University of Seville
Dpt. of Computer Science
and Artificial Intelligence

Accelerating Membrane Systems Simulators using High Performance Computing with GPU

A Thesis submitted for the degree of
Doctor of Philosophy
School of Computer Engineering
University of Seville

Miguel Ángel Martínez del Amor

Approval of the Thesis Supervisors

PhD. Mario de J. Pérez Jiménez   PhD. Ignacio Pérez Hurtado de Mendoza

May 14, 2013

A Chary,
por ser el mejor regalo de mi vida.

A mis padres,
por darme todo lo que tengo y soy.

Y a mis hermanos y amigos,
siempre por su apoyo incondicional.

Agradecimientos

Esta tesis representa la capacidad de trabajo y sacrificio que he ido adquiriendo desde donde alcanzan mis recuerdos. Nada de esto habría sido posible sin la calidad humana de los que me rodean, que siempre me han alentado a continuar, han conseguido moldear mi personalidad para ser tal y como soy, y han conseguido que me encuentre ahora mismo escribiendo estas palabras: la mejor oportunidad para manifestar mi gratitud.

Me gustaría comenzar por dar las gracias a Chary, el regalo de mi vida, que me ha enseñado a vivir y a amar, y que sin sus ánimos, consejos y apoyo, no habría llegado tan lejos. A mis padres, Lorenzo y Loli, por ofrecerme toda una vida, por su educación y por transmitirme su capacidad de sacrificio, trabajo, y superación. A mis hermanos, José Luis (junto a María y el pequeño Luis) y Mariloli, por siempre estar ahí y por enseñarme a luchar por lo que uno quiere. También a éste primero, mi hermano mayor, por haber sido siempre mi guía y un ejemplo a seguir. A mi abuela, por su cariño, transmitirme toda su experiencia y sabiduría, y ser una segunda madre para mí. Al resto de mi familia, tíos y primos, y a todos mis amigos de infancia y de la peña, por llenar mis recuerdos con muy buenos momentos, de los mejores de mi vida, y que hacen que tenga unas fuertes raíces en mi tierra (Calasparra). Por supuesto, agradecer a todos aquellos que han conseguido hacer de Sevilla un nuevo hogar para mí. Especialmente a mi segunda familia: Carmen y Paco, por abrirme las puertas de su casa y cuidarme como a un hijo más; a Cisco y Virginia, por aceptarme como a un hermano; a Chico y Taor por su fiel amistad; y al resto de familia política que me ha apoyado.

Quiero expresar mi infinita gratitud a todos mis compañeros durante mi etapa de investigador primerizo en el Grupo de Computación Natural de la Universidad de Sevilla. En especial, a mi director Mario de Jesús Pérez Jiménez, por acogerme en su grupo, y por brindarme esta oportunidad única de iniciarme en hacer ciencia bajo su guía. Su capacidad de trabajo y ayuda siempre desinteresada hacia los demás me ha causado un gran efecto positivo. También a mi co-director, Ignacio Pérez Hurtado, por su gran compañerismo, y ser siempre esa mano amiga. Gracias a él mi trayecto en el grupo ha sido mucho más sencillo. A Francisco José Romero, por su gran apoyo e ingenio (agradecimientos especiales por ser el que dio la idea que culminó en el trabajo presentado en mi tesis); a Agustín Riscos, por tener siempre una lección diaria para mí; a Miguel Ángel Gutiérrez por su capacidad de apoyo y de humor; a Luis, Manu y Luis Felipe por ofrecerme siempre su ayuda, apoyo, y fomentar la cooperación y buen ambiente en el grupo; a Ana, Carmen, Álvaro, Fernando y Andrés también por su ayuda incondicional. Asimismo, deseo hacer extensivo este sentimiento de gratitud al resto de los compañeros incluido el personal de administración y servicios, del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. *I would like also to thank all the collaborators (now friends) that passed by our group: Niall, Enrico, Henry, Xiangxiang, and Daniel; because they do not only infected me their passion for science, but also their ability to learn and be open mind with other cultures. Special thanks to Henry Adorna, who started a really interesting and active collaboration with us together with Francis and the Membrane Computing Research Group. Thanks for give us this opportunity to work together, and be also part of your group. Furthermore, I really acknowledge the support of Anne C. Elster during my short stay at Trondheim (Norway), what gave to me the best experience of my life. I would like to extend this to the rest of the HPC-LAB at NTNU, specially for Ian Karlin and Rune E. Jensen, who helped me a lot developing part of my thesis and taught me really interesting and useful things.* Con respecto a la estancia en Trondheim, he de dar las gracias a Alfredo Pérez, quien me cuidó como a un hermano. También debo agradecer la colaboración inicial con Chema y Ginés de la Universidad de Murcia, con quienes comenzamos a desarrollar el trabajo de manera conjunta. Gracias a ellos todo evolucionó rápidamente.

Deseo continuar expresando mis agradecimientos a todos mis formadores y compañeros de formación. A todos los profesores que tuve, desde primaria (EP Nª Sª de la Esperanza) a la universidad (Universidad de Murcia), pasando por secundaria y bachiller (IES Emilio Pérez Piñero), que me motivaron, y sembraron la semilla de la pasión por la ciencia. A todos mis compañeros de fatiga durante aquellos años, en especial en los últimos a José Luis, por su gran apoyo y compañerismo que acabó de forma natural en una gran amistad, al igual que con Fernando, Ana, Alfredo, Juan, Andrés y Fran. *To all the (Portuguese, Brazilian, Italian, German, Belgium and French) people that supported me at my Erasmus experience in Trondheim. Thanks to them all, it was the most personally enriching experience in my life.*

Finalmente, agradezco profundamente todo el apoyo económico ofrecido por la beca de personal investigador en formación asociado al proyecto de excelencia con investigador de reconocida valía P08-TIC04200, de la Junta de Andalucía, y por los proyectos de investigación nacionales I+D+i del Ministerio de Economía y Competitividad TIN2006-13425, TIN2009-13192 y TIN2012-37434; todos ellos confinanciados por los fondos FEDER de la Unión Europea.

Contents

List of Figures

Page

List of Tables

Motivation

Natural Computing is a discipline aiming the study and simulation of the dynamic processes that occur in nature and that are subject to be interpreted as calculation procedures. Inside this discipline, models and computational techniques are researched. They are inspired by nature to better understand the world around us, in terms of information processing.

Membrane Computing [141] is an emerging branch within this area, initiated by Gheorghe Păun at the end of 1998 [140]. This new model of computation starts from the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this model are called *P systems*. They have several *syntactic* ingredients: a *membrane structure* consisting of a hierarchical arrangement of membranes embedded in a *skin* membrane, and delimiting *regions* or compartments where multisets of objects and sets of evolution rules are placed. P systems have also two main *semantic* ingredients: their inherent *parallelism* and *non-determinism*. The objects inside the membranes can evolve according to given rules in a synchronous (in the sense that a global clock is assumed), parallel, and non-deterministic way.

It is worthy to note that we have here a *double parallelism*, once at the level of regions (the rules are used in a parallel way), and once at the level of the system (all regions evolve concurrently). Is this parallelism and non-determinism able to solve computationally hard problems in a "feasible" time? The answer is affirmative [133], but we must point out two considerations. On the one hand, we have to deal with the non-determinism in such a way that the classical notion of acceptance is not a true algorithmic concept [72]. On the other hand, the drastic decrease of the execution time from an exponential to a polynomial one is not achieved for free, but by the use of an exponential workspace (in the form of membranes and objects), although this space is created in polynomial (often linear) time.

Although most research in P systems concentrates on the computational power and efficiency of the devices involved, lately they have been used to *model* biological phenomena within the framework of *Computational Systems Biology* and *Population Dynamics*. In this case, P systems are not used as a computing paradigm, but rather as a formalism for describing the behavior of the system to be modeled. They offer an approach to the development of models for biological systems that meets the requirements of a good modeling framework: relevance, understandability, extensibility and computational / mathematical tractability [142]. In this respect, several P systems models have been proposed to describe *oscillatory systems* [66], *signal transduction* [47, 128], *gene regulation control* [144], *quorum sensing* [152, 143] and metapopulations [25], *metabolic algorithm* [27], *dynamical probabilistic P systems* [25] and *(multicompartmental) Gillespie Algorithm* [143]. These models differ from each other in the type of the rewriting rules, membrane structure and the strategy applied to run the rules in the compartments defined by membranes. Furthermore, *probabilistic P systems* have also been successfully applied as a tool for macroscopic level processes, such as the computational modeling of real ecosystems [33, 50, 34, 53].

In order to *experimentally validate* P systems based models, it is necessary to develop simulators able to be executed on electronic computers, which can help researchers to compute, analyze and extract results from a model [141]. These simulators have to be as *efficient* as possible to handle large size instances, what is one of the major challenges facing today's P systems simulators. In this regard, software applications for Membrane Computing typically implement *sequential* (or with a limited parallelism) simulation algorithms adapted to conventional CPU architectures [141], so they lack the possibility of exploiting the massively parallel nature that P systems present by definition. This is necessary for obtaining a simulation model closer to the theoretical one.

This parallel computation model leads us to look for a massively-parallel technology where a parallel simulator can run more efficiently. The newest generations of *Graphics Processor Units (GPUs)* are *massively parallel processors* which can support several thousand of concurrent *threads*. To date, many general purpose applications have been migrated to these platforms obtaining good *speedups* compared to their corresponding sequential versions [146, 147, 5, 13]. Current NVIDIA GPUs, for example, contain thousand of scalar processing elements per chip [94], and they are programmed using a *C* programming language [88] extension called *CUDA (Compute Unified Device Architecture)* [89, 5, 94, 119, 62].

The aim of this thesis is to *develop* more *efficient* P systems simulators by using *parallel architectures*. This is crucial for the construction of new tools that enable scientists to interact with their models. In particular, NVIDIA *GPUs* with *CUDA* are considered for the development of parallel P systems simulators. Previous results, concerning the stochastic simulation algorithms on the GPU [93], put the seed that encouraged us to consider this technology to face positive results. For this purpose, this work started with the creation of a parallel simulator in CUDA for *recognizer P systems with active membranes* [41]. This variant was chosen for their natural capacity to create an exponential amount of resources in polynomial time, expressed in terms of membranes and/or objects. This first simulator permitted to analyze the behavior of the GPU on the simulation of P systems. Since the simulation of P systems involves addressing a problem with dynamic data nature (objects and membranes are created and deleted along the computation) and with a very large amount of data (the number of membranes created by division rules is exponential), their simulation is memory bound.

After the creation of the parallel simulator for active membranes, another parallel simulator was developed. It is specifically designed for a *family of recognizer P systems with active membranes solving the* `SAT` *problem in linear time* [133, 40]. Thus, this *ad hoc* simulator improves performance of the more generic (flexible) simulators for these specific P systems. In addition, this work was extended to a family of *tissue-like P systems* that efficiently solves the same problem. P systems characteristics that are well suited to be simulated on the GPU are deduced by studying both simulators. Many efforts have been carried out on the GPU simulation of P systems serving as a framework for modeling *population dynamics (PDP systems)*. In this regard, two new *simulation algorithms* have been proposed to better reproduce the semantics of the models. They are called DNDP (Direct Non-Deterministic distribution with Probabilities) [100] and DCBA (Direct Distribution Algorithm based on Consistent Blocks) [98]. After their experimental validation using the standard simulation library *pLinguaCore* [101, 99], a parallel version of DCBA was implemented, which works on both *multicore processors* (OpenMP) [103] and *manycore GPUs* (CUDA) [104], improving the simulation performance.

