Design of Specific P Systems Simulators on GPUs

Miguel Á. Martínez-del-Amor(✉), David Orellana-Martín, Ignacio Pérez-Hurtado, Luis Valencia-Cabrera, Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez

Research Group on Natural Computing,
Universidad de Sevilla, Avda. Reina Mercedes S/N, 41012 Sevilla, Spain
{mdelamor,dorellana,perezh,lvalencia,ariscosn,marper}@us.es

Abstract. In order to validate P system models and to assist on their formal verification, simulators are indispensable. Moreover, having effi-cient simulation tools is crucial, and for this purpose, parallel platforms should be employed. So far, several parallel simulators for P systems have been developed, specifically targeting GPUs (Graphics Processing Units). Although being a hot topic within Membrane Computing, map-ping P system parallelism on GPUs is still not a mature area. In the past, we have successfully accelerated the simulation of two specific fam-ilies of P systems solving SAT with GPUs, and learned in the process some semantics ingredients that fit well on these parallel devices. We are extending this exploration by designing an specific simulator of a P system model for the `FACTORIZATION` problem. In this paper, we analyse the two main approaches for simulators, and depict some design decisions required for this case study.

Keywords: P systems · Parallel simulation · GPU computing

1 Introduction

Parallel simulation of P systems is of increasingly importance. Simulating membrane systems enable model designers to verify and validate their work, so efficient simulation tools can save time in this process. Moreover, identifying how the bioinspired parallelism of these devices can be handled by current parallel platforms can help to drive next generation technology.

Previous work in this concern has been to simulate different solutions of the same problem in order to isolate P system ingredients that fit well into the parallel architecture of GPUs. In [2], a GPU simulator for a family of P systems with active membranes and division of elementary membranes solving `SAT` was introduced. This was the first *specific* simulator on the GPU to be defined. In [5], a family of tissue P systems with cell division solving `SAT` was introduced. The former achieved an speedup of 63× compared to a sequential counterpart, while the latter obtained a 10× of acceleration. Using polarizations in the cell-like

model helped to reduce the amount of present objects in the membranes, and its representation has a minor impact to the GPU in terms of memory consumption.

The aforementioned simulators were designed for specific families of P systems, so tailored code was developed. There were also developments concerning *generic* simulators for P system variants, being the first one for P systems with active membranes and elementary division [1]. In this simulator, any P system for that model can be simulated (under certain pre-established restrictions), requiring to invest lot of resources for worst cases that are rare to happen in a real model. For this reason, a stressing test with toy models lead to speedups of up to 7×, but with the family solving `SAT` just 1.5× [4].

A new solution to the `FACTORIZATION` problem using computing P systems has been proposed in [6]. Let us recall that the version of `FACTORIZATION` problem considered in the cited paper is the following: *given a natural number which is the product of two prime numbers, find its decomposition.* A solution to this problem is provided by a family of (binary) computing polarizationless P systems with active membranes making use of minimal cooperation and minimal production (without dissolution rules and without division rules for non-elementary membranes). Minimal cooperation stands for having rules with left-hand side (LHS) length of at most 2, and minimal production means to have rules with right-hand side (RHS) lengths of at most 1.

In this paper, we present the first design decisions of a GPU simulator for the family of computing membrane systems solving the `FACTORIZATION` problem. This design is feasible thanks to a key feature: minimal production. In this way, it is possible to constrain the size of the membrane representation in the simulator. Furthermore, the amount of present objects can be saved by using internal counters. Moreover, binary representation of the input natural number can be natively represented using unsigned integer numbers on the GPU. This also shades a light on how to design simulators for specific models.

The rest of the paper is structured as follows: Sect. 2 introduces very briefly the key concepts of GPU computing to understand the taken decisions. Section 3 discuses the two approaches when developing parallel P system simulators. Section 4 depicts the design of the simulator while discussing the ingredients that have enabled that. Section 5 ends the paper with conclusions and future work.

2 Core Concepts of GPU Computing

When programming a GPU with CUDA, firstly, the data structures that are going to be used have to be allocated on the device. This constrains the flexibility while increasing the efficiency: reserving new memory on the fly can really slowdown the performance of the code [3]. Once all the data structures are created, the necessary data is sent from the CPU. At this point, the GPU is ready to launch threads that will execute the same code (called kernel) in parallel. The work assignation has to be carefully done by the programmer to balance the amount of threads doing actual job, and how to access the

memory (which requires coalesced access to contiguous data to harness best performance). Thread execution is made hierarchically, being arranged in thread blocks. Threads within thread blocks can be synchronized and can cooperate through efficient small memory called shared memory.

