Parallel Simulation of PDP Systems: Updates and Roadmap

Miguel Ángel Martinez-del-Amor[1], Luis Felipe Macías-Ramos[2], Mario J. Pérez–Jiménez[2]

[1] Moving Picture Technologies, Fraunhofer IIS
Am Wolfsmantel 33, 91058 Erlangen, Germany
[2] Research Group on Natural Computing, Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mails: mdelamor@us.es, lfmaciasr@us.es, marper@us.es

Summary. PDP systems are a type of multienvironment P systems, which serve as a formal modeling framework for Population Dynamics. The accurate simulation of these probabilistic models entails large run times. Hence, parallel platforms such as GPUs has been employed to speedup the simulation. In 2012 [14], the first GPU simulator of PDP systems was presented. In this paper, we present current updates made on this simulator, and future developments to consider.

1 Introduction

P systems[16, 17] have become good candidates for computational modeling thanks to the compartmental and discrete features, both in Systems Biology [19, 20] and Population Dynamics [3]. In this concern, it is worth to mention the achieved success in real ecosystem modeling through probabilistic P systems, such as the Bearded Vulture in the Catalan Pyrenees (endangered species) [2], and the zebra mussel in Ribarroja reservoir (exotic invasive species) [1]. These works have lead to a formal, computational modeling framework called Population Dynamics P systems (PDP systems) [4].

In order to experimentally validate these P systems based models, the development of simulators is requested [17]. P-Lingua [5, 25] is a simulation framework for P systems, which aims to be generic, multi-platform (it is written in Java) and to provide a standard description language for P systems. It has been used to develop simulators for many variants of P systems, specially for PDP systems. Furthermore, experts and model designers are able to run virtual experiments in an abstracted way (without the need of accessing to details of P systems) through a special software called MeCoSim [18, 24]. MeCoSim uses P-Lingua as the simulation core.

The run times offered by these general simulation frameworks are high for some scenarios involving large and complex models. This lack of efficiency is mainly given from the facts of both using Java Virtual Machine and implementing sequential algorithms [10]. Indeed, simulating massively parallel devices like P systems in a sequential fashion is twice inefficient. This issue is can be addressed by harnessing the highly parallel architecture within modern processors to map the massively parallelism of P systems [10, 11].

Whereas commodity CPUs can contain dozens of processors, current graphic processors (GPUs) [8, 15] provide thousands of computing cores. They can be programmed using general-purpose frameworks such as CUDA [9, 23], OpenCL and OpenAcc. GPUs exploit data parallelism by using a very fast memory and simplistic cores. Given the high level of parallelism within modern GPUs (up to 3500 cores per device [23]), they have provided a platform to implement real parallelism of P systems in a natural way. Many P system models have been considered to be simulated with CUDA [11]: P systems with active membranes, SAT solutions with families of P systems with active membranes and of tissue P systems with cell division, Enzymatic Numerical P systems, Spiking Neural P systems without delays, and Population Dynamics P systems [14], among others. Most of these simulators are within the scope of PMCGPU (Parallel simulators for Membrane Computing on the GPU) software project [26], which aims to gather efforts on parallelizing P system simulators with GPU computing.

As shown by all of these research works, the development of a new P system simulator requires a big research and development effort. For example, in the case of the simulator for PDP systems, the simulation algorithm called DCBA [13, 3] was implemented. It is based on 4 different phases with completely different characteristics, and the parallelization effort is also different in each one (e.g. second phase of DCBA is a random sequential loop that cannot be easily parallelized). Therefore, the different semantical and syntactical elements of each P system variant lead to completely different GPU-based simulators. Not only does the GPU code depend on the simulated variant, but its efficiency also depend on the simulated P system within the variant [14].

In this paper, we show new developments on the GPU simulator for PDP systems. In summary, a new input module received binary files has been created, allowing to run real ecosystem models defined with P-Lingua. Moreover, we present a road map proposal, a set of research lines for future work that is going to be addressed.

The paper is structured as follows: Section 2 provides an overview of the required concepts to understand this paper. Section 3 presents the new feature of the simulator consisting in a input module to read binary files, and also some preliminary results. Finally, Section 4 discuses future developments to take into consideration.

2 Preliminaries

In this section we briefly provide the minimum concepts for the understandability of the paper. We will not introduce the model of PDP systems and GPU computing into detail. Instead, we provide short descriptions along with useful references.

