are written at the global scope and are related to the whole P system. In fact, the example above represents global features.
- Rule features: This type is related to only a particular rule. It is written at the rule scope after providing the rest of the syntax of the rule (i.e. right before the semicolon). For example:

  ```
  [A]'1 --> [B]'1 :: 0.9 @stage = "generation";
  ```

Features are stored inside the serialised object created by P-Lingua, and the place depends on the type (global or rule level). They can be fetched by explicit calls, meaning that simulators that do not support features will work as usual.

4. Adaptative GPU simulator

This section provides a detailed description of the adaptative PDP system simulator for GPUs. The mechanism to declare modules as features and how to handle them is first explained, and later the extension of DCBA to take advantage of modules is discussed.

4.1. Modules as features

In [9], a protocol for describing PDP system models is standardised. In stage 4 of the protocol, the algorithmic scheme captured by the model is sketched. Let us recall that a cycle corresponds to a certain time in the simulated biological system (e.g. a year) and that lasts a prefixed number of steps in the model. Cycles are repeated along the computation. Inside a cycle, the same sequence of processes will be repeated. They take predetermined time intervals, ensuring in this way that a cycle always lasts the same transition steps. These processes are known as modules, and represent specific parts of the system to be modelled. Moreover, given that modules can bifurcate to others, their execution can be made either sequentially or in parallel, but always in a synchronous way.

Moreover, defining models by modules is not unique for biological modelling. When dealing with solutions to computationally-hard problems, designers usually develop models by stages (that can be seen as a sequence of modules), what eases the design and posterior formal verification. This can be seen for example in the solution to the subset-sum problem in [42], where the stages are: generation, weight calculation, checking, and output. Similarly, a solution to SAT problem based on P systems with active membranes can be constructed over the following stages: generation, synchronisation, check-out and output [43]. These stages cover the whole computation of the P system, given that it stops after completing the last stage. In particular, this was the key when developing specific simulators for this solution [30,44], because it is easier to adapt the kernels to each stage: rules to be applied in each stage are previously known, allowing to automatically discard rules from other stages.

In this sense, an adaptative simulator has the goal of getting closer to specific simulators without losing generality; that is, they are generic simulators with improved performance by taking advantage of extra information provided directly by designers (e.g. modules). The concept of adaptative simulator is here tested by an extension of ABCD-GPU. For simplicity, the scheme of modules supported by this simulator is restricted to a directed graph without cycles. This way, the activation of modules is predefined by a *transition* graph, where modules get active for a certain period of time (measured in transition steps of the model) according to a relationship of succession: after one module, others can get active. We will say that a module is active in a certain transition step when its rules are electable for execution. Therefore, rules from inactive modules can be automatically discarded in selection stage. Let us remark that several modules can be active at a certain step, but cycles in the transition graph are not permitted (after a module, none of its predecessors can come next). In fact, the whole graph must be synchronised: modules (e.g. b and c) following a certain one (e.g. a) must be active in the next step after finishing the activation of the ancestor (a); and if a module (e.g. c) has more than one ancestor (e.g. a and b), they (a and b) must finish their activation right in the step before it (c) gets active. Similarly to the modular scheme of ecosystem models, this transition graph is defined at the level of a cycle in the simulation time. Consequently, the first modules getting active at the beginning of each cycle become at step 0. Last modules end along with the cycle at step n, where n is the duration of a cycle minus 1. In Fig. 2, an example of a transition graph is shown.

Modules can be defined in P-Lingua files in a straightforward way through features. This means that they are included alongside the model description as an extension, being easily accessible for model designers. It is noteworthy that this information is transparent for simulators not supporting modules, given that features are extra items that have to be fetched explicitly. There are two main pieces of information that has to be declared in order to define modules:

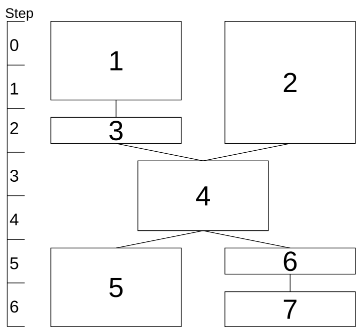


Fig. 2. Example of a transition graph formed by 8 modules, and with a cycle of 7 steps. Module 1 is active in the first two steps of the cycle and module 3 will follow by getting active in the third step. Module 2 is active simultaneously with 1 and 3 during 3 steps. After modules 2 and 3, the fourth module gets active from steps 3 to 4. After it, module 5 gets active for 2 steps, while in parallel, modules 6 and 7 get active consecutively by just one step.

1. Information about the modular structure of the model. This includes module names and their temporal relation. The latter indicates when a module starts inside a cycle and which modules will follow a given one.
2. Information about distribution of rules in modules. That is, which module each rule belongs to.

The modular structure consists of the following: module names, duration of each module measured in transition steps of the model, and successors according to the graph of temporal relation. This is declared in a global feature as follows:

@modules="($module_1$,$duration_1$,$module_{k^1_1}$,...,$module_{k^1_{z_1}}$);...;
($module_n$,$duration_n$,$module_{k^n_1}$,...,$module_{k^n_{z_n}}$)";

where n is the total amount of modules; $module_i$ is the identifier of module with index i (it can be either just the index i or a descriptive name, e.g. “feeding”, “mortality”, etc.); $duration_i$ is the number of steps that module i is active; z_i is the number of modules that get activated after the end of $module_i$; and $module_{k^i_1}, \dots, module_{k^i_{z_i}}$ are the identifiers of the modules following module i (that is, its successors list), with $0 \le z_i \le n - 1$. Therefore, for each module i, the designer has to provide a name or index of such module, its duration in steps, and the modules that come next after it finishes. Specifically, the successors list is empty when the module is the last one in the cycle or there are no modules starting after it. For example, the following declaration in P-Lingua corresponds to the graph in Fig. 2:

```
@modules= "(1,2,{3});(2,3,{4});(3,1,{4});
(4,2,{5,6});(5,2,{}); (6,1,{7});(7,1,{})";
```

Membership to each module is declared by a local feature to the corresponding rules in the following form: $@module =$ “$module_i$”. This indicates the identifier of the module (as described in the feature *@modules*) where the corresponding rule belongs to. For example, a set of rules belonging to module 1 can be defined as follows:

```
[X{i}]'1 --> +[X{i}*(1+d{i})]'1::(k{i,1}*0.5)
          @module="1": 2<=i<=M;
```

In the above example, an iterator is employed, so for each value of i (which iterates until the constant value M) a rule is generated with that pattern. All of them belong to module 1, and each one has probability $\frac{k\{i,1\}}{2}$.