3 Generic Versus Specific Parallel Simulators

There are two main approaches when developing GPU simulators for P systems: generic and specific. The former refers to simulators designed to accept a broad range of models within a variant. The latter corresponds to ad-hoc simulators for certain families or models. In CUDA, kernels run faster using static data structure without dynamic memory (i.e. allocated at the beginning of the code). For this purpose, simulators have to consider the worst cases in order to avoid memory conflicts during the simulation.

Generic simulators require to allocate GPU memory for both rule information, auxiliary data and system configurations (see Fig. 1, top-left), with enough space for all possible objects that can be generated (in the worst case, the whole alphabet). The simulation algorithm conceives two main steps: selection (Fig. 1, top-right) and execution of rules (Fig. 1, bottom-left). In order to perform selection of rules, threads need to access to rule information (to consult the LHS) and the current configuration (to seek existing objects and membrane charges), and write the result in the auxiliary data structure (executions of each rule). For execution, threads have to read the auxiliary data (rule selections) and the rule information (for the RHS) and write the new configuration. Finally, the result of the simulation is copied back to the host space for its output (Fig. 1, bottom-right).

On the contrary, specific simulators only need to allocate GPU memory for objects that are known to appear at the same time. This is normally related to the input multiset for some systems. Moreover, rule information is directly encoded in the source code. There is no need to implement a two-step algorithm given that it is known which rules are going to be executed at every step.

4 Design of a Parallel Simulator

The first step to develop a specific simulator is to design an efficient data structure: it has to be limited by containing just the required information, and has to dispose the data contiguously to enable coalesced accesses by threads. As discussed in [3, 4], one performance attribute of P system GPU simulators is object density: if we cannot estimate an upper bound for the number of different objects that can appear in a membrane, then we have to allocate space for all objects defined in the alphabet. In such a case, an array of integers stating the multiplicity of each object defined in the alphabet is required, called *unbounded* representation (see Fig. 2, top). However, if few different objects appear inside the membranes, the GPU will handle a sparse array given that the majority of multiplicities are 0. This has a negative impact in the performance, because if we use a thread to process each object, most of the threads will be idle in any case.

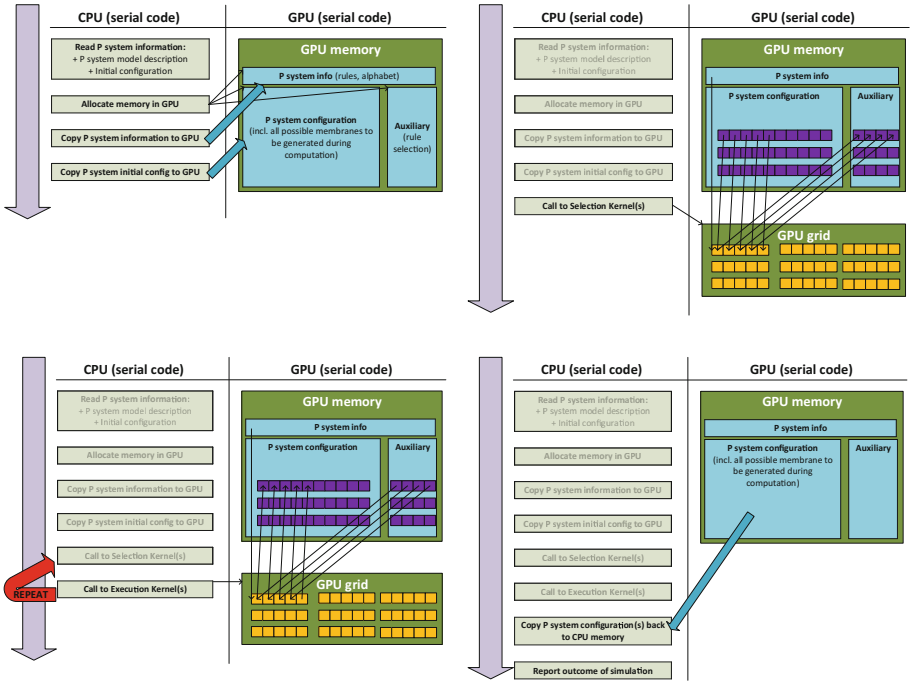

Fig. 1. Scheme of a generic P system simulator on CUDA. For all images, the left part belongs to the CPU side (host) and the right part the GPU (device). Top-left image corresponds to the initialization, where the input model is read and GPU memory is allocated. Top-right figure shows a GPU grid of threads reading information for rule selection, and annotating the outcome. Bottom-left shows the execution of rules by the threads, and the loop over selection and execution. Finally, bottom-right shows the retrieval of the result of the simulation.