2.1 PDP systems model and simulation: DCBA

PDP systems [4, 3] are a branch of multienvironment P systems [6], which consists in a directed graph whose nodes are called environments. Each environment contains a single cell-like P system. Moreover, the arcs of the graph is implicitly given by a set of communication rules which allow the movement of objects between environments in a one-to-many fashion. Thus, these rules are of the form: $(x)_{e_j} \xrightarrow{pr} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$. All the P systems within a PDP system have the same skeleton: the same membrane structure (with three polarizations), the same working alphabet, and the same set of (skeleton) evolution rules. These rules are of the form: $u\ [\ v\ ]_h^\alpha \to u'\ [\ v'\ ]_h^\beta$. It can be seen that these P systems are an extension of the active membranes model. However, no dissolution neither division are allowed, and special care on the consistency of rules has to be taken.

PDP systems have also a probabilistic flavor in terms of probabilities associated to the rules. On the one hand, a probability is associated to each skeleton rule for each environment, thus being of the following form: $u\ [\ v\ ]_h^\alpha \xrightarrow{f_{r,j}} u'\ [\ v'\ ]_h^\beta$. On the other hand, a probability is associated to each communication rule globally to the PDP system. Rules are executed in a maximal parallel way according to the probabilities. Rules having the same left-hand side must satisfy the following condition: the sum of their probabilities has to be 1. Eventually, rules having an "unique" left-hand side have associated the probability 1. Inherently to the model is the concept of rule block: a block is formed by rules having the same left-hand side.

For the syntax of the models, refer to [3, 4] and [6]. Concerning the semantics of the model, several simulation algorithms have been proposed since the introduction of PDP systems. Each new 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 from different blocks (note that rules within a block have a the same left-hand side, and the objects are consumed according to the probabilities) [10].

The latest introduced algorithm for PDP system is called *Direct distribution based on Consistent Blocks Algorithm (DCBA)* [13]. The approach taken in it is based on the idea of distributing the objects along the rule blocks in a proportional way. After this distribution, the rules within the blocks are selected according to their probabilities using a multinomial distribution. In summary, DCBA consists in 4 phases: 3 for selecting rules and the last one for performing the execution. The scheme of DCBA is the following:

1. Initialization of the algorithm: *static distribution table* (**columns:** blocks, **Rows:** (objects,membrane))
2. **Loop over Time**
3. **Selection** stage:
4. **Phase 1** (Distribution of objects along rule blocks)
5. **Phase 2** (Maximality selection of rule blocks)
6. **Phase 3** (Probabilistic distribution, blocks to rules)
7. **Execution** stage

The proportional distribution of objects along the blocks is carried out through a table which implements the relations between blocks (columns) and objects in membranes (rows). We always start with a static (general) table, and depending on the current configuration of the PDP system, the table is dynamically modified by deleting columns related to non-applicable blocks. Note that after phase 1, we have to assure that the maximality condition still holds. This is normally conveyed by a random loop over the remaining blocks.

Finally, DCBA also handles the consistency of rules by defining the concept of consistent blocks [13, 10]: rules within a block have the same left-hand side and the same charge in the right-hand side. There is a further restriction within phase 1: if two non-consistent blocks (having different associated right-hand charge) can be selected in a configuration, the simulation algorithm will return an error, or optionally non-deterministically choose a subset of consistent blocks.

2.2 GPU computing

Today, PC's processors offer from 2 to 16 computing cores, and this number can be increased to 64 or even 128 in high end equipments. These cores are complex enough to run threads simultaneously, each one with its own context, exploiting a coarse grain level of parallelism. For example, OpenMP [22] is a threading library for multicore processors, which can be used in C/C++.

High Performance Computing world has changed in the past years. The introduction of the GPU [8] as a co-processor unit to compute and render 3-D graphics, encouraged the change of trend in HPC solutions and start to consider heterogeneous platforms having CPUs and co-processors. The GPU has been devised as a highly parallel processor since it was conceived, and now, GPGPU enables the GPU to be used for general purpose scientific applications [21].

A GPU consists in SIMD multiprocessors interconnected to a fast bus with the main memory system [15, 9]. Each multiprocessor has a set of computing cores that execute instructions synchronously (they always perform the same instruction over different data) and a small portion of sketchpad memory (similar to caches in CPUs, but manually managed by programmers), among other elements. Current GPUs also implement cache memories (one L2 at the level of the memory system, and a L1 cache which resides within the sketchpad memory).