All rules must belong to a module. By default, if the *@module* feature is not provided for a rule, it is assumed to be part of a default, virtual module that gets active during the whole cycle. Moreover, it is easy to demonstrate that all rules inside each consistent block belong to the same module. Finally, a further restriction is imposed: two rule blocks from two different modules getting active at the same step cannot compete for objects (as in Definition 4). That is to say, DCBA is executed locally to each module given that rule blocks from simultaneous modules are independent. Of course, rule blocks inside modules or from consecutive modules can have overlapping LHS. This is usual in modular designs and it is required in order to obtain better efficiency.

4.2. Modular DCBA

The module structure is retrieved from P-Lingua 5 data structures in the binary file, together with the model itself. This is important because the simulator has to process this information before reading the rule blocks. For example, with the goal of increasing coalesced access to data on the GPU, rules blocks are sorted first according to the module they belong to.

Four main arrays are employed to handle modules during the simulation. All of them have a size equals to the number of modules:

- *startStep*: for each module, the step since it gets active.
- *endStep*: for each module, the step since it is no longer active.
- *blockIndexSkeleton*: for each module, the index of the first skeleton rule block (i.e. of the form $B_{i,\alpha,\alpha',u,v}$) belonging to it.
- *blockIndexCommunication*: the index of the first communication rule block (i.e. of the form $B_{e_j,x}$) belonging to it.

In other words, a module i starts its activation in step *startStep*[i] inside a cycle, and ends at *endStep*[i] $-$ 1. The rule blocks from the skeleton belonging to module i are *blockIndexSkeleton*[i] to *blockIndexSkeleton*[$i+1$] (idem for communication rule blocks). When *blockIndexSkeleton*[i] $=$ *blockIndexSkeleton*[$i+1$], no skeleton rule block belongs to module i (vice versa for communication blocks). These arrays can be constructed from the transition graph, as defined in previous section, using Algorithm 2. Although it is sequential, it has to be run only once at the beginning. Moreover, the number of modules is not expected to be very high. As an example, next, the *startStep* and *endStep* vectors are defined for the graph in Fig. 2: *startStep* $= \{0, 0, 2, 3, 5, 5, 6\}$, *endStep* $= \{2, 3, 3, 5, 7, 6, 7\}$ (each position corresponds to the module index as shown in the figure).

Let us recall that Fig. 1 shows an example of the original design of ABCD-GPU. It can be seen that while environments and simulations are completely parallelised by thread blocks, rule blocks (or rules for phase 4) are distributed among the threads, which are restricted to 256 for best performance according to our experiments. The higher the amount of rule blocks, the larger is the loop inside environments. In fact, ABCD-GPU first visits every block but for only detecting its applicability. Hence, if the block is not applicable, it will be dismissed for normalisation and updating kernels. However, three downsides happen:

- Threads need to visit *activationVector* for every block, to check whether it is active.
- *activationVector* can be sparse, so its access by threads is not fully coalesced and would require more iterations to cover all of them.
- Given that *activationVector* is sparse and not compacted, access to rule block information (LHS, etc.) is also not coalesced.

Fig. 3 exemplifies the work distribution of ABCD-GPU with modules. This version increases parallelism by launching concurrent thread blocks, assigning each module to a stream. The loop over thread blocks is further decreased by only considering those inside the active modules, avoiding in this way to visit *activationVector* for all of them.

As a matter of fact, modules are windows over the rule blocks, allowing to skip those not belonging to active modules (Fig. 3). Thus, the advantage is twofold: (1) it reduces the loop over rule blocks made by threads, and (2) it compacts *activationVector* to module level, reducing sparsity. The larger a module, the sparser *activationVector*. Furthermore, DCBA is executed locally inside modules, enabling the possibility to run it in parallel over simultaneously-active modules. Specifically in ABCD-GPU, the implementation has been slightly modified:

- Kernels receive the range of rule blocks (or rules) to visit. Originally in ABCD-GPU, kernels looped over every rule block defined in the PDP system.
- For every module, a kernel is launched for a specific range of transition steps. If more than one module is active, these kernels run in parallel by assigning them to non-default CUDA streams.
- As an extra improvement, it is easy to check that communication rule blocks do not require DCBA for object distribution, because their LHS (a single object from Σ) cannot overlap with skeleton rules (see Definition 1). Therefore, two kernels, one for skeleton rule blocks and another for communication rule blocks, are launched at different CUDA streams.

However, there are some aspects to be considered. First, as mentioned in Section 2.2, if the accurate mode with n/d representation is used, non-active blocks have to subtract their contribution to *numerator* vector. That is to say, blocks from non-active modules have to be visited in any case, losing the advantage of this technique. Instead, local copies of *numerator* and *denominator* for each module are previously populated and employed when needed. Therefore, the algorithm needs to upload to the GPU only those copies for active modules by copying just the non-null values at each transition step. Given that rules from different simultaneously-active modules cannot compete for objects, they will not interfere. In contrast, for non-accurate mode, the *addition* vector can be used as previously safely. Secondly, kernels might run out of parallelism when handling small modules (with few rule blocks). Thirdly, memory for B^j_{sel} and R^j_{sel} has to be initialised at every step. In the original version of the simulator, this initialisation is done implicitly because all rule blocks and rules are visited. This way, threads write at least a 0 in their positions if the corresponding rules are not selected. Fourthly, given that DCBA is performed per module, it might be possible to reduce space in some auxiliary data structures, like B^j_{sel}, *activationVector*, etc. This was not done at the time of writing, but it is planned for future work. Lastly, the same aforementioned design concepts have been also applied to the multicore (OpenMP) version of the PDP system simulator. Here, simulations are distributed among cores, and each thread loops over the rule blocks that are only active according to the module information, as done by the GPU simulator. The same data structures are employed, so compaction also takes place.

The source code of the adaptative PDP system simulator implemented in this work is available at the PMCGPU website [25] (section “Files”, folder “ABCD-GPU”), and at the following repository https://github.com/RGNC/abcd-gpu (tag adaptative-1.0).