The source codes of the implemented simulators on the GPU (with CUDA) are available in the software project *PMCGPU (Parallel simulators for Membrane Computing on the GPU)* [17], under the GNU GPLv3 license [2].

Content of the document

This document is structured in four parts that make up a total of eight chapters. Next, we briefly describe its content.

Part I: Preliminaries

The **first chapter** introduces the disciplines of Natural Computing and Membrane Computing. Moreover, formal concepts related to the syntactic and semantics components of P systems is provided. The chapter ends with a brief description of their applications in the computational complexity field.

In **Chapter 2**, the common parts of P systems simulators are analyzed. A summary of the available simulation tools for P systems is also presented. Furthermore, we describe a chronological overview of the P-Lingua simulation framework [70, 16, 124], which is the starting point of the described work herein. Finally, we analyze the necessity of efficient simulators for Membrane Computing, providing a survey of parallel simulators already existing in the area.

This first part ends with **Chapter 3**, which introduces High Performance Computing as a source of solutions to improve the efficiency of the simulators, with special emphasis on Parallel Computing and GPU computing. It also presents the CUDA and Tesla technologies of NVIDIA, that will be used for the experiments throughout the work.

Part II: Parallel simulation applied to efficient solutions of computationally hard problems

Chapter 4 describes the design and development of the simulator for P systems with active membranes based on CUDA and, additionally, the benchmark carried out with the GPU Tesla C1060. Two test cases were analyzed: a simple testing case and a solution from literature for the `SAT` problem. These tests permit to study the properties that P systems have to verify for successfully accelerating their simulation on the GPU.

This second part ends with **Chapter 5**, which describes the development of more specific simulators of two solutions (based on both cell-like and tissue-like P systems) for the `SAT` problem. Moreover, it provides an analysis of the P systems characteristics promoting better performance on the GPU, since two solutions based on different variants are involved.

Part III: Parallel simulation applied to computational models in biology

Chapter 6 presents two simulation algorithms designed to better capture the semantics of P systems modeling population dynamics (called PDP systems, Population Dynamics P systems). To this end, the syntactic and semantic elements of PDP systems are introduced. Then, the DNDP and DCBA algorithms (that will act as inference engines) are described.

This third part of the document ends with **Chapter 7**, which describes the implementations developed for the algorithms introduced in the previous chapter. A description and an analysis of the simulators created within the pLinguaCore simulation library is provided, as well as those independently developed in C++ with OpenMP and CUDA.

Part IV: Conclusions

The document concludes with a chapter devoted to the presentation of conclusions and suggestions of future research directions. Furthermore, it provides best practice guidelines for developers of parallel P systems simulators. They have been obtained from the experience gained during the development of the work presented herein. Finally, two additional chapters are supplied as technical appendices. The first one shows how to use the described simulators, and the second describes the GPU server implemented to run the experiments.

Contributions

It is worth to note the following original contributions of the work described in this document:

- *Development of a parallel simulator in CUDA for recognizer P systems with active membranes.* This simulator receives as input a P system described in a binary file (generated by the pLinguaCore compiler from a P-Lingua file). This simulator is based on the one implemented in pLinguaCore, and uses the GPU to accelerate the simulation. After testing with simple examples, the good performance of the GPU to simulate P systems is showed, since both share a double parallel nature (membranes and rules against thread blocks and threads). This simulator has been developed in collaboration with researchers from the Parallel Computing Architecture Group from the University of Murcia. In addition, this work has been presented at several conferences of different areas (*First International Workshop on High Performance Computational Systems Biology*, Trento, Italy, 2009; *Tenth Workshop on Membrane Computing*, Curtea de Arges, Romania, 2009; and *Symposium on Application Accelerators in High-Performance Computing*, Urbana-Champaign, Illinois, USA, 2009), and published in journals, the first of which is included in the ISI Journal Citation Reports ranking, with great impact on the scientific community:

  – J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11, 3 (2010), 313-322. $\boxed{\textbf{JCR 9.283}}$
  – J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Implementing P systems parallelism by means of GPUs. *Lecture Notes in Computer Science*, 5957 (2010), 227-241.

- *Development of a parallel simulator in CUDA for a family of P systems that solves* `SAT` *in linear time.* After experimenting with the previous parallel simulator, the simulation time on the GPU for P systems based solutions to **NP**-complete problems is not decreased compared to the equivalent sequential version. The achieved performance by the previous flexible simulator depends on the P system to simulate, and it has to follow the behavior of the simulator on the GPU. This result leads us to develop simulators for specific P systems solutions. This is achieved by optimizing the *ad hoc* simulation algorithm and its data structures, while obtaining an improvement of the order of up to 90 times faster. Thus, the GPU is demonstrated to be a good choice for simulating P systems, in case of properly adapting the specific P system simulation to the GPU architecture. This simulator has also been developed in the collaboration framework with researchers from the University of Murcia and the University of Malaga. In addition, this work has led to new contributions in the area of Parallel Computing, with new implementations optimized for GPU and supercomputer architectures, resulting in a series of contributions in various conferences of the computer architecture and Parallel Computing fields. We highlight the *Third International Workshop on Parallel Architectures and Bioinspired Algorithms*, Vienna, Austria, 2010, and the *Symposium on Application Accelerators in High*

Performance Computing, Knoxville, Tennessee, USA, 2010. It has been also published in the following publications in ISI journals:

  – 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. *Journal of Logic and Algebraic Programming*, 79 (2010), 317-325. $\boxed{\textbf{JCR 0.552}}$

  – 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 Computing*, 16, 2 (2012), 231-246. $\boxed{\textbf{JCR 1.880}}$

- *Design of simulation algorithms for the modeling framework based on probabilistic P systems.* After the successes with previous simulators, the natural step was to apply the acquired knowledge to the area of computational modeling in population dynamics. First, two new simulation algorithms were designed for PDP systems, so that the probabilistic semantics are captured with more fidelity. The DNDP algorithm introduced a new way to simulate PDP systems: three phases conduct a random distribution of the rules over the available objects in the multisets of a PDP system configuration. Although the behavior was improved, it had not yet met all expectations; for instance, several real ecosystems models were detected to provide large dispersed results. Therefore, a new algorithm was designed, called DCBA, which performs a proportional distribution of objects to rules, what avoids such mentioned dispersion. Both simulation algorithms have been validated towards a real ecosystem model related with the Bearded Vulture in the Pyreneens. It is noteworthy that the development of these algorithms was carried out in collaboration with researchers from the University of Lleida. This work was presented to the scientific community at the international conferences *IEEE Fifth International Conference on Bioinpired Computing: Theories and Applications*, Changsha, China, 2010, and *Thirteenth International Conference on Membrane Computing*, Budapest, Hungary, 2012; and resulted in the following publications in prestigious scientific journals, the first of which indexed in the ISI ranking:

  – M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, F. Sancho-Caparrini. A simulation algorithm for multienvironment probabilistic P systems: A formal verification.

International Journal of Foundations of Computer Science, 22, 1 (2011), 107-118. $\boxed{\textbf{JCR 0.379}}$

– 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, *Lecture Notes in Computer Science*, 7762 (2013), 257-276.

- *Development of high performance simulators for the PDP systems modeling framework.* After experimentally validating the design simulation algorithms (DNDP and DCBA) using the pLinguaCore library, the problem concerning the required simulation resources (CPU time and memory space) arises. Thus, the next natural step was to implement efficient simulators for the last defined algorithm, DCBA. First, an approximation in C++ was developed, which helped to substantially improve performance. Next, parallel versions were implemented using technologies such as multicore processors with OpenMP (obtaining an acceleration of 2.5x) and GPUs with CUDA (providing acceleration 7x). Concerning the CUDA implementation, a new binomial random-variate generator on the GPU was required. To do this, we have developed the cuRNG_BINOMIAL library, which is based on the cuRAND library. The solution is based on the binomial approximation with the normal distribution, and a binomial algorithm, called BINV, for small values. This work was carried out in collaboration with the HPC-Lab group of the NTNU (Norwegian University of Science and Technology), involving a stay of three months during the summer of 2011. The produced works were presented at the international conference *Tenth Conference on Computational Methods in Systems Biology*, London, UK, 2012; and the *First International Conference on Developments in Membrane Computing*, Sevilla, Spain, 2012, resulting in the following publication:

  – M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P Systems on CUDA. *Lecture Notes in Bioinformatics*, 7605 (2012), 247-266.

Part I

Preliminaries

"While lying there admiring this amazing enzyme, I was struck by its similarity to something described in 1936 by Alan Turing, the famous British mathematician."

Leonard Adleman

1 Bioinspired and Natural Computing

The human being has always had a sublime aspiration: a constant improvement of life quality. Since the beginning of time, many questions have raised to be faced by trying to find answers. However, the study of these problems has led, in a natural way, to search for systematic procedures, which permit him to obtain solutions by performing a number of properly sequenced elementary tasks. This mechanization process of solutions facilitated the transmission of knowledge and science learning in general, while opening the possibility of using devices capable of performing these tasks and assist the human being in the general process of troubleshooting.

The advent of electronic computers in the mid last century represented a qualitative advance in the resolution of specific problems that, until then, had been intractable. In 1983, the euphoria produced by this great achievement was braked when R. Churchhouse established the physical limitations of the calculation speed in conventional electronic processors. He demonstrated that there is an upper bound, under the principles of physics, for the speed and the size that the microprocessor can reach. This bound would also impede current techniques to solve problems that are considered intractable, because they require a very *high* amount of time (and/or space) to solve relatively very *large* instances. However much we accelerate the microprocessors (even reaching this physical bound), there would be still having relevant instances of these problems taking years or centuries to be solved in these super-machines.

Unfortunately, the scenario became considerably worst, because this limit was found to be clearly insufficient to attack the resolution of important real life problems. Specifically, it was demonstrated that such problems could never be solved by machines which were on electronic hardware, unless a *miracle* would happen, expressed as a refutation of the famous conjecture $\mathbf{P} \neq \mathbf{NP}$, which is today, one of the most important open problems of science (for more details, see [54]).

Given that circumstance, the human being faces the possibility of designing new devices which respond to other ways of performing calculations, and their hardware is different, so they can get to overcome the inherent barrier of electronic devices that from now, we call *conventional.* In that scenario, the living Nature arises as an inspiration source, so giving the origins of the *Natural Computing* discipline. Its purpose is the analysis of models and computational techniques inspired by Nature, and to understand the world around us in terms of processing information, starting on the basis that a series of dynamic processes that are likely to be interpreted as calculation procedures has been producing in Nature from billions of years ago. The *Zuse-Fredkin thesis* (1960) states that the universe is a cellular automata, even claiming that the information is more important than the matter and energy. The *S. Lloyd thesis* (2006) states that the universe is a quantum computer.

The aim of this chapter is to analyze the above described problematic by the historical context, and introducing properly illustrated basic concepts. In the next section, Natural Computing is presented as the discipline to focus on for the general framework in which this work is developed, namely Membrane Computing. Sections 1.3 and 1.4 are devoted to the description of two of the most common variants of this branch: the devices working like cells, and those working like tissues. The purpose of every computing model is the resolution of problems testing. Therefore, the recognizer membrane systems are introduced, inasmuch as they are specifically designed for this aim. In Section 1.6, the `SAT` problem of Propositional Logic satisfiability is presented, providing two solutions through families of P Systems with active membranes and tissue P systems with cell division, respectively. The chapter ends with a description of some of the might called practical applications of Membrane Computing.

1.1 Natural Computing

Electronic machines are physical devices and, therefore, they have limitations for both the speed of calculation and the miniaturization of the physical components that comprise it.