Fortunately, all these aspects are abstracted to the programmer with high level programming models such as CUDA [9, 23]. Introduced by NVIDIA in 2007, CUDA

allows to run thousands of lightweight *threads* concurrently arranged in *blocks*. Threads belonging to the same block can cooperate and easily be synchronized. Threads from different blocks can only be synchronized by finishing their execution. All these threads execute the same code, called *kernel*, in a SPMD (single-program, multiple-data) fashion, since they can access to different pieces of data by using the identifiers associated to each thread and block. Moreover, each thread can also take different branches of execution, but this is penalized when happened within a *warp* (a group of 32 threads), given that it will makes the execution to be serialized. The largest but slowest memory system is called global memory, whereas the smallest but fastest sketchpad memory belonging to each block is called shared memory. The access to these memories should be done carefully, since best bandwidth is achieved when threads access to memory in coalesced (to contiguous addresses) and aligned way [15].

Finally, the GPU architecture has been improving by the different releases. GT800, Fermi, Kepler and Maxwell are the codename of each NVIDIA GPU generation. Each one has been associated to a Compute Capability (CC), 1.X, 2.X, 3.X, and 5.X, respectively [23].

2.3 PDP systems parallel simulation on the GPU

As mentioned above, the main objective of DCBA is to improve the accuracy of the algorithm. However, it comes at expenses of low efficiency. Currently, P-Lingua framework implements the algorithm, but it is usually not recommended when dealing with large models because of the large simulation times. This lack of efficiency is mainly due to the use of Java Virtual Machine and sequential algorithms. Indeed, simulating massively parallel devices like P systems in a sequential fashion is twice inefficient. A solution to outcome this issue is by harnessing the highly parallel architecture within modern processors to map the massively parallelism of P systems [10].

GPUs provide a good platform to implement real parallelism of P systems in a natural way, by using their high level parallelism [11]. Most of P systems simulators based on GPU are within the scope of PMCGPU (*Parallel simulators for Membrane Computing on the GPU*) software project [26], which aims to gather efforts on parallelizing P system simulators with GPU computing. Specifically, there is a subproject for PDP systems, called ABCD-GPU.

ABCD-GPU started with a multi-core version [12, 10], based on C++ and OpenMP, in which the environments and/or the simulations are distributed along the processors. Experiments showed that parallelizing by simulations leads to better speedups; that is, in a multiprocessor CPU, it is better to parallelize coarsely. In order to deal with finer-grain parallelism, a CUDA version has been also developed [14, 10]. In general, these parallel simulators are based on the following principles:

- Efficient representation of the data, both for PDP system syntactical elements and auxiliary structures of DCBA. In this concern, the static and dynamic

tables for phase 1 are not really implemented. Instead, the operations over these tables are translated to operations over the syntactical elements of the PDP system, together with much smaller structures. This approach is called virtual table, and has shown to dramatically decrease the required amount of data and time in DCBA.
- Exploiting levels of parallelism presented in the simulation of PDP systems: processing of rule blocks and rules, evolution of environments, and conducting several simulations to extract statistical data from the probabilistic model.

As mentioned in previous section, CUDA requires a large amount of parallelism to effectively use GPUs resources [9]. Parallelizing only by simulations as in the OpenMP version is not enough, and the parallelism level is coarse. Instead, the solution was to extract more parallelism from the PDP systems as follows [14]:

- Thread blocks: they are assigned to each environment and each simulation. For each transition step, there is a minimal communication along environments (only when executing communication rules), and each simulation can be executed independently.
- Threads: each thread is assigned to each rule block/column in selection phases (1, 2 and 3). In execution phase (4), threads will execute rules in parallel. As it is possible to have more rule blocks than threads per thread block, they perform a loop over rule blocks in tiles.

So far, ABCD-GPU simulator has been tested by using randomly generated PDP systems. The goal of this was to provide a flexible way to construct benchmarks for performance analysis, by stressing the simulator with different topologies. For example, Table 1[14] shows the performance of the simulator with PDP systems having different lengths of the left-hand sides (in terms of number of different objects in the multisets u and v) in average, and running on a NVIDIA Tesla C1060 GPU, which has 240 cores and CC 1.3. These results clearly show that phase 2 is the bottleneck of the simulator, since it is the less parallel phase consisting in a random sequential loop. Moreover, when the competition for objects increase (having more objects in the LHS leads to more competitions), overall performance drastically decreases.

	Test with average LHS length of 1.5		
	% CPU	% GPU	Speedup
Phase 1	53.7%	30.1%	14.23x
Phase 2	12.6%	47%	2.13x
Phase 3	22.6%	13.7%	13.2x
Phase 4	11.1%	9.2%	9.7x

Table 1. Performance testing through randomly generated PDP systems.

3 A new input module: binary files

After the first version of ABCD-GPU [14], the efforts were focused on creating a input module to read PDP system descriptions. In this section, we briefly present the new features of the ABCD-GPU simulator, which is a module to read binary files defining PDP systems models. We also show preliminary results of the simulator with a real ecosystem model.