Algorithm 2 MODULES PROCESSING PROCEDURE

```
Require: A transition graph of modules defined as a set of tuples: $RGM = \{(module_i, duration_i, successors_i = \{module_{k_1^i}, \dots, module_{k_{z_i}^i}\})\}$, for $1 \le i \le n$,
    where $n$ is the number of modules.
 1: initialise arrays $startStep$, $endStep$, $duration$ and $predecessor$ with 0
 2: for each $(module_i, duration_i, successors_i)$ from $RGM$ do              ▷ (create aux arrays)
 3:     $duration[i] \leftarrow duration_i$
 4:     for each module $j$ from $successors_i = \{module_{k_1^i}, \dots, module_{k_{z_i}^i}\}$ do
 5:         $predecessor[j] \leftarrow i$              ▷ (assuming synchronisation, choose any predecessor)
 6:     end for
 7: end for
 8: for $i \leftarrow 1$ to $n$ do                                                ▷ (create final arrays)
 9:     $j \leftarrow i$                                                   ▷ (seeking the first predecessor)
10:     $t \leftarrow 0$                                        ▷ (accumulating duration of predecessors)
11:     while $predecessor[j] \neq 0$ do
12:         $t \leftarrow t + duration[j]$
13:         $j \leftarrow predecessor[j]$
14:     end while
15:     $startStep[i] \leftarrow t$
16:     $endStep[i] \leftarrow t + duration[i]$
17: end for
```

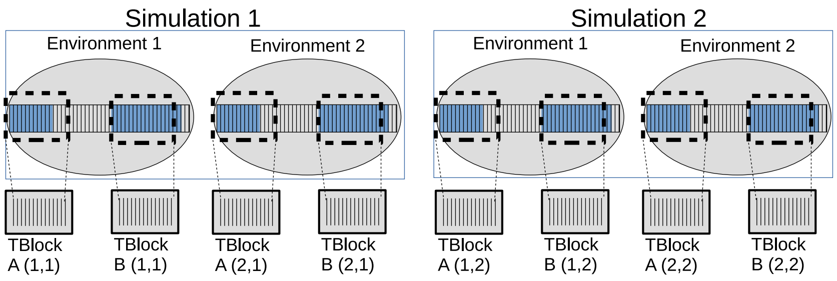


Fig. 3. Example of the design for modular ABCD-GPU, for a PDP system with 2 environments and for 2 simulations. Inside environments, rule blocks are represented. There are two active modules (in shaded). Thread blocks with 256 threads are launched in two CUDA streams, A and B. Each stream is for each module.

5. Case study

Next, the behaviour and performance of the adaptative PDP system simulators for GPU and OpenMP are analysed. The model employed as benchmark is introduced in the first subsection, and the detailed analysis is made in the second one.

5.1. Generalised tritrophic interactions model

The model employed in this work to test the performance of the simulators is based on the tritrophic interactions presented in [33,45]. This is a virtual ecosystem that was defined to illustrate PDP systems as a modelling framework. In this model, three trophic levels are represented: grass, herbivores and carnivores. These species interact with each other, reproduce and move along the 10 environments when no food is encountered. Rule block competitions take place. For instance, all herbivores compete for grass, that is represented by a single object, *G*.

For benchmarking purposes, the model has been generalised so that the number of species can be changed. The corresponding parameters (probabilities, amount of copies eaten per species, etc.) are generated randomly. This was possible thanks to the ability of P-Lingua 5 to incorporate calls from the model to random number generation functions. The P-Lingua file corresponding to this generalised model is available along with the source code of the simulator.

Fig. 4 shows the transition graph (and hence, the modules) in the model. A cycle takes 9 steps, and there are 5 modules, each one lasting 1 transition step excepting module 3, which takes 5 steps. Environment communication rules are executed only in module 3. The @*modules* feature definition is therefore as follows:

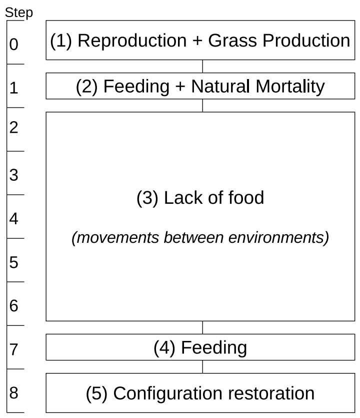

Step
0
1
2
3
4
5
6
7
8
(1) Reproduction + Grass Production
(2) Feeding + Natural Mortality
(3) Lack of food
(movements between environments)
(4) Feeding
(5) Configuration restoration

Fig. 4. Transition graph corresponding to the modules in the algorithmic scheme of tritrophic interactions model [45].

```
@modules="(1,1,{2});(2,1,{3});(3,5,{4});(4,1,{5});
          (5,1,{})";
```

There is no connection from module 5 back to 1. As mentioned in Section 4.1, the transition graph has no cycles and moreover, the graph is defined till the end of cycle. Thus, it is assumed that the next cycle will start again with module 1. Last but not least, identifiers of modules can be also strings, but we opted to provide just the indexes for simplicity.

5.2. Analysis of performance results

In this section, the benchmark carried out to the adaptative PDP systems simulator is analysed. The two versions of the simulator are compared: **original** (ABCD-GPU as in [39]) and **modular** (adaptative variant of ABCD-GPU). The extended tritrophic model is used as input. In all experiments, 20 years of the virtual ecosystem are simulated (corresponding to 180 transition steps of the PDP systems). The *A* parameter of DCBA is set to 2, and the static approach for phase 2 was chosen. No output was asked, so only the simulation runtimes were measured. The scalability of the simulators is analysed by increasing the number of species. Specifically, 7 will be used to denote the *base* model, which has in fact 7 species. In order to have an idea of the dimensions of the model, the ratio of rule blocks per species is approximately 22: 21 985 rule blocks are generated for 1000 species, being 9990 communication rule blocks and 11 995 skeleton rule blocks. Another parameter affecting scalability is the amount of simulations running in parallel. For this reason, 50 and 100 simulations were launched for the tests. Accurate, n/d, representation of additions in normalisation for phase 1 was only employed for the original model. In larger models, the non accurate mode is used. The following two configurations of CPU and GPU hardware were used to run the simulations (short names are provided in bold):

- **(i7)** Intel i7-8700 CPU at 3.20 GHz, having 12 logical cores (6 physical), 8 GB of DDR4 system memory. g++ version 7.4 was used.
- **(Xeon)** Intel Xeon CPU E3-1230 v3 at 3.30 GHz, having 8 logical cores (4 physical), 32 GB of DDR3 system memory. g++ version 4.9 was used.
- **(P100)** Tesla P100 GPU, having 3584 cores at 1.33 GHz, 16 GB of device memory with 4096-bit memory bus width at 715 Mhz. Compute Capability 6.0. CUDA 10.0 was used.
- **(GTX1050)** GeForce GTX 1050Ti GPU, having 768 cores at 1.42 GHz, 4 GB of device memory with 128-bit memory bus width at 3.5 GHz. Compute Capability 6.1. Plugged to the CPU i7. CUDA 10.0 was used.
- **(K40)** Tesla K40c GPU, having 2880 cores at 0.75 GHz, 12 GB of device memory with 385-bit memory bus width at 3 GHz. Compute Capability 3.5. Plugged to the CPU Xeon. CUDA 9.0 was used.