In the late fifties of the past century, the Nobel-prize winning R.P. Feynman [65] described *sub–microscopic* computers as a revolutionary alternative in the race for the miniaturization of physical components in conventional computers (silicon based circuits). He proposed the computation at the *molecular level* as a possible model in which to implement such operations. Thus, the molecular complexes are considered as virtual components of a information processing device. In 1987, T. Head [82] proposed the first abstract computational model based on the manipulation of the DNA molecules: the *splicing model*. In this model, the information is stored in characters strings like in the DNA molecules. The operations to be performed on these strings are similar to the operations of certain specific enzymes on DNA. In November 1994, L. Adleman [20] carried out the first experiment in a laboratory that solves a specific instance of a **NP**-complete problem via the manipulation of DNA molecules. Feynman's ideas are particularly important since 1983, when R. Churchhouse demonstrates the existence of a limitation on the calculation speed of electronic machines, and in the size of microprocessors. Feynman stated that *one day we would be able to write the entire Encyclopedia Britannica on the head of a pin*.

Many interesting problems that can be solved by means of algorithms of a certain model require a high cost for their resolution, in time and/or space. It is common that for the attempt of reducing one of the two measurements results in a exponential growth of the other. Thus, it is necessary to find new models capable of reducing both parameters; or, at least, include methods in which a high cost of the measures is *absorbed*, in some way, by the model itself in benefit of a substantial reduction over the other.

In this context, the search for new alternative models of computation is necessary. This leads to the design of new machines that can overcome, someday, the barrier followed by the result of R. Churchhouse, while representing a quantitative improvement of the results provided by the *Theory of Computational Complexity*. Moreover, in recent decades, such search has led to the introduction of new models of computation that substantially differ from more classical or conventional ones (Turing machines, recursive functions, λ-calculus, etc.). They provide an important improvement in the quantification of computational resources as part of a possible practical implementation.

Natural Computing is a discipline whose aim is the study and implementation of the dynamic processes that occur in the living Nature and that are likely to be interpreted as calculation procedures. In other words, it tries to capture how Nature acts/operates over the matter, and how it has been *calculating* from billions of years ago. This new research field arises as one alternative for more classical computation models, regarding the seek of new paradigms that can provide an *effective* solution to the limitations of conventional models. Currently, within the field of Natural Computing, there are a wide range of models, some of them extends this concept to include *quantum computing*, that is not closely based on the above interpretation. These models study how several laws of nature produce changes in certain systems (from habitats to sets of molecules, and living organisms) which may be interpreted as calculation processes on its elements. This simulation addressed by Natural Computing can have different interpretations when describing the new models: to be used for the design of new algorithmic schemes using techniques inspired by nature; or to suggest the physical creation of new experimental models. In the latter, the electronic media of conventional computers is replaced with other substrate that can implement certain processes shown in the operating mode of nature.

Living Nature has been a computational inspiration source since the mid-forties of the last century. In 1943, W.S. McCulloch and W. Pitts [109] proposed the first model of artificial neural networks, inspired by the interconnections and functioning of neurons in the brain. They are the example for subsequent models of M. Minsky [110] and F. Rosenblatt [145], among others. In 1975, J. Holland introduced the genetic algorithm, a computational model inspired by the processes of evolution and natural selection of living beings. These methods aims to find a *good* solution from a large number of possible candidate solutions.

Natural Computation was born, in fact, as a discipline at the end of 1994 after the experiment of L. Adleman. Adleman solved a small instance of a *presumably intractable* problem from a computational viewpoint: any known mechanical solution requires a number of resources that is exponential in the size of the input data. One of the basic objectives of *molecular computing* is the usage of organic molecules (DNA, RNA, proteins, etc.) as biological hardware that allows to perform computations.

Membrane Computing was introduced by Gh. Păun [140] in 1998, and it is inspired by the structure and functioning of the cells of living organisms in their ability to process and generate information. In October 2003, the Institute for Scientific Information (ISI, USA), has designated *Membrane Computing* as *Fast Emerging Research Front* in the area of *Computer Science*.

Before going further, it is convenient to distinguish two concepts related to computing models: implementation and simulation. Recall that any computing model consists of a syntax specification, and its dynamics is governed by a formal semantics. When we talk about the implementation of such a model, we are referring to the construction of a physical device (a machine) whose behavior is identical to the model itself, in the following sense: once it receives an input, the machine performs each transition step of the model in a computation step of the machine (in a unit of time) so that in both cases the same results are obtained.

In this thesis, we use the term *simulator* of a formal model to a software/hardware computer application, capable of being executed on an electronic computer, which describes the specification through a programming language and captures the semantics through a simulation algorithm. This simulation algorithm must faithfully reproduce the dynamics, i.e., each computing step of the formal model is reproduced in the simulator through a finite number of steps (greater than 1), so that the simulator is capable of determining the basic elements of the model which intervened in a relevant way in that step.

Genetic algorithms and artificial neural networks have been implemented through programs running on electronic conventional computers. DNA-based Molecular Computing has been implemented by biochemical means: L. Adleman's experiment allowed to solve a particular instance of the Hamiltonian path problem, in its directed and two with distinguished nodes, by the manipulation of DNA molecules in the laboratory. On the contrary, Membrane Computing has not yet been implemented neither electronically nor biochemically.

The next section is devoted to present the most relevant aspects of Membrane Computing, the framework where this thesis is focused on.

1.2 Membrane Computing

At the end of 1998, Gheorghe Păun [140] introduced a new natural computing paradigm, called *Membrane Computing*, inspired by the structure and the functioning of the cells of living organisms. In the original ideas of Gh. Păun, membrane systems provide distributed parallel and non-deterministic devices, and they were not properly introduced to fully model the structure and functioning of a cell, but to analyze some computationally relevant facts that can be abstracted from them.

Cells are the basic unit of every living organism. They have a complex, but very organized, structure that allows the simultaneous execution of chemical reactions. There are two types of cells: *prokaryotes* (typical of certain unicellular organisms, such as bacteria and cyanobacteria), that lack of a nuclear membrane, and therefore, have no separation between nucleus and cytoplasm; and *eukaryotes* (typical of animals and plants) whose nucleus is separated from the cytoplasm by a double membrane. In both cell types, a series of processes that are essential to life takes place in a similar way.

In a first analysis, three distinct parts can be differentiated in a cell (see Figure 1.1): a kind of very thin film (*plasma membrane*) which delimits the cell from its environment; a central corpuscle (*nucleus*), which contains and stores the genetic information in molecules of DNA; and *cytoplasm*, which is the part between the core and the plasma membrane. In the cytoplasm of eukaryotic cells there are different fundamental components (see Figure 1.1): the *mitochondria*, which is responsible for the generation of molecules enclosing useful energy in metabolic processes; the *Golgi apparatus*, that is responsible for intracellular transportation of several substances; *ribosomes*, which are a protein factory and play an essential role in cellular metabolism; the *endoplasmic reticulum*, which is a network of interconnected membranes structured in two parts: one that is part of the nuclear membrane and facilitates the passage of the messenger RNA from the nucleus into the cytoplasm, and other that handles the communication between the various cell components; and finally, the *lysosomes*, which are vesicles surrounded by a single membrane containing enzymes, and responsible for digesting substances which come from outside, as well as they degrade the residual components that are no longer useful for the cell.

A striking feature about the internal cell structure is that the parts of the biological system are delimited by several types of *membranes* (in its broadest meaning). They range from the outer membrane separating the internal and external parts of the cell, to the several membranes which delimit the internal vesicles. Furthermore, and regarding the functionality of these membranes in nature, it is noteworthy that the generated compartments are not watertight, but they allow the sometimes selective, or even only the one-directional, passage (flow) of certain chemical compounds selectively. *Biological membranes* are dynamic basic structures for the cell, and play an essential role in defining the phenomenon usually called *life*. An entity is a living organism if it can autonomously perform the following tasks: (a) replicate DNA; (b) energy production; (c) synthesize proteins; and (d) perform metabolic processes.

Because the processes that occur in a cell are highly complex, it is impossible to completely model them. A computing model that attempts to *literally* simulate such processes would not be practical, except from a biological point of view. It aims to create an abstract computing model that simulates, in a

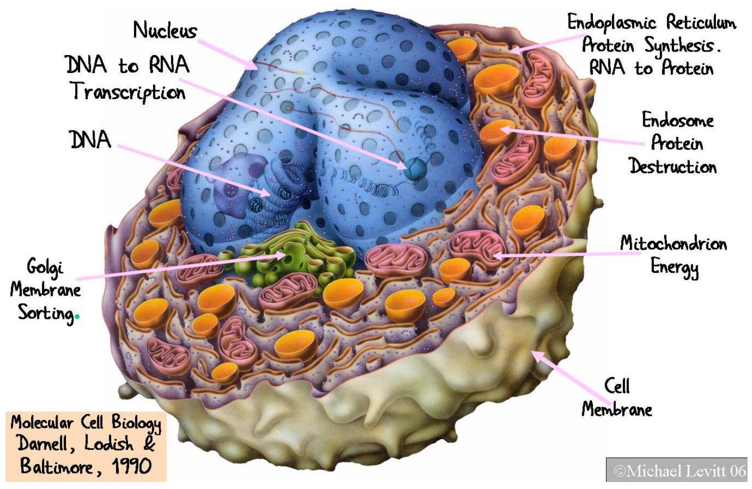

Nucleus
DNA to RNA Transcription
DNA
Golgi Membrane Sorting.
Endoplasmic Reticulum Protein Synthesis. RNA to Protein
Endosome Protein Destruction
Mitochondrion Energy
Cell Membrane
Molecular Cell Biology Darnell, Lodish & Baltimore, 1990
©Michael Levitt 06

Figure 1.1: The eukaryotic cell

simplified form but as closely as possible, the behavior of cells. This allows us (at least in principle) to obtain alternative solutions to computationally intractable problems from a conventional viewpoint. To do this, it is necessary to bring out all those behavioral characteristics and constitutions of the cell. They may be useful for the development of a computing model that have to be both powerful (in terms of the problems they can solve) and simple (concerning the definition, implementation and execution).

In summary, the behavior of a cell can be considered as a machine that performs a calculation process: a non trivial machine, from the biological point of view, in which occurs a flow and alteration of chemical substances, as the cell itself processes, by means of a hierarchical distribution of inner membranes.

1.2.1 Preliminaries

In this section we introduce some concepts and notations which we will use throughout this thesis.

An *alphabet*, Γ, is a non–empty set whose elements are called *symbols*. An ordered finite sequence of symbols is a *string* or *word*. The set of all strings over an alphabet Γ is denoted by Γ^*. A *language* over Σ is a subset of Γ^*.

A *multiset* m over an alphabet Γ is a pair (Γ, f) where $f : \Gamma \to \mathbb{N}$ is a mapping. If $m = (\Gamma, f)$ is a multiset then its *support* is defined as $supp(m) =$

$\{x \in \Gamma \,|\, f(x) > 0\}$. A multiset is finite if its support is a finite set. If $m = (\Gamma, f)$ is a finite multiset over Γ, and $supp(m) = \{a_1, \ldots, a_k\}$ then it will be denoted as $m = a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ (here the order is irrelevant), and we say that $f(a_1) + \cdots + f(a_k)$ is the cardinal of m, denoted by $|m|$. The empty multiset is denoted by $\emptyset$. We also denote by $M_f(\Gamma)$ the set of all finite multisets over Γ.

If $m_1 = (A, f_1)$, $m_2 = (A, f_2)$ are multisets over A, then we define the union of m_1 and m_2 as $m_1 + m_2 = (A, g)$, where $g = f_1 + f_2$.