3.1 Format definition

Similarly to the simulator of P systems with active membranes [10, 11], the design decision for the input file was a binary format. The reason for this is twofold:

- *Size of files*: the GPU simulator is conceived for running very large models. Otherwise, it is not worth to be used. Thus, the communication with the simulator should be as efficient as possible to avoid overheads. Since we use P-Lingua for describing PDP system models, it makes sense to use pLinguaCore to parse the files. In this concern, P-Lingua is used as the parser and compiler which send a file to the simulator with unwrapped rules (recall that rules in P-Lingua can be defined in a symbolic way). Thus, in order to reduce the size of the file as much as possible, we have defined a binary format which assign the less bits to each syntactic element.
- *Efficiency*: related with the latter, the binary file is also organized in such a way that it fits well with the initialization of structures in the simulator. This helps the efficiency of the parser, while reducing the size of the files.

Although using this kind of format lead to a coupled design (between the P-Lingua parser and the simulator), it will allow to use the GPU engine while reducing the communication/storage cost.

Next, we show the structure of the format for the binary file, which is divided into 5 sections:

- Header: unequivocally identify this file as a binary description file for PDP systems.
- Sub-header: defines the accuracy used along the file, for the different fields. This allows to use the exact number of bytes according to the number of objects, rules, etc.
- Global sizes: define the size of alphabet, number of rules, membranes, environments and membrane structure.
- Rule blocks: their information is given in 3 subsections, each one giving information for allocating space related with the next one.
- Initial configuration description.

```
1 ######################################################################
2 # Binary file format for the input of the simulator: PDP systems
3 # (revision 16-09-2014). The encoded numbers must be in big-endian
```

```
4
5
6   # Header (4 Bytes):
7   0xAF
8   0x12
9   0xFA
10  0x21 (Last Byte: 4 bits for P system model, 4 bits for file version)
11
12  # Sub-header (3 Bytes):
13  Bit-accuracy mask (2 Bytes, 2 bits for each number N (meaning a precision
14  of 2^N Bytes)), for:
15  - Num. of objects (2 bits (meaning 2^0 -- 2^2 Bytes))
16  - Num. of environments (2 bits (meaning 2^0 -- 2^2 Bytes))
17  - Num. of membranes (2 bits (meaning 2^0 -- 2^2 Bytes))
18  - Num. of skeleton rules (2 bits (meaning 2^0 -- 2^2 Bytes))
19  - Num. of environment rules (2 bits (meaning 2^0 -- 2^2 Bytes))
20  - Object multiplicities in rules (2 bits (meaning 2^0 -- 2^2 Bytes))
21  - Initial num. of objects in membranes (2 bits (meaning 2^0 -- 2^2 Bytes))
22  - Multiplicities in initial multisets (2 bits (meaning 2^0 -- 2^2 Bytes))
23  Listing char strings (1 Byte, 5 bits reserved + 3 bits), for:
24  - Reserved (5 bits)
25  - Alphabet (1 bit)
26  - Environments (1 bit)
27  - Membranes (1 bit)
28
29
30  #---- Global sizes
31
32  # Alphabet
33  Number of objects in the alphabet (1-4 Bytes)
34  ## For each object (implicit identificator given by the order)
35  Char string representing the object (finished by '\0')
36
37
38  # Environments
39  Number of environments, m parameter (1-4 Bytes)
40  ## For each environment (implicit identificator given by the order)
41  Char string representing the environment (finished by '\0')
42
43
44  # Membranes (including the environment space as a membrane)
45  Number of membranes, q parameter + 1 (1-4 Bytes)
46  ## For each membrane (implicit identificator given by the order,
47                        from 1 (0 denotes environment))
48  Parent membrane ID (1-4 Bytes)
49  Char string representing the label (finished by '\0')
50
51
```

```
# Number of rule blocks
Number of rule blocks of Pi/Skeleton (1-4 Bytes)
Number of rule blocks of the environments (1-4 Bytes)


#---- Information of rule blocks: number rules and length LHS

# For each rule block of Pi (skeleton)
Information Byte (1 Byte: 2 bits for precision of multiplicity in L/RHS
                  (2^0 -- 2^2 Bytes) + 1 bit precision number of objects
                  in LHS (2^0 -- 2^1 Bytes) + 1 bit precision number of
                  objects in RHS (2^0 -- 2^1 Bytes) + 2 bits precision
                  number of rules in the block (2^0 -- 2^2 Bytes) + 1 bit
                  don't show probability for each environment + 1 bit show
                  parent membrane)
Number of rules inside the block (1-4 Bytes)
Number of objects in LHS; that is, length U + length V (1-2 Bytes)
Active Membrane (1-4 Bytes)
# If show parent membrane flag is active (deprecated)
Parent Membrane (1-4 Bytes, this is deprecated)
Charges (1 Byte: 2 bits for alpha, 2 bits for alpha', 4 bits reserved,
         using 0=0, +=1, -=2)

# For each rule block of environment
Information Byte (1 Byte: 2 bits for precision of multiplicity in LHS
                  (2^0 -- 2^2 Bytes) + 1 bit precision number of objects
                  in LHS (2^0 -- 2^1 Bytes) + 1 bit precision number of
                  objects in RHS (2^0 -- 2^1 Bytes) + 2 bits precision of
                  number of rules in the block (2^0 -- 2^2 Bytes) + 1 bit
                  probability for each environment + 1 bit show parent
                  membrane)
Number of rules inside the block (1-2 Bytes)
Environment (1-4 Bytes)


#---- Information of rule blocks: length RHS, probabilities and LHS

# For each rule block of Pi
## For each rule
Number of objects in RHS; that is, length U' + length V' (1-2 Bytes)
### For each environment
Probability first 4 decimals (prob*10000) (2 Bytes)
## For LHS U: multiset in the LHS in the parent membrane U [ V ]_h^a
Number of objects in U (1-2 Bytes)
### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)
## For LHS V: multiset in the LHS in the active membrane U [ V ]_h^a
```

```
Number of objects in V (1-2 Bytes)
### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

# For each rule block of environment
Object in LHS (1-4 Bytes)
## For each rule
Number of objects (involved environments) in RHS (1-2 Bytes)
Probability first 4 decimals (prob*10000) (2 Bytes)


#---- Information of rule blocks: RHS

# For each rule block of Pi
## For each rule
### For RHS U': multiset in the RHS in the parent membrane U' [ V' ]_h^a'
Number of objects in U' (1-2 Bytes)
#### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)
### For RHS V': multiset in the RHS in the active membrane U' [ V' ]_h^a'
Number of objects in V' (1-2 Bytes)
#### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

# For each rule block of environment
## For each rule
#### For each object in RHS
Object ID (1-4 Bytes)
Environment (1-4 Bytes)


#---- Initial multisets and sekeleton states

# For each environment
## For each membrane (membrane 0 for environment)
Charge (1 Byte: 2 bits, 6 bits reserved, using 0=0, +=1, -=2)
Number of different objects in the membrane (1-4 Bytes)
## For each object:
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