Let us start by analysing the impact of the *modular* scheme to the multicore implementation. As mentioned in Section 2.2, OpenMP is employed to distribute simulations among threads. The i7 CPU is tested with 4 (i7-omp4) and 8 (i7-omp8) threads, while the Xeon runs only 4 (Xeon-omp4) threads. The achieved speedups by using *modular* against *original* version are displayed in Fig. 5, where 100 simulations were run (when using 50, the numbers are similar). It can be seen that for all tested cases, the *modular* version outperforms the *original* one by up to 2.7x in Xeon CPU for 1000 species. *modular* simulator has a better impact in Xeon than in i7, where the accelerations obtained are lower (up to 2.3x). Moreover, lower improvements, between 1.5 and 1.9x, are found in a small model like the base one, while the accelerations seem to increase slightly with larger models. Furthermore, experiments with large models using 8 threads and modular version show a bit faster simulation when compared to using 4 threads.

Fig. 6 shows the speedups obtained by the simulators when running on the GPU. Similarly to the multicore version, the adaptative (*modular*) simulator outperforms the *original* one, excepting for the case of simulating the base model on GTX1050 and P100. Here, the introduced overload downgrades slightly the performance, as it will be seen in the profiling Table 3. The accelerations are much higher on the K40 GPU (from an older generation), reaching 2.5x for 500 species and 50 simulations. On the GTX1050, up to 1.7x is achieved with 1000 species and 50 simulations. On the P100, up to 1.8x is achieved with 300 species and 50 simulations. For 5000 species, the GTX1050 was not able to run 100 simulations due to memory constraints. For the three tested GPUs, the accelerations tend to be constant with larger models. Runtimes for P100 are much lower than in K40 and GTX1050 due to its higher capacity and memory bandwidth. For example, *modular* version of the simulator running 100 simulations for 1000 species took 736.37 ms in the P100, but 2224.39 ms and 2885.41 ms in K40 and GTX1050.

A cross comparison of runtimes and speedups achieved by GPU compared to CPU is shown in Fig. 7. Fig. 7(a) shows the runtimes in the fastest GPU tested (P100) and in the fastest CPU configuration (i7 with 8 OpenMP threads). Fig. 7(b) corresponds to the speedups reached by the above simulation times. The GPU is faster, in both *modular* and *original* versions, than the multicore counterparts when handling middle and large models. Only for the small base model, the GPU is a bit slower (above 0.9x). Speedups are higher with larger models, being around 30x and 50x for *modular* and *original* simulators, respectively, and for 2000 species. When simulating hundreds of species, 6x and 10x accelerations were obtained for *modular* and *original* versions. Finally, the speedup of the GPU is lower when using the *modular* version, given that the impact of the *modular* scheme is better for the CPU than for the GPU.

Lastly, let analyse the impact of the adaptative simulator in each kernel launched by the GPU. Tables 1–3 summarise the profiling performed on the simulator (numbers were obtained with *nvprof* tool provided in the CUDA toolkit). Kernels for *Filters*, *Normalisation* and *Update* belong to phase 1 (*modular* version includes a fourth kernel named *reset addition*, which explicitly set to 0 the row sums for each iteration of *A* (accuracy) parameter in DCBA). Both phases 2 and 3 are performed by single kernels. Phase 4 (execution) is divided in two kernels, one executing rules from skeleton and another for environment communication rules. Last two rows show the total time corresponding to the part for DCBA and the initialisation overhead required by the GPU (which is the same for both *modular* and *original* versions). The overall percentage of time consumed by each kernel (and phase) is shown in the column Overall, while the total runtime (for the total of 180 simulated steps) is in the column named Time (using milliseconds). Last column contains the speedups for each kernel and phases achieved by the *modular* against original version.

Tables 1 and 2 correspond to the simulation of the model with 1000 species and running 50 simulations on K40 and P100, respectively. It is easy to see that the larger impact of using *modular* version takes place in phase 2. This sequential phase is highly accelerated (13.8x on K40, 7.8x on P100) by reducing the loops over rule blocks taking into account the modules. Phases 3 and 4 are also well accelerated (around 4x and 2x), but phase 1 is barely improved. Although kernels for filters and normalisation are twice faster in the *modular* simulator, the update kernel is slower (double runtime in P100). Furthermore, kernel for executing environment communication rules is much slower on *modular* version, given the low degree of parallelism. The cause of the negative effects in these two kernels could not be determined yet, but the source of it is the decrease of parallelism (fewer rule blocks have to be visited). In all cases, phase 1 is the bottleneck in the simulation process, taking the majority of the time (from 30/50%, in *original*, to 70%, in *modular*).

Finally, the worst case where the *modular* simulator is slower than the *original* when using the base model is analysed in Table 3. The advantage of using the GPU to handle as many rule blocks as possible is restricted by using the *modular* version. In such case, very few threads will perform effective work, so

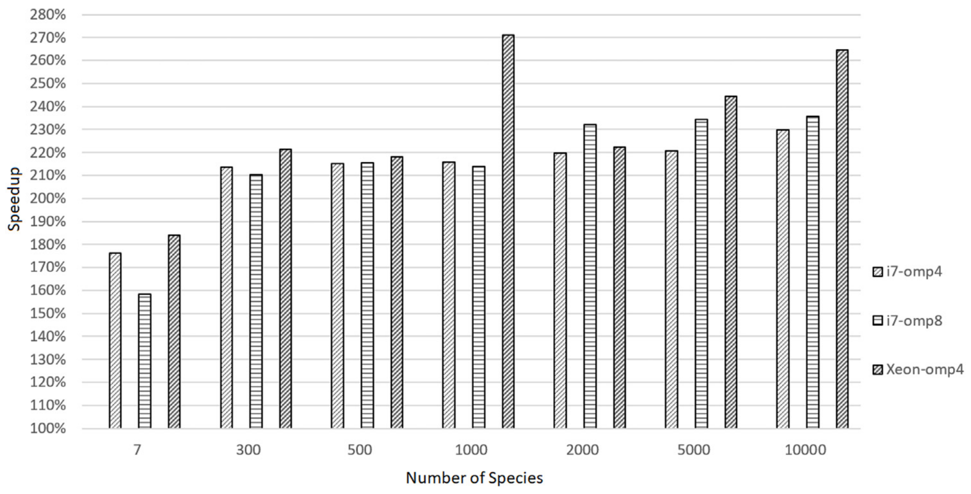


Fig. 5. Speedup achieved by using the adaptative simulator compared to the *original* version, for three configurations of multicore processors. The number of species are increased up to 10 000, and 100 simulations were run.