1.3 Cell-like P systems

Membrane Computing devices are generically called *P systems*. They constitute a theoretical computing model of a distributed, parallel and non-deterministic type. In Membrane Computing there are basically two ways to consider computational devices: cell–like membrane systems and tissue–like membrane systems. The first one, using the biological membranes arranged hierarchically, inspired from the structure of the cell, and the second one using the biological membranes placed in the nodes of a graph, inspired from the cell inter–communication in tissues.

The main *syntactic* ingredients of a cell–like P system are the *membrane structure*, the *multisets*, and the *evolution rules*.

- A *membrane structure* consists of several membranes arranged in a hierarchical structure (understood as three dimensional vesicles) inside a main membrane (the *skin*), and delimiting *regions* (the space in–between a membrane and the immediately inner membranes, if any). Each membrane identifies a region inside the system. A membrane without any membrane inside is called *elementary*. A membrane structure can be considered as a rooted tree.

- Regions defined by a membrane structure contain objects corresponding to chemical substances present in the compartments of a cell. The objects can be described by symbols or by strings of symbols, in such a way that *multiset of objects* are placed in regions of the membrane structure.

- The objects can evolve according to given *evolution rules*, associated with the regions (hence, with the membranes).

The *semantics* of the cell–like membrane systems is defined through a non deterministic and synchronous model (in the sense that a global clock is as-

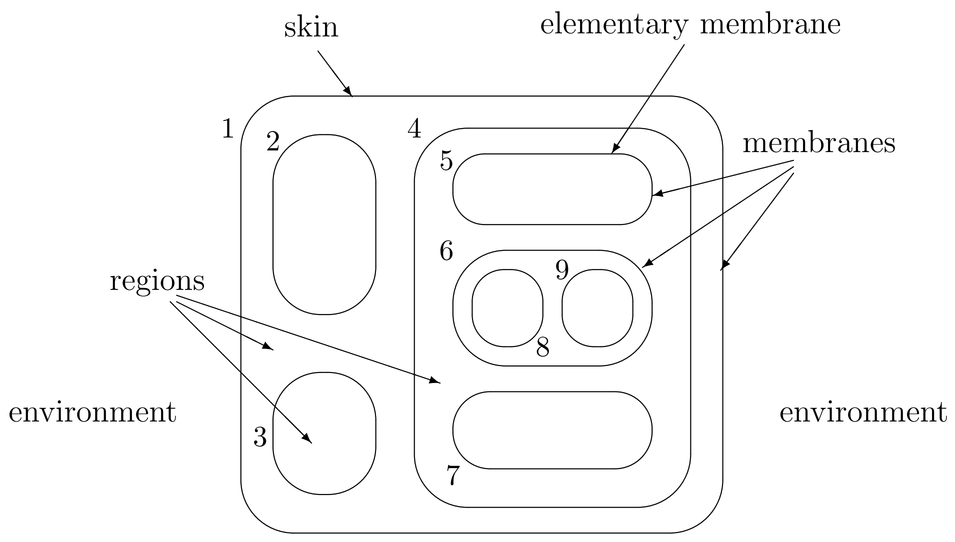


Figure 1.2: A membrane structure

sumed), by introducing the concepts of *configuration*, *transition step*, and *computation*.

Next, let us to describe the basic model of a cell-like P system, introduced by Gh. Păun in the seminal paper [140]. A *basic transition P system* of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:

1. Γ is a finite *alphabet*.

2. $H = \{1, \ldots, q\}$.

3. μ is a rooted tree.

4. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ.

5. $\mathcal{R} = \{R_1, \ldots, R_q\}$, where R_i, $1 \leq i \leq q$, are finite sets of *evolution rules* over Γ of the form $u \to v$ where

   - u is a finite multiset over Γ.
   - $v = (v_1, here)(v_2, out)(v_3, in_j)$ or $v = (v'_1, here)(v'_2, out)(v'_3, in_j)\ \delta$, being $v_1, v'_1, v_2, v'_2, v_3, v'_3$ finite multisets over Γ, and δ is a distinguished symbol such that $\delta \notin \Gamma$.

6. $i_{out} \in H \cup \{0\}$.

A *basic transition P system* $\Pi = (\Gamma, H, \mu, , \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, of degree $q \geq 1$ can be viewed as a set of q membranes (the nodes ot the tree μ), injectively labeled by $1, \dots, q$, with an environment labeled by 0, and such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over Γ representing the objects initially placed in the q membranes of the system; (b) $\mathcal{R}$ is the set of rules that allows to evolve the system (R_i is the set of rules associated with membrane i); and (c) $i_{out} \in \{0, 1, 2, \dots, q\}$ represents a distinguished membrane (when $i_{out} \in \{1, 2, \dots, q\}$), or the environment when $i_{out} = 0$, which will encode the output of the system.

Consider a rule $r \equiv u \to v$ from a set R_i. To apply this rule in region i means to remove the multiset of objects specified by u from region i, and to introduce the objects specified by v, in the regions indicated by the target commands associated with the objects from v. Specifically, if v contains $(a, here)$, then the object a will be placed in the same region i where the rule is applied. If v contains (b, out), then the object b will be moved to the region immediately outside membrane i (this region can be the environment in the case when i is the skin membrane; in this case, the object leaves the system and it never comes back). If v contains (c, in_j), then the object c should be moved in the region j, providing that this membrane is immediately inside membrane i; otherwise, the rule r cannot be applied. If δ appears in v, then membrane i is dissolved, that is, the membrane i is removed and all objects and membranes previously present in it become elements of the contents of the immediately upper membrane that has not been dissolved. The skin membrane is never dissolved.

We define the *cooperation degree* of a rule $u \to v$ as the cardinal of the multiset involved in the left-hand side, $|u|$. If the cooperation degree of a rule is strictly greater than 1, then it is called a *cooperative rule*.

The rules are applied in a non-deterministic maximally parallel manner, that is, the objects to evolve in a step and the rules by which they evolve are chosen in a non-deterministic manner, but in such a way that in each region we have a maximally parallel application of rules: we assign objects to rules, non-deterministically choosing the rules and the objects assigned to each rule, but in such a way that after this assignation no further rule can be applied to the remaining objects. A rule can be applied in the same step as many times as we want, only the number of copies of objects matters.

The *semantics* of the cell–like membrane systems is defined through a non deterministic and synchronous model (in the sense that a global clock is assumed) as follows:

- An *instantaneous description* or a *configuration* at any instant of a basic cell-like P system $\Pi = (\Gamma, H, \mu, , \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, consists of a membrane structure and a family of multisets of objects of objects over Γ associated with each region of the structure. The *initial configuration* is $(\mu, \mathcal{M}_1, \cdots, \mathcal{M}_q)$. A configuration is a *halting configuration* if no rule of the system is applicable to it. In each time unit we can transform a given configuration in another configuration by applying the evolution rules to the objects placed inside the regions of the configurations, in a non–deterministic, and maximally parallel manner (the rules are chosen in a non–deterministic way, and in each region all objects that can evolve must do it). In this way, we get *transitions* from one configuration of the system to the next one

- We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step*, denoted by $\mathcal{C}_1 \Rightarrow_\Pi \mathcal{C}_2$, if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks.

- A *computation* of Π is a (finite or infinite) sequence of configurations such that:

  1. the first term of the sequence is the initial configuration of the system;
  2. each non-initial configuration of the sequence is obtained from the previous configuration by applying rules of the system in a maximally parallel manner with the restrictions previously mentioned; and
  3. if the sequence is finite (called *halting computation*) then the last term of the sequence is a *halting configuration.*

  All computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output region i_{out} in the halting configuration.

  If $\mathcal{C} = \{\mathcal{C}_t\}_{t<r+1}$ of Π ($r \in \mathbb{N}$) is a halting computation, then the *length of* $\mathcal{C}$, denoted by $|\mathcal{C}|$, is r, that is, $|\mathcal{C}|$ is the number of non-initial configurations which appear in the finite sequence $\mathcal{C}$.

We assume that a global clock exists, marking the time for the whole system (for all compartments of the system), and we have here a double parallelism, once at the level of each region (the rules are used in parallel) and once at the level of the system (all regions evolve concomitantly).

1.3.1 P systems with active membranes

One of the explicit goals of various branches of natural computing is to find ways to address computationally hard problems (typically, **NP**-complete problems) in order to solve them in a "feasible" time.

The rules of a basic transition P system are used in parallel. This is a good level of parallelism, which, however, is not sufficient to devise polynomial time solutions to **NP**-complete problems (unless $\mathbf{P} = \mathbf{NP}$, which is not at all plausible); the proof of this result can be found in [155]. However, biology suggests operations with membranes such *membrane division* which, sometimes surprisingly, make possible polynomial (often linear) solutions to **NP**-complete problems. Membrane division brings a further level of parallelism, making possible to construct an exponential workspace expressed in terms of the number of membranes and the number of objects, in polynomial time.

P systems with active membranes having associated electrical charges with membranes were first introduced by Gh. Păun [139].

Definition 1.1. *A P* system with active membranes *of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:*

1. *Γ is a finite alphabet.*

2. *$H = \{1, \ldots, q\}$.*

3. *μ is a rooted tree.*

4. *$\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ.*

5. *$\mathcal{R}$ is a finite set of rules, of the following forms:*

   (a) $[\, a \to u \,]_h^{\alpha}$*, for $h \in H$,$\alpha \in \{+, -, 0\}$, $a \in \Gamma$, $u \in \Gamma^*$* (object evolution rules)*.*

   (b) $a [\ \ ]_h^{\alpha_1} \to [\, b \,]_h^{\alpha_2}$*, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$* (send–in communication rules)*.*

   (c) $[\, a \,]_h^{\alpha_1} \to [\ \ ]_h^{\alpha_2}\, b$*, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$* (send–out communication rules)*.*

   (d) $[\, a \,]_h^{\alpha} \to b$*, for $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in \Gamma$* (dissolution rules)*.*

   (e) $[\, a \,]_h^{\alpha_1} \to [\, b \,]_h^{\alpha_2}\ [\, c \,]_h^{\alpha_3}$*, for $h \in H$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $a, b, c \in \Gamma$* (division rules for elementary membranes)*.*

6. *$i_{out} \in H \cup \{0\}$.*

A *P system with active membrane* $\Pi = (\Gamma, H, \mu, , \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, of degree $q \geq 1$ can be viewed as as a set of q membranes (the nodes ot the tree μ) injectively labeled with elements of H, with electrical charges $(+, -, 0)$ associated with them, and with an environment labeled by 0 such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over Γ representing the objects initially placed in the q membranes of the system; (b) $\mathcal{R}$ is the set of rules that allows to evolve the system; and (c) $i_{out} \in \{0, 1, 2, \dots, q\}$ represents a distinguished membrane when $i_{out} \in \{1, 2, \dots, q\}$, or the environment when $i_{out} = 0$, which will encode the output of the system.

P systems with active membranes differ from the basic transition P systems on the type of rules. The evolution rule $[\, a \to u \,]_h^{\alpha}$ is associated with membrane h and depending on the label and the charge of the membrane, but not directly involving the membrane, in the sense that the membrane is neither taking part in the application of this rule nor it is modified by the rule. In reaction with an object a, a finite multiset v is produced in that membrane.

When applying a send-in rule $a\, [\ ]_h^{\alpha_1} \to [\, b \,]_h^{\alpha_2}$, in reaction with object a in the parent membrane, an object b is introduced in the membrane h, and the polarization of the membrane can be modified, but not its label. When applying a send-out rule $[\, a \,]_h^{\alpha_1} \to [\ ]_h^{\alpha_2}\, b$, in reaction with object a in membrane h, an object b is sent out of the membrane, and the polarization of the membrane can be modified, but not its label.

When applying a dissolution rule $[\, a \,]_h^{\alpha} \to b$, under the influence of object a, the membrane h is dissolved, while the object a is transformed in object b.