```

3.2 Input/output parsers

The ABCD-GPU simulator has been extended with a input module which is able of reading the above described binary format. Currently, the version is still experimental, and in order to decouple the input parser from the simulator structures, the module creates temporal data structures. Of course, in the final version, these structures should be avoided, making the reading of input files more efficient. The input PDP systems can be used both by the CPU and the GPU simulators.

On the other side, a first output module has been also developed. So far, the results were printed on screen. Today, it is possible to generate CSV (Comma Separated Values) files, which can be opened by statistics software such as R and Excel.

3.3 Preliminary results: a real ecosystem model

Thanks to this input module, we have been able to test our simulator with a real ecosystem model. We have chosen the model of the Bearded Vulture ecosystem in the Pyrenees, presented in [2], for its simplicity, allowing us to perform debugging and performance testing.

We have run two tests with 100 simulations, and 47 time steps (as required by the model for 10 years), using two GPUs from different generations: (a) Tesla C1060 (GT800 architecture), and (b) GeForce GTX550 (Fermi architecture). Unfortunately, we couldn't use our Tesla K40 (Kepler architecture) yet since some artifacts happened in the simulator, that requires more debugging and testing. Table 2 shows the preliminary results extracted from our simulators.

	Tesla C1060			GTX550		
	% CPU	% GPU	Acc	% CPU	% GPU	Acc
Phase 1	53.8%	56%	4.2x	53.8%	61.2%	12.6x
Phase 2	1.6%	2%	3.4x	1.6%	6.5%	3.5x
Phase 3	37%	9.4%	17.2x	37%	22.8%	23.3x
Phase 4	7.6%	32.6%	1.02x	7.6%	9.5%	11.6x
Total			4.38x			14.4x

Table 2. Profiling the Bearded Vulture ecosystem model (2008)

At first glance, the results show that the GPU (b) (GTX550) achieves better performance, up to 14.4x of speedup with respect to the sequential version, while the GPU (a) achieves barely 4.38x. As we have shown before, GPU (a) can achieve up to 7x of speedup with randomly generated P systems. However, we can see two new behaviors that were not expected before:

- Phase 2 is not the bottleneck: it is easy to see that the considered model has no competition for objects. Thus, phase 2 is not required for its simulation (the only mechanism carried out is the checking of remaining active blocks).