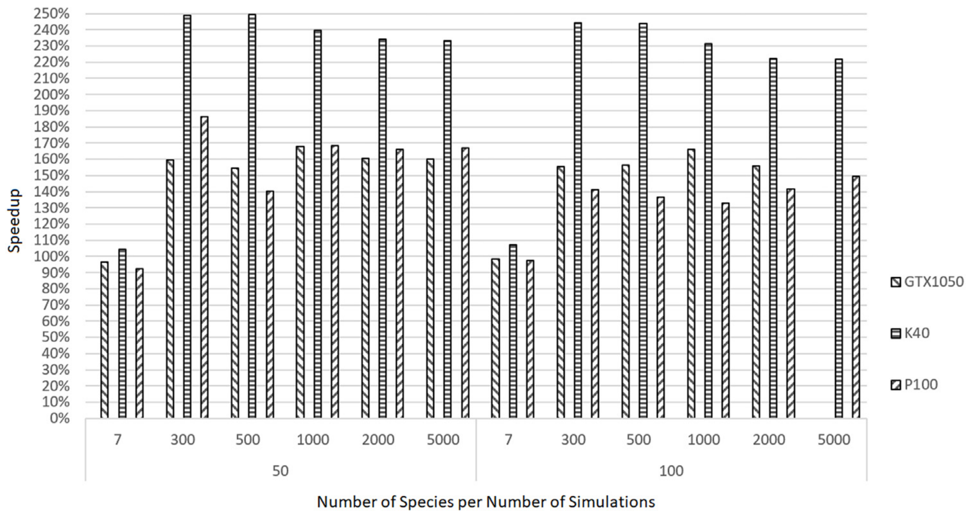


Fig. 6. Speedup achieved by using the adaptative simulator compared to the *original* version, for three GPUs: Tesla K40, GTX1050Ti and Tesla P100. The number of species are increased up to 5000. 50 and 100 simulations were run.

Table 1
Modular against *original* simulator in K40 GPU, simulating the model with 1000 species and running 50 simulations.

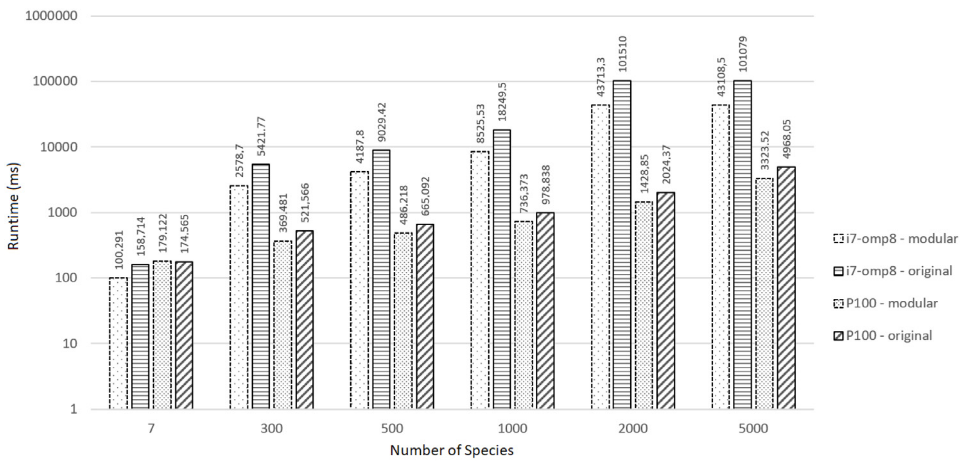
	Original		Modular		Speedup
	Overall	Time	Overall	Time	
Filters	32%	456.32 ms	31%	194.08 ms	2.35x
Normalisation	13%	179.49 ms	10%	64.40 ms	2.79x
Update	12%	168.37 ms	31%	189.42 ms	0.89x
Reset addition			2%	14.59 ms	
Phase 1	57%	804.18 ms	75%	462.50 ms	1.74x
Phase 2	22%	305.60 ms	4%	22.13 ms	13.81x
Phase 3	17%	237.27 ms	15%	90.15 ms	2.63x
Exec skel	2%	27.90 ms	1%	6.47 ms	4.31x
Exec env	0%	1.53 ms	1%	7.02 ms	0.22x
Phase 4	2%	29.43 ms	2%	13.49 ms	2.18x
Total DCBA	**98%**	**1376.48 ms**	**95%**	**588.27 ms**	**2.34x**
Init overhead	2%		5%		

Table 2
Modular against *original* simulator in P100 GPU, simulating the model with 1000 species and running 50 simulations.

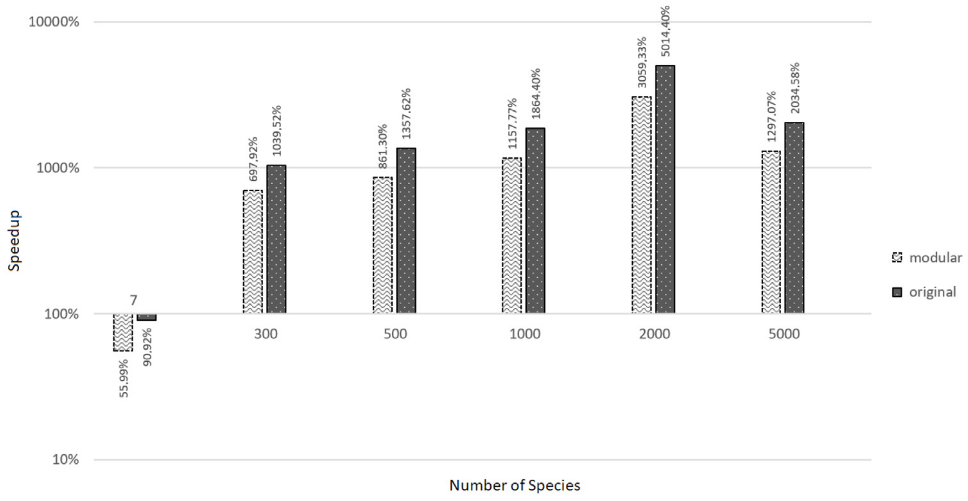
	Original		Modular		Speedup
	Overall	Time	Overall	Time	
Filters	14%	78.25 ms	9%	38.01 ms	2.06x
Normalisation	23%	129.69 ms	21%	94.36 ms	1.37x
Update	20%	112.93 ms	44%	195.07 ms	0.58x
Reset addition			5%	20.44 ms	
Phase 1	56%	320.87 ms	79%	347.87 ms	0.92x
Phase 2	14%	79.36 ms	2%	10.14 ms	7.83x
Phase 3	24%	135.97 ms	14%	62.67 ms	2.17x
Exec skel	3%	21.65 ms	1%	5.75 ms	3.76x
Exec env	0%	0.67 ms	1%	4.39 ms	0.15x
Phase 4	3%	22.32 ms	2%	10.15 ms	2.20x
Total DCBA	**97%**	**558.52 ms**	**97%**	**430.83 ms**	**1.30x**
Init overhead	3%		3%		