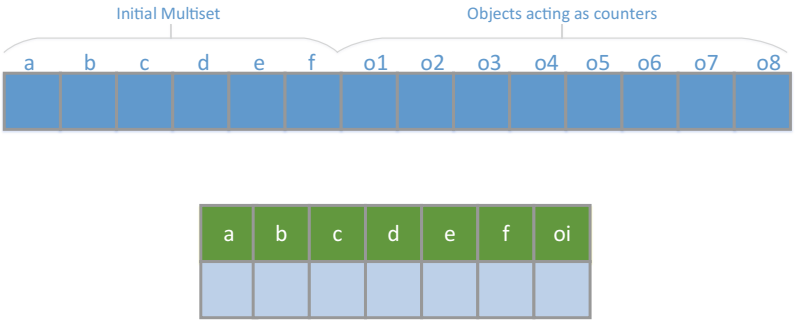

Fig. 2. Unbounded (top) vs limited (bottom) representation of membranes.

In the solution for FACTORIZATION problem, the family of computing P systems uses minimal production. Thus, and when not having send-in rules, we can guarantee that inside membranes there will be no more objects than the initial multiset, but there can be fewer because the RHS of rules can be the empty multiset. The latter causes minor impact for threads, if it is not frequent. The data structure reduces the object density impact thanks to restricting the size of membranes to just the size of the input multiset, but it has to store two values per object: the multiset value, and the corresponding object symbol or identifier. Indeed, now the objects are replaced when applying the rules, and since there is not an array position for each object, we need to annotate which object is being represented. This is called *bounded* representation, and can be seen in Fig. 2, bottom. Thus, the downside of this design is that when using minimal cooperation, a search for the objects appearing in the LHS has to be performed. However, for this specific case (an ad-hoc simulator), it is not an issue, given that we know exactly where to search in each stage beforehand. But a future generic simulator will need to implement more elaborated algorithms.

Moreover, in a specific simulator, not all objects have to be defined in an explicit way; that is, stored in the data structure for multiplicities. Given that the simulator is very specific for a solution, it is possible to represent objects as variables in the source code, or just to depend on certain variables, so the simulator can infer easily the corresponding multiplicities. This is of special interest for counters in the P system models. Normally, P system designs include objects acting as counters. Their symbols depend on subscripts, but they are different objects in any case. If they are treated as normal objects, and we cannot provide an upper bound for the size of membranes, then we would be wasting lot of resources because only one counter object appear at once while allocating a position to each of them. Moreover, specific simulators can use variables corresponding to the subscript of counter object subscripts for the simulator. That would be enough to maintain all the required information of the model, given that it is easy to infer the multiplicity of an object acting as a counter if we know the counter value.

5 Conclusions and Future Work

In this paper we show the main differences when implementing simulators for generic and specific purposes. We also briefly describe some design decisions to develop a specific simulator for a family of computing P systems solving the FACTORIZATION problem. We have seen that thanks to using minimal production, we are able to restrict the size of membrane representation in the simulator to just the input multiset. Moreover, objects acting as counters do not need to be explicitly represented in the simulator.

For future work, we plan to finalize the design and perform the implementation in order to test all the ideas and identify, if possible, more semantics ingredients that help the parallel execution of rules. Another research line would be to explore new features in GPUs that can help to the performance of the

simulator, such as Dynamic Parallelism or Cooperative Groups. Moreover, kernel code compilation in runtime open doors to develop automatic CUDA code generation tailored to input models.

Acknowledgments. The authors acknowledge the support from the research project MABICAP TIN2017-89842-P, cofinanced by "Ministerio de Economa, Industria y Competitividad" (MINECO) of Spain, through the "Agencia Estatal de Investigacin" (AEI), and by "Fondo Europeo de Desarrollo Regional" (FEDER) of the European Union.

References

1. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with active membranes on CUDA. Briefings Bioinform. **11**(3), 313–322 (2010)
2. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. J. Logic Algebraic Program. **79**(6), 317–325 (2010)
3. Martínez-del-Amor, M.A.: Accelerating membrane systems simulators using high performance computing with GPU, Ph.D. thesis, University of Seville, May 2013
4. Martínez-del-Amor, M.A., García-Quismondo, M., Macías-Ramos, L.F., Valencia-Cabrera, L., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Simulating P systems on GPU devices: a survey. Fundam. Inform. **136**(3), 269–284 (2015)
5. Martínez-del-Amor, M.A., Pérez-Carrasco, J., Pérez-Jiménez, M.J.: Characterizing the parallel simulation of P systems on the GPU. Int. J. Unconventional Comput. **9**(5–6), 405–424 (2013)
6. Orellana-Martín, D., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: The factorization problem: a new approach through membrane systems. In: Accepted paper in Membrane Computing Satellite Workshop of 17th International Conference on Unconventional Computation and Natural Computation, 25–29 June, Fontainebleau, France (2018)