When applying a division rule $[\, a \,]_h^{\alpha_1} \to [\, b \,]_h^{\alpha_2}\ [\, c \,]_h^{\alpha_3}$ for an elementary membrane h, in reaction with object a, membrane h is divided into two membranes with the same label, possibly of different polarizations; object a is replaced in the two new membranes by objects b and c, respectively.

These rules are applied according to the following principles ([139]):

- All the rules are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by only one rule (chosen in a non deterministic way), but any object which can evolve by one rule of any form, must evolve.

- If a membrane is dissolved, its content (multiset and internal membranes) is left free in the surrounding region.

- If at the same time a membrane labeled by h is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then we suppose that first the evolution rules of type

(a) are used, and then the division is produced. Of course, this process takes only one step.

- The rules associated with membranes labeled by h are used for all copies of this membrane. At one step, a membrane can be the subject of *only one* rule of types (b)-(e).

Note that these P systems have some important features: (a) they are non-cooperative systems (there is no cooperative rule); (b) they use three electrical charges; (c) the polarization of a membrane, but not the label, can be modified by the application of a rule; and (d) they do not use cooperation (the left-hand side of the rules consist of only one symbol).

1.4 Tissue-like P systems

In this section we consider computational devices inspired from the cell inter–communication in tissues, and adding the ingredient of cell division rules of the same form as in cell–like membrane systems with active membranes, but without using polarizations. In these systems, the rules are used in the non-deterministic maximally parallel way, as usual, but we suppose that when a cell is divided, its interaction with other cells or with the environment is blocked; that is, if a division rule is used for dividing a cell, then this cell does not participate in any other rule, for division or communication. The set of communication rules implicitly provides the graph associated with the system through the labels of the membranes. The cells obtained by division have the same labels as the mother cell, hence the rules to be used for evolving them or their objects are inherited.

Definition 1.2. *A tissue P system with symport/antiport rules of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:*

1. *Γ is a finite alphabet.*

2. *$\mathcal{E} \subseteq \Gamma$.*

3. *$\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over Γ.*

4. *$\mathcal{R}$ is a finite set of communication rules of the form $(i, u/v, j)$, for $i, j \in \{0, 1, 2, \dots, q\}, i \neq j$, $u, v \in \Gamma^*$, $|uv| > 0$.*

5. *$i_{out} \in \{0, 1, 2, \dots, q\}$.*

A *tissue P system with symport/antiport rules*

$$\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$$

of degree $q \geq 1$ can be viewed as a set of q cells, labeled by $1, \ldots, q$, with an environment labeled by 0 such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ representing the objects (elements in Γ) initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects located initially in the environment of the system, all of them appearing in an *arbitrary number of copies*; and (c) $i_{out} \in \{0, 1, \ldots, q\}$ represents a distinguished cell when $i_{out} \in \{1, \ldots, q\}$, or the environment when $i_{out} = 0$, which will encode the output of the system.

When applying a rule $(i, u/v, j)$, the objects of the multiset represented by u are sent from region i to region j and, simultaneously, the objects of multiset v are sent from region j to region i. The length of the communication rule $(i, u/v, j)$ is defined as $|u| + |v|$, that is, the total number of objects which appear in the rule.

A communication rule $(i, u/v, j)$ is called a *symport rule* if $u = \lambda$ or $v = \lambda$. A symport rule $(i, u/\lambda, j)$, with $i \neq 0, j \neq 0$, provides a virtual arc from cell i to cell j. A communication rule $(i, u/v, j)$ is called an *antiport rule* if $u \neq \lambda$ and $v \neq \lambda$. An antiport rule $(i, u/v, j)$, with $i \neq 0, j \neq 0$, provides two arcs: one from cell i to cell j and another one from cell j to cell i. Thus, every tissue P systems has an underlying directed graph whose nodes are the cells of the system and the arcs are obtained from communication rules. In this context, the environment can be considered as a virtual node of the graph such that their connections are defined by communication rules of the form $(i, u/v, j)$, with $i = 0$ or $j = 0$.

The rules of a system like the one above are used in a non-deterministic maximally parallel manner as it is customary in Membrane Computing. At each step, all cells which can evolve must evolve in a maximally parallel way (at each step we apply a multiset of rules which is maximal, no further applicable rule can be added).

An *instantaneous description* or a *configuration* at any instant of a tissue P system is described by all multisets of objects over Γ associated with all the cells present in the system, and the multiset of objects over $\Gamma - \mathcal{E}$ associated with the environment at that moment. Bearing in mind that the objects from $\mathcal{E}$ have infinite copies in the environment, they are not properly changed along the computation. The *initial configuration* is $(\mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$. A configuration is a *halting configuration* if no rule of the system is applicable to it.

Let us fix a tissue P system with symport/antiport rules Π. We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step*, denoted $\mathcal{C}_1 \Rightarrow_{\Pi}$

$\mathcal{C}_2$, if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks. A *computation* of Π is a (finite or infinite) sequence of configurations such that:

1. the first term of the sequence is an initial configuration of the system;
2. each non-initial configuration of the sequence is obtained from the previous configuration by applying the rules of the system in a maximally parallel manner with the restrictions previously mentioned; and
3. if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration.

All computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output region (a cell or the environment) i_{out} in the halting configuration.

1.4.1 Tissue P Systems with Cell Division

Cell division is an elegant process that enables organisms to grow and reproduce. Mitosis is a process of cell division which results in the production of two daughter cells from a single parent cell. Daughter cells are identical to one another and to the original parent cell. Through a sequence of steps, the replicated genetic material in a parent cell is equally distributed to two daughter cells. While there are some subtle differences, mitosis is remarkably similar across organisms.

Before a dividing cell enters mitosis, it undergoes a period of growth where the cell replicates its genetic material and organelles. Replication is one of the most important functions of a cell. DNA replication is a simple and precise process that creates two complete strands of DNA (one for each daughter cell) where only one existed before (from the parent cell).

Let us recall that the model of *tissue P systems with cell division* is based on the cell-like model of P systems with active membranes [139]. In these models, the cells are not polarized; the cells obtained by division have the same labels as the original cell, and if a cell is divided, its interaction with other cells or with the environment is locked during the division process. In some sense, this means that while a cell is dividing it closes its communication channels.

Definition 1.3. *A tissue P system with cell division of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:*

1. *Γ is a finite alphabet.*
2. *$\mathcal{E} \subseteq \Gamma$.*
3. *$\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over Γ.*
4. *$\mathcal{R}$ is a finite set of rules of the following forms:*
   - *(a)* Communication rules*: $(i, u/v, j)$, for $i, j \in \{0, 1, 2, \dots, q\}, i \neq j$, $u, v \in \Gamma^*$, $|u \cdot v| \neq 0$;*
   - *(b)* Division rules*: $[a]_i \to [b]_i[c]_i$, where $i \in \{1, 2, \dots, q\}$, $i \neq i_{out}$ and $a, b, c \in \Gamma$.*
5. *$i_{out} \in \{0, 1, 2, \dots, q\}$.*

A *tissue P system with cell division* is a *tissue P system with symport/antiport rules* where division rules of cells are allowed.

When applying a division rule $[a]_i \to [b]_i[c]_i$, under the influence of object a, the cell with label i is divided into two cells with the same label; in the first copy, object a is replaced by object b, in the second one, object a is replaced by object c; all the other objects are replicated and copies of them are placed in the two new cells. The output cell i_{out} cannot be divided.

The rules of a tissue P system with cell division are applied in a non-deterministic maximally parallel manner as it is customary in membrane computing. At each step, all cells which can evolve must evolve in a maximally parallel way (at each step we apply a multiset of rules which is maximal, no further rule can be added), with the following important remark: if a cell divides, only the division rule is applied to that cell at that step; the objects inside that cell do not evolve by means of communication rules. In other words, we can think that before division a cell interrupts all its communication channels with the other cells and with the environment. The new cells resulting from division will only interact with other cells or with the environment at the next step – providing they do not divide once again. The label of a cell identifies the rules which can be applied to it precisely.

1.5 Recognizer membrane systems

Throughout this chapter we use the term *membrane system* to refer to both a cell-like P system or a tissue-like P system. In both cases we can describe them by $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where in the case of cell-like P system, the alphabet of the environment $\mathcal{E}$ can be considered as the empty set.

Usually, computational complexity theory deals with decisions problems which are problems that require a "*yes*" or "*no*" answer. A *decision problem*, X, is a pair (I_X, θ_X) such that I_X is a language over a finite alphabet (whose elements are called *instances*) and θ_X is a total boolean function (that is, a predicate) over I_X. Of course, many abstract problems are not decision problems. For example, in combinatorial optimization problems some value must be optimized (minimized or maximized). In order to deal with such problems, they can be transformed into roughly equivalent decision problems by supplying a target/threshold value for the quantity to be optimized, and then asking whether this value can be attained.

A natural correspondence between decision problems and languages can be established as follows. Given a decision problem $X = (I_X, \theta_X)$, its associated language is $L_X = \{w \in I_X |\ \theta_X(w) = 1\}$. Conversely, given a language L, over an alphabet Σ, its associated decision problem is $X_L = (I_{X_L}, \theta_{X_L})$, where $I_{X_L} = \Sigma^*$, and $\theta_{X_L} = \{(x,1)|\ x \in L\} \cup \{(x,0)|\ x \notin L\}$.

The solvability of decision problems is defined through the recognition of the languages associated with them. Let M be a Turing machine with working alphabet Γ and L a language over Γ. Assume that the result of any halting computation of M is *yes* or *no*. If M is a *deterministic* device then we say that M *recognizes* or *decides* L whenever, for any string u over Γ, if $u \in L$, then the answer of M on input u is *yes* (that is, M accepts u), and the answer is *no* otherwise (that is, M rejects u). If M is a *non-deterministic* device, then we say that M *recognizes* or *decides* L if for any string u over Γ, $u \in L$ if and only if there exists a computation of M with input u such that the answer is *yes*.

Throughout this chapter, it is assumed that each abstract problem has an associated fixed *reasonable encoding scheme* that describes the instances of the problem by means of strings over a finite alphabet. We do not define *reasonable* in a formal way, however, following [72], instances should be encoded in a concise manner, without irrelevant information, and where relevant numbers are represented in binary (or any fixed base other than 1). It is possible to use multiple reasonable encoding schemes to represent instances, but it is proved that the input sizes differ at most by a polynomial. The *size* $|u|$ of an instance u is the length of the string associated with it, in some reasonable encoding scheme.

In order to study the computational efficiency of membrane systems, the notions from classical *computational complexity theory* are adapted for Membrane Computing, and a special class of cell-like P systems is introduced in [134]: *recognizer P systems* (called *accepting P systems* in a previous paper

[133]).

Definition 1.4. *A* recognizer membrane system *of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$, where:*

1. *$\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$ is a membrane system.*
2. *The working alphabet Γ has two distinguished objects* `yes` *and* `no` *being, at least, one copy of them present in $\mathcal{M}_1 \cup \dots \cup \mathcal{M}_q$, but, in the case of a tissue-like P system, none of them are present in $\mathcal{E}$.*
3. *Σ is a finite alphabet strictly contained in Γ, and such that $\mathcal{E} \cap \Sigma = \emptyset$. It is is called the input alphabet.*
4. *$\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over $\Gamma \setminus \Sigma$.*
5. *$i_{in} \in \{1, \dots, q\}$ is the input membrane/cell.*
6. *$i_{out} = 0$ is the label of the environment that represents the output region.*
7. *All computations halt.*
8. *If $\mathcal{C}$ is a computation of Π, then either object* `yes` *or object* `no` *(but not both) must have been released into the environment, and only at the last step of the computation.*

A *recognizer membrane system* of degree $q \geq 1$ is a tuple

$$\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$$

can be viewed as a membrane system such that has an input alphabet Σ and an input region i_{in}. The initial multisets of the system are multisets over $\Gamma \setminus \Sigma$.