the performance is negatively affected. Indeed, only kernels for normalisation, phase 2, and execution of skeleton are accelerated. Here again, phase 2 runs twice faster on the *modular* version, demonstrating the positive effect in this serial kernel. The slowest kernel is again *Update from phase 1*. In this case, the overhead for initialisation is high, consuming half of the time required for simulation.

In light of the results here analysed, it can be concluded that using the adaptative simulator in small models only pays off when using the CPU. On the contrary, *modular* version is



(a)



(b)

Fig. 7. Comparison of P100 versus i7 with 8 threads. Both *original* and *modular* simulators are tested for different number of species in the model. (a) shows the runtimes in ms for P100 and i7 executing *original* and *modular* versions, while (b) shows the corresponding speedups of P100 against i7 for both version. 50 simulations were run. Bar plots use logarithmic scale for *y*-axis.

Table 3
Modular against *original* simulator in GTX1050 GPU, simulating the base model with 7 species and running 100 simulations.

	Original		Modular		Speedup
	Overall	Time	Overall	Time	
Filters	6%	11.87 ms	8%	15.05 ms	0.79x
Normalisation	13%	23.28 ms	11%	21.19 ms	1.10x
Update	8%	14.83 ms	11%	21.43 ms	0.69x
Reset addition			3%	4.86 ms	
Phase 1	27%	49.99 ms	33%	62.54 ms	0.80x
Phase 2	7%	12.22 ms	3%	5.05 ms	2.42x
Phase 3	10%	17.99 ms	11%	20.80 ms	0.87x
Exec skel	2%	4.08 ms	2%	3.34 ms	1.22x
Exec env	1%	2.55 ms	1%	1.42 ms	1.80x
Phase 4	4%	6.63 ms	3%	4.77 ms	1.39x
Total DCBA	**47%**	**86.85 ms**	**49%**	**93.16 ms**	**0.93x**
Init overhead	53%		51%		

always better in large models for both GPU and CPU. Moreover, the adaptative simulator has better acceleration effects in older generations of GPUs (K40) and CPUs (Xeon). Finally, the GPU turns out to be faster than the CPU counterparts when used for medium and large models. However, *modular* version for tritrophic model reduces the amount of parallelism (fewer blocks have to be visited), so the benefit of using the GPU is lower.

6. Conclusions and future work

Generic P system simulators are designed to assume that worst situations might arise, such as applying all the defined rules at a given step. This flexibility usually comes with a performance cost. In contrast, specific simulators are designed for a target model, restricted to just one family of P systems. In this paper, adaptative simulators are explored as a middle ground. The model designer is asked to provide high-level information, in form of so-called features, to the simulator in a straightforward and transparent way (similarly to directives in programming languages). For this purpose, a new P-Lingua version supporting these features has been developed.

As a case study, ABCD-GPU has been extended. It is a simulator for PDP systems aimed at ecosystem modelling. This extension enables the simulator to handle modules, typically defined in PDP system models, so that rules are grouped and then sequentially executed. In this way, the internal loops over the rules are significantly reduced, showing speedups of up to 2.5x in

GPU implementations, and 2.7x in CPU counterparts. Adaptative simulators can be constructed easily from previous one by implementing extensions. This comes with slight a overhead, while reducing the parallelism in GPU simulators, so paying off when simulating large models.

The concept of features offers new opportunities to enhance the performance of P system simulators. These have been integrated into P-Lingua in a transparent manner, such that existing simulators remain unaffected but can be extended to support them. We believe that this will open new research lines, some of them discussed next.

In this work, we did not pay attention to the memory footprint. The memory employed by the adaptative simulator increases slightly (specially when using the n/d representation, where separated arrays are populated per module). However, the largest data structures (like B^j_{sel}, R^j_{sel}, *activations* and *additions*) can be highly decreased by taking into account the modules, considering that DCBA is applied locally to each module. In order to do so, translating global to local identifiers for blocks and objects would help.

Modules are disjoint subsets of the set of rules of the P system. The rules of each module are supposed to be executed in parallel. The idea of module is not unique for PDP systems. Other P system models, like multicompartmental P systems (for systems biology modelling) and solutions to computationally-hard problems (where the computation is defined by stages) also define modules as stages of the computation. This work has explored expressing modules as features that divide the rule traversal into stages. We note other possible implementations as a natural extension to this work: (1) objects acting as counters make data structures very sparse [29], so a feature can help to identify them and use an efficient representation; (2) competitions of rules happen locally inside each module, and a feature specifying where can localise even more DCBA; (3) features can have the ability of disabling environments and even membranes at certain steps, avoiding wasting resources; (4) features can be also used to improve the accuracy rather than just the efficiency, like some specific simulation parameter (e.g. the accuracy, A, of DCBA), or even different simulation algorithms (BBB, DNDP) can be selected per rule and step. Concerning the modules, an improved transition graph where conditional transitions between modules (e.g. if some objects appear in the configuration) are allowed can be explored.

Finally, the roadmap for PDP system GPU simulators was discussed in [39]. The idea of using features to help the simulator was first introduced in item 8 of this roadmap. There are more ideas for improving efficiency to be explored. For instance, item 6 corresponds to micro-DCBA, where competitions are automatically detected to perform DCBA locally. Although it shares similarities with the goal of this work, the main difference is that it does not require extra information from the model designer, but it depends only on the rules defined. One downside of defining modules as features is that unexpected results can be obtained from the simulation when this information is wrongly provided, specially if the model has not been validated nor debugged by the designer yet. Thus, features are recommended in deployment environments (i.e. virtual experimentation and parameter calibration), while micro-DCBA is intended for the validation processes of the model.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the research project TIN2017-89842-P (MABICAP), co-financed by *Ministerio de Economía, Industria y Competitividad (MINECO)* of Spain, through the *Agencia Estatal de Investigación (AEI)*, and by *Fondo Europeo de Desarrollo Regional (FEDER)* of the European Union. The authors also acknowledge the donation of a Tesla K40 by the *NVIDIA CUDA Research Center* programme in 2014. Moreover, the authors also thank D. Campora for his comments on the manuscript.