- Phase 4 is the bottleneck: this results is completely unexpected at first glance. However, a deeper analysis shows that since a few ratio of rules is executed, and most of them has common objects in the right hand side, the generation of objects is not performed completely efficiently. The main reason is the usage of atomic operations for adding new objects.

However, the behavior is completely different on GPU (b):

- Phase 2 is again the bottleneck. Although it is not required, the first phase of the kernel is run, which checks the remaining active blocks.
- The rest of phases are well accelerated. This demonstrates that the better bandwidth and the L2 cache of this GPU help to achieve better speedups when simulating PDP systems.

4 Road map

In what follows, we will discuss some of the future development lines under our consideration for next versions. In fact, this list is the road map for the ABCD-GPU project.

1. Making available Phase1_filter Kernel for GPUs with CC 3.x. In a Tesla K40 GPU, the consistency checking between rule blocks fails randomly.
2. More work on the obtaining the results from the GPU:
   (a) Use asynchronous copy from the GPU (it will require a double buffer for the multisets).
   (b) Filtering the multisets on the GPU, according to some parameters defined by the user.
   (c) Finishing the output module for binary files.
   (d) Development of a module that uploads the results into a database (interoperability with MeCoSim framework).
3. Phase 4 is becoming a bottleneck when running real ecosystem models. We have to change the scatter strategy into a gather one. That is, the threads reads the selection number for each rule, and create the corresponding objects. Why not using hybrid approaches, or a queue-levels approximation? That is, perform some atomics operations on shared memory, and then dump them to global memory. However, this is not easy, because the multiset structure might not fit into shared memory.
4. Phase 2 is also very slow.
   (a) Auto-detect if Phase 2 is really required. For example, if we know that the model has no competition of objects. Or if we analyze the number of active blocks remaining after Phase 1. Otherwise, we can skip it.
   (b) Compact active blocks after phase 1 for more efficiency.
   (c) Real (random) disorder of rule blocks (maybe taking some ideas from [7]). Currently, the random order is given by the thread scheduler (not a really random).

5. Avoid current synchronization of DCBA phases. That is, run all the phases with one single kernel (perhaps one global kernel which calls to __device__ versions of current kernels). It could be convenient to maintain the original version for GPUs that are used for the graphic system on the computer (limitation of kernel time).
6. In PDP systems, the working alphabet for the skeleton and for the environments are disjoint. That is, $\Gamma \cap \Sigma = \emptyset$. Therefore, we can work with all the communication rules apart from the virtual table.
7. Implement a variant of DCBA, called μDCBA:
   (a) It will allow to extract more parallelism within each environment. If we pre-calculate the group of rules that really depend on each other because they compete for objects, we will be able to apply DCBA separately to each group, i.e. more locally and in parallel. Moreover, there will be less resources to handle (and perhaps we would be able to move more data into shared memory, such as the multisets).
   (b) We define a transitive relation between rule blocks, called *competition*: block b_i *directly* compete for objects with block b_j if they have overlapping but not equal left-hand side. Moreover, if b_k directly compete with B_j, but not with B_i, then B_i and B_k also compete for objects (however, indirectly through B_j).
   (c) The idea is to define disjoint sets of rule blocks holding the competition relation, and apply DCBA to each one.
   (d) It would be desirable to use this variant only when the sets are balanced. We could also assign different "small" sets to one thread block.
8. Improving the data structures. Concerning the storage of objects appearing in the left-hand side of the rule (blocks), the current implementation on the GPU could be improved.
   (a) Current implementation is based on CSR representation of sparse matrices. All the objects of all the left-hand sides (LHS) are stored consecutively in a single array (see Figure 1), the array at the bottom). Each rule block has a corresponding entry in an array that contains a pointer (the index) to the array for LHS objects. This index says the first object of the LHS, and the end is given by the index of the next rule block. In this way, for example, ruleblock 4 has no object in LHS (what is weird, but can take place in our implementation), since idx5-idx4=0. However, rule block 2 has 3 objects, since idx3-idx2=3.
   (b) The problem is that each thread will iterate the objects of the LHS of each rule block. However, the access to the array of LHS is not coalesce, what is really bad for performance.
   (c) The idea would be to use the ELL representation of sparse matrices, and compact the objects in the LHS by chunks of consecutive rule blocks. Rule blocks with short LHS will need to replicate with dummy objects. In this way, the access is made coalesced (see Figure 2). The array of

objects should be linear (that's why there are lines connecting each chunk). Moreover, the array of lengths could be avoided.