For each multiset m over Σ, the *computation of the system* Π *with input* m starts from the configuration of the form $(\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_{i_{in}} + m, \dots, \mathcal{M}_q; \emptyset)$, that is, the input multiset m has been added to the contents of the input region i_{in}. Therefore, we have an initial configuration associated with each input multiset m (over the input alphabet Σ) in this kind of systems.

Given a recognizer membrane system Π, and a halting computation $\mathcal{C} = \{\mathcal{C}_i\}_{i<r+1}$ of Π ($r \in \mathbf{N}$), the result of $\mathcal{C}$ is `yes` (respectively, `no`) if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$, and neither object `yes` nor `no` appears in the environment associated with any non–halting configuration of $\mathcal{C}$. If the result of a computation $\mathcal{C}$ is `yes` (respectively, object `no`), then we say that We say that a computation $\mathcal{C}$ is an *accepting computation* (respectively, *rejecting computation*) if the result of $\mathcal{C}$ is `yes` (respectively, `no`).

1.5.1 Polynomial complexity classes of membrane systems

Now, we define what it means to solve a decision problem in the framework of membrane systems efficiently and in a uniform way. Since we define each membrane system to work on a finite number of inputs, to solve a decision problem we define a numerable family of membrane systems.

Definition 1.5. *We say that a decision problem* $X = (I_X, \theta_X)$ *is solvable in a uniform way and polynomial time by a family* $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ *of recognizer membrane systems if the following holds:*

1. *The family* $\mathbf{\Pi}$ *is* polynomially uniform *by Turing machines, that is, there exists a deterministic Turing machine working in polynomial time which constructs the system* $\Pi(n)$ *from* $n \in \mathbb{N}$.

2. *There exists a pair* (cod, s) *of polynomial-time computable functions over* I_X *such that:*

   (a) *for each instance* $u \in I_X$, $s(u)$ *is a natural number and* $cod(u)$ *is an input multiset of the system* $\Pi(s(u))$;

   (b) *for each* $n \in \mathbb{N}$, $s^{-1}(n)$ *is a finite set;*

   (c) *the family* $\mathbf{\Pi}$ *is* polynomially bounded *with regard to* (X, cod, s), *that is, there exists a polynomial function* p, *such that for each* $u \in I_X$ *every computation of* $\Pi(s(u))$ *with input* $cod(u)$ *is halting and it performs at most* $p(|u|)$ *steps;*

   (d) *the family* $\mathbf{\Pi}$ *is* sound *with regard to* (X, cod, s), *that is, for each* $u \in I_X$, *if* $\underline{\textit{there exists}}$ *an accepting computation of* $\Pi(s(u))$ *with input* $cod(u)$, *then* $\theta_X(u) = 1$;

   (e) *the family* $\mathbf{\Pi}$ *is* complete *with regard to* (X, cod, s), *that is, for each* $u \in I_X$, *if* $\theta_X(u) = 1$, *then* $\underline{\textit{every}}$ *computation of* $\Pi(s(u))$ *with input* $cod(u)$ *is an accepting one.*

From the soundness and completeness conditions above we deduce that every membrane system $\Pi(n)$ is *confluent*, in the following sense: every computation of a system with the *same* input multiset must always give the *same* answer.

Let $\mathbf{R}$ be a class of recognizer tissue P systems. We denote by $\mathbf{PMC_R}$ the set of all decision problems which can be solved in a uniform way and polynomial time by means of families of systems from **R**. The class $\mathbf{PMC_R}$ is closed under complement and polynomial–time reductions [133].

Different polynomial time solutions for **NP**–complete problems have been obtained using this class of cell–like recognizer membrane systems: *Knapsack* ([129]), *Subset Sum* ([130]), *Partition* ([78]), *SAT* ([133]), *Clique* ([21]), *Bin Packing* ([131]), and *CAP* ([132]).

In the framework of P systems without input membrane, C. Zandron, C. Ferretti and G. Mauri [155] proved that confluent recognizer P systems with active membranes making use of no membrane division rule, can be efficiently simulated by a deterministic Turing machine.

1.6 P system based solutions to the SAT problem

In this section, we provide two efficient solutions to the `SAT` problem of satisfiability of propositional logic. One is given by a family of P systems with active membranes, and the other one by a family of tissue P systems with cell division, according to the Definition 1.5.

First, we will describe the `SAT` problem of satisfiability of the propositional logic. Recall that this problem was the first to be demonstrated to be **NP-complete** (S. Cook, 1971). Thus it begun itself the theory of computational complexity.

The language of propositional logic consists of: (a) a countably infinite set, VP, of propositional variables x_i, (b) two logical connectives: negation ($\neg$) and disjunction ($\vee$), and (c) some auxiliary parenthesis symbols: (and).
The set $PForm$ of *propositional formulas* is the smallest set, $\mathcal{F}$, which contains VP and verifies the following conditions: (a) if $\varphi \in \mathcal{F}$, then $\neg\varphi \in \mathcal{F}$, and (b) if $\varphi, \psi \in \mathcal{F}$, then $(\varphi \vee \psi) \in \mathcal{F}$.

From the logical connectives of negation $\neg$ and disjunction $\vee$, the logical connectives of conjunction $\wedge$, implication $\rightarrow$ and double implication $\leftrightarrow$ are defined as follows:

- $\varphi \wedge \psi \equiv \neg((\neg\varphi) \vee (\neg\psi))$.
- $\varphi \rightarrow \psi \equiv (\neg\varphi) \vee \psi$.
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

Usually, we note $\neg\varphi$ as $\overline{\varphi}$,$\varphi \vee \psi$ as $\varphi + \psi$, and $\varphi \wedge \psi$ as $\varphi \cdot \psi$.

A *literal* is a propositional variable or the negation of a propositional variable, and a *clause* is a disjunction of a finite number of literals. We say that a propositional formula is in *conjunctive normal form* (**CNF**) if it is the conjunction of a finite number of clauses, that is, whether the formula is a finite conjunction of a finite disjunction of literals. We assume that any propositional formula in **CNF** is in a *simplified form*, i.e., in each clause appears, at most, a literal corresponding to each one of the variables. For example, it can not appear, in a clause, the same variable x_i twice, neither may appear simultaneously a variable x_i and its negation $\neg x_i$.

A *truth assignment* (or *truth valuation*) is an application of the set of propositional variables, VP, in $\{0, 1\}$, that is, it assigns a boolean value (1, true, 0, false) to each propositional variable. Every truth assignment is naturally extended to an application of the propositional formulas set $PForm$ in $\{0, 1\}$, through *truth tables*. We say that a propositional formula, φ is *true* (respectively, *false*) by a truth assignment if it is assigned the 1 value, true (respectively, the value 0, false) to such formula.

Note that an truth assignment is an application whose domain is an infinite set, the set of propositional variables VP. However, any propositional formula contains a finite number of propositional variables. Therefore, when calculating the value of a formula by a truth assignment, it will be sufficient to know the values of such assignment to the variables appearing in the formula. Thus, given a propositional formula, a *relevant valuation* is the restriction of a truth assignment to the set of propositional variables in such formula. Obviously, different truth assignment can provide the same relevant valuation to a given propositional formula. Furthermore, the number of different relevant valuations associated with a propositional formula with n variables and assigning 0 value to all variables which do not appear in the formula, is 2^n.

We say that a propositional formula, φ, is *satisfiable* if and only if there is at least a truth assignment, σ, such that $\sigma(\varphi) = 1$. Thus, a propositional formula is not satisfiable if any truth valuation makes false this formula.

Two propositional formulas are *semantically equivalent* if $\underline{\text{any}}$ truth valuation assigns the same value to both of them. One can easily prove that every propositional formula has a semantically equivalent formula in **CNF**, such that it is in simplified form.

The `SAT` problem is the following: *given a boolean formula in conjunctive normal form (CNF), in a simplified way, to determine whether or not there exists an assignment to its variables on which it evaluates true.* This is a well known **NP**-complete problem [72].

The satisfiability problem (`SAT`) has a simple statement and many solutions can be provided following a brute force algorithm: firstly, all possible relevant valuations to the formula are generated (what will require an exponential amount of time), and then, each truth valuation will be checked (polynomial time) if it makes true the formula, in which case the algorithm stops and returns an affirmative answer. If an affirmative answer is not returned after making all possible checks, then the answer will be negative.

1.6.1 An efficient solution to SAT by means of P systems with active membranes

This section presents an efficient solution to the `SAT` problem by a family of recognizer P systems with active membranes, in accordance with the Definition 1.5 given Section 1.5.1.

We start by recalling that the map f from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$ defined by $f(m,n) = \frac{(m+n)\cdot(m+n+1)}{2} + m$ is a polynomial–time computable function (the *pair function*) which is also a primitive recursive and bijective function. We denote $f(m,n) = \langle m,n \rangle$.

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

- The input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} | 1 \le i \le m, 1 \le j \le n\}$.
- The working alphabet is

$$\begin{aligned}\Gamma \;=\; & \Sigma \cup \{c_k | 1 \le k \le m+2\} \;\cup\; \{d_k | 1 \le k \le 3n+2m+3\} \cup \\ & \cup\; \{r_{i,k} | 0 \le i \le m, 1 \le k \le 2n\} \;\cup \{e, t\} \;\cup\; n\{Yes, No\}\end{aligned}$$

- The set of labels is $\{1, 2\}$.
- The initial structure of membranes is $\mu = [\;[\;]_2\;]_1$.
- The initial multisets associated with the membranes are $\mathcal{M}_1 = \emptyset$ y $\mathcal{M}_2 = \{d_1\}$.
- The input membrane is the one labeled by 2.
- The set $\mathcal{R}$ consists of the following rules:

  (a) $[d_k]_2^0 \to [d_k]_2^+ [d_k]_2^-$, for $1 \le k \le n$.

(*b*) $[x_{i,1} \to r_{i,1}]_2^+$, $[\ \overline{x}_{i,1} \to r_{i,1}]_2^-$, for $1 \le i \le m$.
$[x_{i,1} \to \lambda]_2^-$, $[\overline{x}_{i,1} \to \lambda]_2^+$, for $1 \le i \le m$.

(*c*) $[x_{i,j} \to x_{i,j-1}]_2^+$, $[x_{i,j} \to x_{i,j-1}]_2^-$, for $1 \le i \le m,\ 2 \le j \le n$.
$[\ \overline{x}_{i,j} \to \overline{x}_{i,j-1}]_2^+$, $[\ \overline{x}_{i,j} \to \overline{x}_{i,j-1}]_2^-$, for $1 \le i \le m,\ 2 \le j \le n$.

(*d*) $[d_k]_2^+ \to [\ ]_2^0 d_k$, , $[d_k]_2^- \to [\ ]_2^0 d_k$, for $1 \le k \le n$.
$d_k[\ ]_2^0 \to [d_{k+1}]_2^0$, for $1 \le k \le n-1\}$.

(*e*) $[r_{i,k} \to r_{i,k+1}]_2^0$, for $1 \le i \le m,\ 1 \le k \le 2n-1$.

(*f*) $[d_k \to d_{k+1}]_1^0$, for $n \le k \le 3n-3$; $[d_{3n-2} \to d_{3n-1}e]_1^0$.

(*g*) $e[\ ]_2^0 \to [c_1]_2^+$; $[d_{3n-1} \to d_{3n}]_1^0$.

(*h*) $[d_k \to d_{k+1}]_1^0$, for $3n \le k \le 3n+2m+2$.