References

[1] Gh. Păun, Computing with membranes, J. Comput. System Sci. 61 (1) (2000) 108–143, http://dx.doi.org/10.1006/jcss.1999.1693.
[2] M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera, D. Orellana-Martín, Results on computational complexity in bio-inspired computing, in: Bio-Inspired Computing Models and Algorithms, (Ch. 2), pp. 33–73, http://dx.doi.org/10.1142/9789813143180_0002.
[3] D. Orellana-Martín, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, A path to computational efficiency through membrane computing, Theoret. Comput. Sci. 777 (2019) 443–453, http://dx.doi.org/10.1016/j.tcs.2018.12.024, In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I.
[4] A. Leporati, L. Manzoni, G. Mauri, A.E. Porreca, C. Zandron, A survey on space complexity of P systems with active membranes, Int. J. Adv. Eng. Sci. Appl. Math. 10 (3) (2018) 221–229, http://dx.doi.org/10.1007/s12572-018-0227-8.
[5] Gh. Păun, F.J. Romero-Campero, Membrane computing as a modeling framework. Cellular systems case studies, in: M. Bernardo, P. Degano, G. Zavattaro (Eds.), Formal Methods for Computational Systems Biology, in: Lecture Notes in Computer Science, vol. 5016, Springer Berlin Heidelberg, 2008, pp. 168–214, http://dx.doi.org/10.1007/978-3-540-68894-5_6.
[6] M. Gheorghe, N. Krasnogor, M. Camara, P systems applications to systems biology, Biosystems 91 (3) (2008) 435–437, http://dx.doi.org/10.1016/j.biosystems.2007.07.002, P-Systems Applications to Systems Biology.
[7] F. Bernardini, M. Gheorghe, N. Krasnogor, Quorum sensing P systems, Theoret. Comput. Sci. 371 (1) (2007) 20–33, http://dx.doi.org/10.1016/j.tcs.2006.10.012, Computing and the Natural Sciences.
[8] R. Barbuti, P. Bove, P. Milazzo, G. Pardini, Minimal probabilistic P systems for modelling ecological systems, Theoret. Comput. Sci. 608 (2015) 36–56, http://dx.doi.org/10.1016/j.tcs.2015.07.035, From Computer Science to Biology and Back.
[9] M.A. Colomer, A. Margalida, M.J. Pérez-Jiménez, Population dynamics P system (PDP) models: a standardized protocol for describing and applying novel bio-inspired computing tools, PLoS One 8 (5) (2013) e60698, http://dx.doi.org/10.1371/journal.pone.0060698.
[10] M. García-Quismondo, J.M. Reed, F.S. Chew, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, Evolutionary response of a native butterfly to concurrent plant invasions: Simulation of population dynamics, Ecol. Model. 360 (2017) 410–424, http://dx.doi.org/10.1016/j.ecolmodel.2017.06.030.
[11] D. Díaz-Pernil, C. Graciani-Díaz, M.A. Gutiérrez-Naranjo, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Software for P systems, in: Gh. Păun, G. Rozenberg, A. Salomaa (Eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, Oxford (U.K.), 2010, pp. 437–454, URL https://global.oup.com/academic/product/the-oxford-handbook-of-membrane-computing-9780199556670.
[12] J. Blakes, J. Twycross, F.J. Romero-Campero, N. Krasnogor, The Infobiotics Workbench: an integrated in silico modelling platform for Systems and Synthetic Biology, Bioinformatics 27 (23) (2011) 3323–3324, http://dx.doi.org/10.1093/bioinformatics/btr571.
[13] M.A. Colomer, A. Margalida, D. Sanuy, M.J. Pérez-Jiménez, A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study, Ecol. Model. 222 (1) (2011) 33–47, http://dx.doi.org/10.1016/j.ecolmodel.2010.09.012.
[14] J.L. Lérida, A. Agraz, F. Solsona, M.À. Colomer, PSysCal: a parallel tool for calibration of ecosystem models, Cluster Comput. 17 (2) (2014) 271–279, http://dx.doi.org/10.1007/s10586-013-0310-7.
[15] M.A. Colomer, A. Margalida, L. Valencia, A. Palau, Application of a computational model for complex fluvial ecosystems: The population dynamics of zebra mussel Dreissena polymorpha as a case study, Ecol. Complex. 20 (2014) 116–126, http://dx.doi.org/10.1016/j.ecocom.2014.09.006.

[16] L. Valencia-Cabrera, D. Orellana-Martín, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, An interactive timeline of simulators in membrane computing, J. Membr. Comput. 1 (3) (2019) 209–222, http://dx.doi.org/10.1007/s41965-019-00016-z.

[17] M. García-Quismondo, R. Gutiérrez-Escudero, M.A. Martínez-del-Amor, E. Orejuela-Pinedo, I. Pérez-Hurtado, P-Lingua 2.0: a software framework for cell-like P systems, Int. J. Comput. Commun. Control 4 (3) (2009) 234–243, http://dx.doi.org/10.15837/ijccc.2009.3.2431.

[18] The P-Lingua web page, 2019, http://www.p-lingua.org, (Last Accessed June 2019).

[19] M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Simulating P systems on GPU devices: a survey, Fund. Inform. 136 (3) (2015) 269–284, http://dx.doi.org/10.3233/FI-2015-1157.

[20] 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, Brief. Bioinform. 11 (3) (2010) 313–322, http://dx.doi.org/10.1093/bib/bbp064.

[21] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, M. Ujaldón, The GPU on the simulation of cellular computing models, Soft Comput. 16 (2) (2012) 231–246, http://dx.doi.org/10.1007/s00500-011-0716-1.

[22] A. Maroosi, R.C. Muniyandi, E. Sundararajan, A.M. Zin, Parallel and distributed computing models on a graphics processing unit to accelerate simulation of membrane systems, Simul. Model. Pract. Theory 47 (2014) 60–78, http://dx.doi.org/10.1016/j.simpat.2014.05.005.