(d) This will entail an interesting research. Is it good to do it or not? How much is the waste of memory? Can we use it only when LHS lengths are more or less balanced? Otherwise, can we use the COO representation? Is it really much more faster? What about Fermi architecture? Will their L2 cache improve the results?, is it good or not?

9. Implement model-oriented optimizations. That is, to analyze the PDP system model prior to the simulation and extract properties that will help to the efficiency. For example, test if there are competition for objects, inconsistent rule blocks, etc.
10. *Parallel P-Lingua*. Moreover, it would be interesting to let the model designer to provide the above mentioned properties to the simulator. For example, to allow in P-Lingua the usage of directives for defining modules of rules that can be executed in parallel, similarly to the `pragma` directives in OpenMP.
11. Hybrid simulation of PDP systems, by using both the CPU and GPU platforms at the same time, and implement a merge module of simulations at the end of the process.

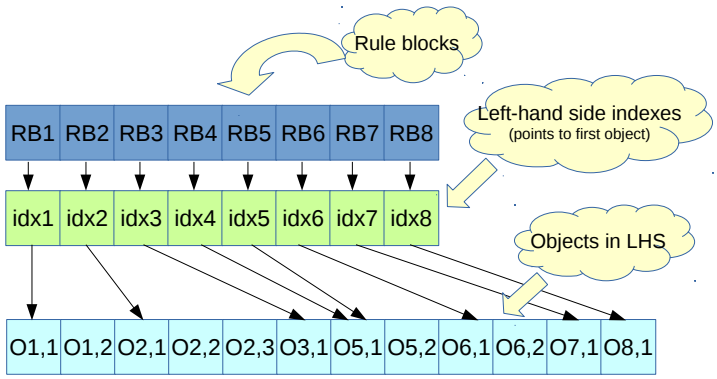

Fig. 1. Data structure for storing the information of left-hand sides of rule blocks, as currently implemented.

5 Conclusions

In this paper, we have shown the preliminary results related with the input module for the CPU/GPU simulators of PDP systems (ABCD-GPU). This module supports files with a binary format, which we have introduced here. The purpose

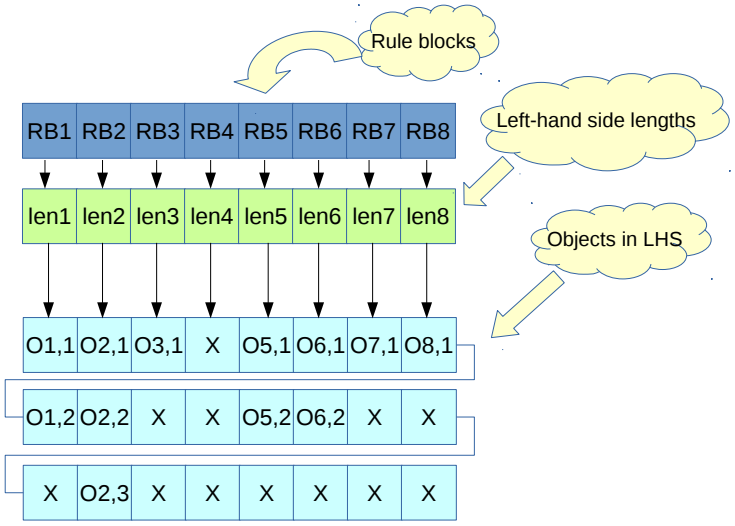

Fig. 2. Data structure for storing the information of left-hand sides of rule blocks, as proposed.

of using a restricted, binary format is for efficiency. We have also shown that simulating a real ecosystem model leads to different behaviors, depending on the GPU generation. Specifically, we have seen that phase 2 is the bottleneck for a Tesla C1060, while phase 4 is for a GTX 550 (Fermi).

Figure 3 shows the current structure of the project. The simulation engine implements DCBA in both multicore (CPU) and manycore (GPU) platforms. The input files are generated by pLinguaCore, which acts as a parser in the creation of binary files. The output files will be both in CSV and binary formats soon, and a module to upload results to a database is also under consideration. Moreover, the platform still support the input of randomly generated PDP systems and the output of corresponding profiling and debugging information, in order to conduct performance benchmarks to new versions of the simulator.

Finally, it is noteworthy that in this case, Parallel Computing is not only used to get faster solutions, but also, to obtain better results, because it enables the users to run DCBA-based simulations in an affordable time.