(*i*) $[r_{1,2n}]_2^+ \to [\ ]_2^- r_{1,2n}$.

(*j*) $[r_{i,2n} \to r_{i-1,2n}]_2^-$, for $1 \le i \le m$.

(*k*) $r_{1,2n}[\ ]_2^- \to [r_{0,2n}]_2^+$.

(*l*) $[c_k \to c_{k+1}]_2^-$, for $1 \le k \le m$.

(*m*) $[c_{m+1}]_2^+ \to [\ ]_2^+ c_{m+1}$.

(*n*) $[c_{m+1} \to c_{m+2}t]_1^0$.

(*o*) $[t\ ]_1^0 \to [\ ]_1^+ t$.

(*p*) $[c_{m+2}]_1^+ \to [\ ]_1^- Yes$.

(*q*) $[d_{3n+2m+3}]_1^0 \to [\ ]_1^+ No$.

Let $\varphi = C_1 \wedge \cdots \wedge C_m$ be a propositional formula in **CNF** such that the set of variables of the formula is $Var(\varphi) = \{x_1, \ldots, x_n\}$, consisting of m *clauses* $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}$, $1 \le i \le m$, where $y_{i,i'} \in \{x_j, \neg x_j : 1 \le j \le n\}$ are the literals of φ. Without loss of generality, we can assume that the formula is in simplified expression, i.e. no clause contains two occurrences of the same literal (the formula is not redundant at clause level), and no clause can contain, simultaneously, a literal and its negation (otherwise, that clause would be satisfiable for any assignment and, consequently, it may be removed from the formula).

Next, we consider a polynomial encoding (cod, s) of the `SAT` problem in the family $\mathbf{\Pi_{am-SAT}} = \{\Pi_{am-SAT}(t) \mid t \in \mathbb{N}\}$. The function *cod* associates to the previously described propositional formula φ, that is an instance of `SAT` with the parameters n (number of variables) and m (number of clauses), the following multiset of objects

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

In this case, object $x_{i,j}$ represents that variable x_j belongs to clause C_i.

The *size* function, s, is defined as follows $s(\varphi) = \langle m, n \rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$. Then, $cod(\varphi)$ is an input multiset of the system $\Pi_{am-SAT}(s(\varphi))$ and the pair (cod, s) is therefore a polynomial encoding of the `SAT` problem in the family $\mathbf{\Pi_{am-SAT}}$. Thus, the system of the family $\mathbf{\Pi_{am-SAT}}$ processing the instance φ will be the P system with active membranes $\Pi_{am-SAT}(s(\varphi))$ with input multiset $cod(\varphi)$.

The execution of the system $\Pi_{am-SAT}(s(\varphi))$ with input $cod(\varphi)$ is structured in four phases:

- *Generation phase*: all possible relevant truth assignment is generated for the set of variables of the formula $\{x_1, \dots, x_n\}$. It is implemented by a loop using division rules in the internal membranes (labeled by 2). This will allow the generation of 2^n membranes that will properly encode all possible assignments. Nevertheless, in this phase, while the valuations are being generated, the clauses that are true by the encoded valuation in each internal membrane are checked. This idea is implemented through a very sophisticated process by which only the truth values 1 and 0 are given to the variable 1. This variable 1 corresponds to the variable x_1 in the first loop step, but by a set of indices, the variable 1 corresponds to the variable x_2 in the second loop step, and so on. This phase is executed in $3n - 1$ computation steps, and only the rules (a), (b), (c), (d) and (e) are applied.

- *Synchronization phase*: it prepares the system for the checking phase synchronizing the execution of the system by unifying certain sub-indices of some objects. The execution of this phase consumes $2n$ computation steps, and only rules (e), (f) and (g) are executed.

- *Check-out phase*: in this phase, it is determined how many clauses are true for each truth assignment encoded by the internal membranes. This is done using the objects c_k $(k > 1)$, whose appearance in a membrane means that exactly $k - 1$ clauses are made true by the encoded valuation in that membrane. This phase is executed in $2m$ steps, and rules (h), (i), (j), (k) and (l) are applied.

- *Output phase*: in this phase the system provides the corresponding output depending on the analysis of the check-out phase. That is, this step performs a search of the internal membranes encoding a solution (i.e., containing object c_{m+1}). If a membrane satisfies the above condition, the object Yes is sent to the environment, and the system stops. Otherwise, the object No is sent to the environment and the system stops. The execution of this phase is done in 4 steps and the used rules are (m), (n), (o), (p) and (q).

1.6.2 An efficient solution to SAT by means of tissue P systems with cell division

This section presents an efficient solution to `SAT` problem by means of family of recognizer tissue P systems with cell division, according to Definition 1.5.

For each pair of natural numbers $m, n \in \mathbf{N}$, we will consider the recognizer tissue P system with cell division $\Pi_{tsp-SAT}(\langle m, n\rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, R, 2)$ of degree 2, defined as follows:

- The input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} |\ 1 \le i \le n,\ 1 \le j \le m\}$

- The working alphabet is

$$\begin{array}{rcl}\Gamma & = & \Sigma \cup \{a_i, t_i, f_i \mid 1 \le i \le n\} \cup \{r_i \mid 1 \le i \le m\} \cup \\ & & \cup\ \{T_i, F_i \mid 1 \le i \le n\} \cup \{T_{i,j}, F_{i,j} \mid 1 \le i \le n, 1 \le j \le m+1\} \cup \\ & & \cup\ \{b_i \mid 1 \le i \le 2n+m+1\} \cup \{c_i \mid 1 \le i \le n+1\} \cup \\ & & \cup\ \{d_i \mid 1 \le i \le 2n+2m+nm+1\} \cup \\ & & \cup\ \{e_i \mid 1 \le i \le 2n+2m+nm+3\}\ \cup\ \{f, g, \texttt{yes}, \texttt{no}\}\end{array}$$

- The environment alphabet is $\mathcal{E} = \Gamma - \{\texttt{yes}, \texttt{no}\}$.

- The set of labels is $\{1, 2\}$.

- The initial multisets associated with the cells are $\mathcal{M}_1 = \{\texttt{yes}, \texttt{no}, b_1, c_1, d_1, e_1\}$ and $\mathcal{M}_2 = \{f, g, a_1, a_2, \ldots, a_n\}$.

- The input cell is the one labeled by 2, and the output region is the environment.

- The set $\mathcal{R}$ is formed by the following rules:

1. **Division rule:**

   (a) $[\ a_i\ ]_2 \rightarrow [\ T_i\ ]_2[\ F_i\ ]_2$, for $i = 1, 2, \ldots, n$.

2. **Communication rules:**

   (b) $(1, b_i/b_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (c) $(1, c_i/c_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (d) $(1, d_i/d_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (e) $(1, e_i/e_{i+1}, 0)$, for $i = 1, \ldots, 2n + 2m + nm + 2$.
   (f) $(1, b_{n+1}c_{n+1}/f, 2)$.
   (g) $(1, d_{n+1}/g, 2)$.
   (h^*) $(1, f^2/f, 0)$.
   (h) $(2, c_{n+1}T_i/c_{n+1}\ T_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (i) $(2, c_{n+1}F_i/c_{n+1}\ F_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (j) $(2, T_{i,j}/t_i\ T_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (k) $(2, F_{i,j}/f_i\ F_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (l) $(2, b_i/b_{i+1}, 0)$.
   (m) $(2, d_i/d_{i+1}, 0)$, for $i = n + 1, \ldots, 2n + m$.
   (n) $(2, b_{2n+m+1}\ t_i\ x_{i,j}/b_{2n+m+1}\ r_j, 0)$.
   (o) $(2, b_{2n+m+1}\ f_i\ \overline{x}_{i,j}/b_{2n+m+1}\ r_j, 0)$, for $1 \leq i \leq n$ and $1 \leq j \leq m$.
   (p) $(2, d_i/d_{i+1}, 0)$, for $i = 2n + m + 1, \ldots, 2n + m + nm$.
   (q) $(2, d_{2n+m+nm+j}\ r_j/d_{2n+m+nm+j+1}, 0)$, for $j = 1, \ldots, m$.
   (r) $(2, d_{2n+2m+nm+1}/f\ \texttt{yes}, 1)$.
   (s) $(2, \texttt{yes}/\lambda, 0)$.
   (t) $(1, e_{2n+2m+nm+3}\ f\ \texttt{no}/\lambda, 0)$.

Let $\varphi = C_1 \wedge \cdots \wedge C_m$ be a propositional formula in **CNF** such that the set of variables of the formula is $Var(\varphi) = \{x_1, \ldots, x_n\}$, and consists of m *clauses* $C_j = y_{j,1} \vee \cdots \vee y_{j,k_j}$, $1 \leq i \leq m$, where $y_{j,j'} \in \{x_i, \neg x_i : 1 \leq i \leq n\}$ are the literals of φ. Without loss of generality, we can assume that the formula is in simplified expression.

Next, we consider a polynomial encoding (cod, s) of the `SAT` problem in the family $\mathbf{\Pi_{tsp-SAT}} = \{\Pi_{tsp-SAT}(t) \mid t \in \mathbb{N}\}$. The function *cod* associates to the previously described propositional formula φ, that is an instance of `SAT` with parameters n (number of variables) and m (number of clauses), the following multiset of objects

$$cod(\varphi) = \bigcup_{i=1}^{m} \{x_{i,j} | x_i \in C_j\} \cup \{\overline{x}_{i,j} | \neg x_i \in C_j\}$$

In this case, object $x_{i,j}$ represents that variable x_i belongs to clause C_j.

The *size* function, s, is defined as follows $s(\varphi) = \langle m, n \rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$.

The system of the family $\mathbf{\Pi_{tsp-SAT}}$ to process the instance φ will be the tissue P system $\Pi_{tsp-SAT}(s(\varphi))$ with input multiset $cod(\varphi)$.

The execution of the system $\Pi_{tsp-SAT}(s(\varphi))$ with input $cod(\varphi)$ is structured in six phases:

- *Valuations generation phase*: in this phase all the possible relevant truth valuations are generated for the set of variables of the formula $\{x_1, \ldots, x_n\}$. It is implemented by using division rules (a), whereby each object x_i produces two new cells, one having the object T_i, that codifies the true value of the variable x_i, y and the other having the object T_i, that codifies the false value of the variable x_i. Thus, 2^n cells are obtained in n computation steps. These cells are labeled by 2, and each one codifies each possible truth valuation of the set of variables $\{x_1, \ldots, x_n\}$. Meanwhile, the objects f, g are replicated in each created cell. This phase spends n computation steps.

- *Counters generation phase*: simultaneously, and using the rules (b), (c), (d) and (e), the counters b_i, c_i, d_i, e_i of the cell labeled by 1, are evolving such that in each computation step the number of objects in each one are doubling. Thereby, through this process and after n steps, we get 2^n copies of the objects b_{n+1}, c_{n+1}, and d_{n+1}. Objects $b's$ will be used to check which clauses are satisfied for each truth valuation. Objects $c's$ are used to obtain a sufficient number of copies of t_i, f_i (namely, m). Objects $d's$ will be used to check if there is at least one valuation satisfying all clauses. Finally, objects $e's$ will be used to produced, in its case, the object `no` at the end of the computation.

- *Checking preparation phase*: this phase aims at preparing the system for checking clauses. For this, at step $n + 1$ of the computation, and by the application of the rules (f) and (g), the counters b_{n+1}, c_{n+1}, d_{n+1} of the cell 1 is exchanged for the objects f and g of the 2^n cells 2. Thus, after this step, each cell labeled by two has a copy of the objects b_{n+1}, c_{n+1}, d_{n+1}, while the cell 1 has 2 copies of the objects f and g.