[23] N. Elkhani, R.C. Muniyandi, G. Zhang, Multi-objective binary PSO with kernel P system on GPU, Int. J. Comput. Commun. Control 13 (2018) 323–336, http://dx.doi.org/10.15837/ijccc.2018.3.3282.

[24] J.P. Carandang, F.G. Cabarle, H.N. Adorna, N. Hope S. Hernandez, M.A. Martínez-del-Amor, Handling non-determinism in Spiking Neural P systems: Algorithms and simulations, Fund. Inform. 164 (2019) 139–155, http://dx.doi.org/10.3233/FI-2019-1759.

[25] The PMCGPU (Parallel simulators for Membrane Computing on the GPU) project website, 2019, http://sourceforge.net/p/pmcgpu, (Last Accessed June 2019).

[26] A. Ciobanu, F. Ipate, Implementation of P systems by using big data technologies, in: A. Alhazov, S. Cojocaru, M. Gheorghe, Y. Rogozhin, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 117–137, http://dx.doi.org/10.1007/978-3-642-54239-8_10.

[27] M.A. Martínez-del-Amor, I. Karlin, R.E. Jensen, M.J. Pérez-Jiménez, A.C. Elster, Parallel simulation of probabilistic P systems on multicore platforms, in: Proceedings of the Tenth Brainstorming Week on Membrane Computing, Vol. II, Fénix Editora, Seville, Spain, 2012, pp. 17–26, URL http://www.gcn.us.es/10BWMC/10BWMCvolII/papers/parallel-dcba.pdf.

[28] J. Quiros, S. Verlan, J. Viejo, A. Millan, M.J. Bellido, Fast hardware implementations of static P systems, Comput. Inform. 35 (3) (2016) 687–718, http://dx.doi.org/10.1007/978-3-642-36751-9_27.

[29] M.Á. Martínez-del Amor, D. Orellana-Martín, I. Pérez-Hurtado, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Design of specific P systems simulators on GPUs, in: T. Hinze, G. Rozenberg, A. Salomaa, C. Zandron (Eds.), Membrane Computing, in: Lecture Notes in Computer Science, vol. 11399, Springer International Publishing, 2019, pp. 202–207, http://dx.doi.org/10.1007/978-3-030-12797-8_14.

[30] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulating a P system based efficient solution to SAT by using GPUs, J. Log. Algebr. Program. 79 (6) (2010) 317–325, http://dx.doi.org/10.1016/j.jlap.2010.03.008.

[31] M.A. Colomer-Cugat, M. García-Quismondo, L.F. Macías-Ramos, M.A. Martínez-del Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera, Membrane system-based models for specifying dynamical population systems, in: P. Frisco, M. Gheorghe, M.J. Pérez-Jiménez (Eds.), Applications of Membrane Computing in Systems and Synthetic Biology, Springer International Publishing, Cham, 2014, pp. 97–132, http://dx.doi.org/10.1007/978-3-319-03191-0_4.

[32] M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M.A. Colomer, M.J. Pérez-Jiménez, DCBA: Simulating population dynamics P systems with proportional object distribution, in: Membrane Computing, Lecture Notes in Computer Science, vol. 7762, Budapest, Hungary, 2012, pp. 291–310, http://dx.doi.org/10.1007/978-3-642-36751-9_18.

[33] M.A. Colomer, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem, Nat. Comput. 11 (3) (2012) 369–379, http://dx.doi.org/10.1007/s11047-011-9289-2.

[34] M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, M.A. Colomer, A new simulation algorithm for multienvironment probabilistic P systems, in: IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications, Vol. 1, BIC-TA 2010, 2010, pp. 59–68, http://dx.doi.org/10.1109/BICTA.2010.5645352.

[35] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors: a Hands-On Approach, third ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016, URL https://www.sciencedirect.com/science/book/9780128119860.

[36] NVIDIA CUDA C Programming guide, 2019, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, (Last Accessed June 2019).

[37] M.A. Martínez-del-Amor, Accelerating Membrane Systems Simulators using High Performance Computing with GPU (Ph.D. thesis), Universidad de Sevilla, 2013, URL http://hdl.handle.net/11441/15644.

[38] M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez, Population dynamics P systems on CUDA, in: D. Gilbert, M. Heiner (Eds.), Computational Methods in Systems Biology, in: Lecture Notes in Computer Science, vol. 7605, Springer Berlin Heidelberg, 2012, pp. 247–266, http://dx.doi.org/10.1007/978-3-642-33636-2_15.

[39] M.A. Martínez-del-Amor, L.F. Macías-Ramos, L. Valencia-Cabrera, M.J. Pérez-Jiménez, Parallel simulation of Population Dynamics P systems: updates and roadmap, Nat. Comput. 15 (4) (2015) 565–573, http://dx.doi.org/10.1007/s11047-016-9566-1.

[40] M. Cardona, M.A. Colomer, M.J. Pérez-Jiménez, D. Sanuy, A. Margalida, Modeling ecosystems using P systems: the bearded vulture, a case study, in: D. Corne, P. Frisco, Gh. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, in: Lecture Notes in Computer Science, vol. 5391, Springer Berlin Heidelberg, 2009, pp. 137–156, http://dx.doi.org/10.1007/978-3-540-95885-7_11.

[41] I. Pérez-Hurtado, D. Orellana-Martín, G. Zhang, M.J. Pérez-Jiménez, P-Lingua in two steps: flexibility and efficiency, J. Membr. Comput. 1 (2) (2019) 93–102, http://dx.doi.org/10.1007/s41965-019-00014-1.

[42] M.J. Pérez-Jiménez, A. Riscos-Núñez, Solving the Subset-Sum problem by P systems with active membranes, New Gener. Comput. 23 (4) (2005) 339–356, http://dx.doi.org/10.1007/BF03037637.

[43] M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini, Complexity classes in models of cellular computing with membranes, Nat. Comput. 2 (3) (2003) 265–285, http://dx.doi.org/10.1023/A:1025449224520.

[44] M.A. Martínez-del-Amor, J. Pérez-Carrasco, M.J. Pérez-Jiménez, Characterizing the parallel simulation of P systems on the GPU, Int. J. Unconv. Comput. 9 (5–6) (2013) 405–424, URL https://www.oldcitypublishing.com/journals/ijuc-home/ijuc-issue-contents/ijuc-volume-9-number-5-6-2013/.

[45] M.A. Colomer, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem, in: IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications, Vol. 1, BIC-TA 2010, 2010, pp. 1621–1628, http://dx.doi.org/10.1109/BICTA.2010.5645258.