Acknowledgements

The authors acknowledge the support of the project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, co-financed by FEDER funds. Miguel A. Martínez-del-Amor also acknowledges the support of the Alain Ben-

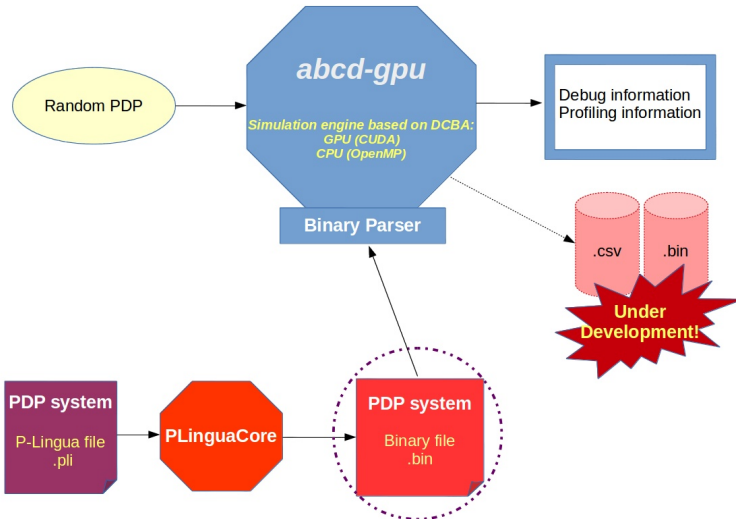

Fig. 3. Structure of ABCD-GPU project.

soussan Fellowship programme of ERCIM, and the hosting institution Fraunhofer IIS.

References

1. M. Cardona, M.A. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M.J. Pérez-Jiménez, D. Sanuy. A computational modeling for real ecosystems based on P systems, *Natural Computing*, **10**, 1 (2011), 39–53.
2. M. Cardona, M.A. Colomer, A. Margalida, I. Pérez-Hurtado, M.J. Pérez-Jiménez, D. Sanuy. A P system based model of an ecosystem of some scavenger birds, *LNCS*, **5957**, (2010), 182–195.
3. 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. Emergence, Complexity and Computation series*, **Volume 7**. Chapter 4, pp. 97–132, 2014, Springer Int. Publishing.
4. 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** (4): e60698 (2013)
5. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Agustín Riscos-Núñez. An overview of P-Lingua 2.0, *LNCS*, **5957** (2010), 264–288.
6. M. García-Quismondo, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez. Probabilistic Guarded P systems: A new formal modelling framework. *LNCS*, **8961** (2014), 194–214.

7. A. Gastalver-Rubio. *Simulation of probabilistic P systems on GPUs*, Final Research Project, University of Seville, September 2012.
8. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY (USA), 2005.
9. D. Kirk, W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*, MA (USA), 2010.
10. M.A. Martínez-del-Amor. *Accelerating Membrane Systems Simulators using High Performance Computing with GPU*, Ph.D. thesis, University of Seville, May 2013.
11. 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. Fundamenta Informaticae, **136**, 3 (2015), 269–284
12. 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, *Proc. Tenth Brainstorming Week on Membrane Computing*, Sevilla, Spain, **Volume II**, 2012, pp. 17–26.
13. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani, A. Riscos-Núñez, M.A. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P Systems with Proportional Object Distribution, *LNCS*, **7762** (2012), 27–56.
14. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P systems on CUDA. *10th Conference on Computational Methods in Systems Biology*, *LNBI*, **7605** (2012), 247–266.
15. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. GPU Computing, *Proceedings of the IEEE*, **96**, 5 (2008), 879-899.
16. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998
17. Gh. Păun, Gz. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, USA, 2010.
18. I. Pérez-Hurtado, L. Valencia-Cabrera, M.J. Pérez-Jiménez, M.A. Colomer, A. Riscos-Núñez. MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems, *Proceedings IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010)*, **volume I** (2010), pp. 637–643.
19. M.J. Pérez-Jiménez, F.J. Romero-Campero. P systems, a new computational modelling tool for Systems Biology. *Transactions on Computational Systems Biology VI. LNBI*, **4220** (2006), 176–197.
20. M.J. Pérez-Jiménez, F.J. Romero-Campero. A model of the Quorum Sensing system in Vibrio fischeri using P systems. *Artificial Life*, **14**, 1 (2008), 95–109.
21. *GPGPU organization*. `http://www.gpgpu.org`
22. *OpenMP webiste*. `http://www.openmp.org`
23. *NVIDIA CUDA website*, 2015. `https://developer.nvidia.com/cuda-zone`
24. *The MeCoSim web page*. `http://www.p-lingua.org/mecosim`
25. *The P-Lingua web page*. `http://www.p-lingua.org`
26. *The PMCGPU project*, 2013. `http://sourceforge.net/p/pmcgpu`