  Subsequently, the presence of an object c_{n+1} in each one of the 2^n cells labeled by 2 allows to generate the objects $T_{i,1}$ and $F_{i,1}$. By the application of rules (j) and (k), these objects allows the emergence of m copies of t_i and m copies of f_i, according to the values of truth or falsity that a cell 2 assigns to a variable x_i. This process spends $n + m$ steps since there is only one object c_{n+1} in each cell 2 and, moreover, for each $i = 1, \ldots, n$, the rules (j) and (k) are applied exactly m consecutively times. Simultaneously, in the first steps of this process, the application of the rule (h^*) makes the cell labeled by 1 to appear only one copy of the object `yes`.

Simultaneously in this phase, the counters b_i, d_i and e_i are evolving by the applications of the corresponding rules.

- *Checking clauses phase*: in this phase it is determined which clauses are true for every truth valuation encoded by a cell labeled by 2. This phase starts at the computation step $(n+1)+(n+m)+1 = 2n+m+2$. Using the rules (n) and (o), the true clauses are checked for each valuation encoded by a cell, so that the appearance of an object r_j in a cell 2 means that the corresponding valuation makes true the clause C_j. Bearing in mind that a single copy of the object b_{2n+m+1} is in each cell, the phase takes nm computation steps.

  Thus, the configuration $\mathcal{C}_{2n+m+nm+1}$ is characterized by the following:

  - It contains exactly 2^n cells labeled by 2. Each one contains the object $d_{2n+m+nm+1}$, and copies of objects r_j for each clause C_j made true by the encoded valuation in the cell.
  - It contains a unique cell labeled by 1, containing a copy of objects `yes`, `no`, f, g and the counter $e_{2n+m+nm+2}$.

  This phase consumes m computation steps.

- *Formula checking phase*: in this phase it is determined if there exists any valuation making true the m clauses of the formula. For this, the rules of type (q) are used, analyzing in an ordered way (first the clause C_1, after that clause C_2, and so on) if the clauses of the formula are being satisfied by the represented valuation in the corresponding cell labeled by 2. For example, from counter $d_{2n+m+nm+1}$ appearing in every cell 2, the appearance of the object r_1 (the valuation makes true clause C_1) permits to generate in that cell the object $d_{2n+m+nm+2}$. This object, in turn, permits to evolve object $d_{2n+m+nm+3}$ if in that cell appears the object r_2. In this manner, a valuation represented by a cell labeled by 2 makes true the formula φ if and only if the object $d_{2n+m+nm+m+1}$ appears in the content of that cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$.

- *Output phase*: in this phase the system will provide the corresponding output, depending on the analysis in the formula checking phase.

If the formula φ is satisfiable, then there is some cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ that contains an object $d_{2n+m+nm+m+1}$. In this case, the application of rule (r) sends an object f and the object `yes` to the cell 1. The object `yes` therefore disappears from cell 1, and consequently, rule (t) can not be applied. In the next computation step, the application of the rule (s) produces an object `yes` in the environment (for the first time during the whole computation) and the process ends.

If the formula φ is not satisfiable, then there no exist any cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ containing an object $d_{2n+m+nm+m+1}$. In this case, the rule (r) is not applicable, and in the next computation step, the counter e_i evolves, providing an object $e_{2n+m+nm+m+3}$ in the cell 1. This object permits the application of rule (t), since the objects `no` and f remains in the cell 1. In this way, the object `no` is sent in the next computation step, and the computation finalizes.

It can be easily proved that the family $\mathbf{\Pi_{tsp-SAT}} = \{\Pi_{tsp-SAT}(\langle m, n\rangle) : n, m \in \mathbf{N}\}$, defined above, is polynomially uniform by deterministic Turing machines. For this, it is enough to keep in mind that the systems of the family have benn defined through recursive expressions, and the amount of resources needed to describe the system $\Pi_{tsp-SAT}(\langle m, n\rangle)$ is quadratic in $\max\{m, n\}$. Indeed:

1. Size of the alphabet: $6nm + 12n + 7m + 12 \in \Theta(nm)$.
2. Number of initial cells: $2 \in \Theta(1)$.
3. Number of initial objects: $n + 8 \in \Theta(n)$.
4. Number of rules: $4nm + 10n + 3m + 16 \in \Theta(nm)$.
5. Upper limit of rule length: $5 \in \Theta(1)$

1.7 Applications of P systems

Although the Membrane Computing paradigm has a biological inspiration, it also constitutes a good theoretical model of distributed computing, where different calculation units operate independently but structured in a certain vertical hierarchy. For example, the hierarchy utilized to establish connections in networks such as the Internet can be represented as a structure of membranes.

Membrane Computing, according to the original motivation, were not intended to provide a comprehensive and accurate model of the living cell, without aiming to faithfully model biological facts in such a way to provide a modeling framework for the use of biologists, rather, to explore the computational nature of various feature of biological membranes. Indeed, most variants of membrane systems have been proved to be computationally complete, that is equivalent in power to Turing machines, and computationally efficient, that is able to solve computationally hard problems in polynomial time by trading time with space.

However, after significantly developing at the theoretical level, the domain started to be useful for different applications in the framework of Systems Biology and Population Dynamics. A standard procedure is the following: a Membrane Computing model is constructed for a given process/phenomena/population, a software application is developed for simulating that model, and then computer experiments are carried out to experimentally validate the model by using experimental data. Once the model is considered as validated, virtual experiments can be executed for scenarios of expertise's interest. Recently membrane systems have been used to model biological phenomena within the framework of computational systems biology presenting models of oscillatory systems [66], signal transduction [128], gene regulation control [144], quorum sensing [143], metabolic systems [96, 38, 97] metapopulations [135] and real ecosystems [35, 34, 50].

Nevertheless, it has been reported that the macroscopic, deterministic and continuous approach followed by ordinary differential equations (ODEs) is questionable in cellular systems with low number of individuals and heterogeneous structures. The approach based on P systems has a series of features which answer several of those limitations or difficulties. For example, the concept of modularity, associated to extensibility (small changes on the modeled system imply few changes on the model), is intrinsic to a membrane system, what is crucial in biology, but not specific to systems of ODEs. Of course, this does not mean at all that Membrane Computing should substitute ODEs in all applications. Membrane Computing – in general, multiset processing by means of rewriting-like rules – is a complementary technique to systems of differential equations, in many cases as relevant as differential equations, in most cases much easier to use, and in some cases the unique technique which can be used; this last situation is met, for instance, when we deal with small populations of reactants, such that a discrete model is the only one adequate (approximating finite by infinite is useful, provided that the "finite" is large enough).

Besides applications in biology, Membrane Computing was considered also in other areas, such as computer graphics (models based on compartmentalized Lindenmayer systems proved to be more powerful and more efficient than those using classic L systems), cryptography, modeling in a uniform way parallel architectures, in linguistics, economics, etc.

"We avoid to plainly say that we have 'implementations' of P systems, because of the inherent non-determinism and the massive parallelism of the basic model, features which cannot be implemented, at least in principle, on the usual electronic computer - but which can be implemented on a dedicated, reconfigurable, hardware [...] or on a local network."

Gheorghe Păun (2005)

2 Software Applications for Membrane Computing

Currently, we lack of a feasible biological implementation, either *in vivo* or *in vitro*, for P systems. The only way to analyze and execute these devices is by using programs running on electronic computers which are limited by physical laws. Therefore, P systems simulators are tools that assist the researchers to extract results from models, and to experimentally validate them by reproducing their computation.

Nevertheless, in order to provide useful tools with a quick response time, these simulators have to be as much efficient as possible. In this regard, parallel platforms are object of study, since they can effectively implement P systems parallelism (normally in a partial way).

A survey on the simulation of P systems is presented in this chapter. First, a discussion of the typical architecture of P systems simulators is given, together with the most known P systems simulators (Section 2.1) and the P-Lingua framework (Section 2.2). Then, we stress the necessity of improving the efficiency of the simulators to handle models with large size. Finally, we show an overview of parallel platforms utilized for this purpose (Section 2.3), and parallel solutions developed so far (Section 2.4).

2.1 P systems simulators

In this section, an introduction to the typical structure of P systems simulators is provided. Then, a survey of the simulators and applications for Membrane Computing is given, together with their categorization in two generations.

2.1.1 P systems simulation overview

P systems are bioinspired devices that work in a massively parallel and non deterministic way. While there are preliminary studies analyzing the problems related to real implementations, there is still a long way to reach this ultimate goal. That is why the *simulation* of P systems by using conventional computers becomes a vital necessity for the advancement of scientific activities in computing paradigms that are still to be implemented (in particular, Membrane Computing).

The usage of P systems simulators provides three main functionalities for the scientific community:

1. *Educational purposes*: all the information of the simulated P systems computations should be shown in a intuitively way.

2. *Assisting the design* of P systems based models: a circular development process is carried out, where both model and simulator are validated towards one to each another.

3. *Serving as the core of more elaborated software tools*: for example, applications oriented to predict the behavior of complex real-life processes, such as Populations Dynamics.

Over the last years, a wide range of such simulators have been reported [57]. Most of them shows a typical software architecture including three main modules: *input* (P system definition), *core* (simulation engine), and *output* (presentation of results). Next, we briefly detail the role of each mentioned module.

Definition of the P system to be simulated

In order to simulate a P system, it is necessary to provide at least the following information: the P system model to be used, the membrane structure, the initial multisets of objects, and the set of rules.

We can observe three main classes of solutions for this module implemented by the reported simulators in [57]:

- Definition of the systems in the source code.
- Definition by using *ad hoc* user interfaces (usually Graphical User Interfaces, GUIs)
- Definition by using external files and parsers to process the files.

There are some considerations that should be taken into account when designing this module. Defining the P system in the source code is a fast solution for prototyping because we can use the same programming language used for the simulation core (or another common programming language). But this solution presents strong dependencies with the simulation core. Thus, the definition is not very reusable, and difficult to be changed without altering the simulation core. In this scenario, an experimental validation for the simulation core can be tricky.

Moreover, an *ad hoc* Graphical User Interface (GUI) provides an user-friendly way to specify the P system to be simulated. But developing *ad hoc* GUIs is a harder task than including the P system definition in the source code. Furthermore, it is a moderately reusable solution, because the GUI could contain dependencies on programming libraries or specific programming languages.

Finally, defining the P system by using text files and parsers is a solution hard to program the first time, but it is a very reusable solution because the P system definitions do not have dependencies with the simulation core or other programming languages. Thus, the same definitions of P systems can be used in different software environments by using parsers. Additionally, a GUI can be used in combination with this solution to provide a more user-friendly solution.

Simulation core

The simulation core is the part of the simulator that captures the semantics of the defined P system and reproduces its behavior along one or more computations within the limitations of conventional machines.

At this stage, the definition of the P system to be simulated should be free of errors. The parser in the previous module and a low level coupling between modules should guarantee it.

The simulation core implements a *simulation algorithm* whose goal is to obtain one or more computations of the defined P systems. It is very important to design efficient simulation algorithms to reproduce the behavior of interesting P systems instances. In this regard, simulation algorithms can be adapted and/or implemented in specific high-throughput hardware, as those presented in this thesis.

Presentation of simulation results

The simulation core reproduces, step by step, one or more computations of the defined P system. After the simulation, it is necessary to extract some relevant information about the simulated computations and show it to the user. The shown information depends mainly on the simulator goal. Some times could be useful to show all the computation (i.e., for pedagogical or debugging purposes); some times it is necessary to filter and process the information (i.e., for simulators that reproduce the behavior of P systems modeling real-life processes). In any case, the simulation results can be shown to the user in several ways: graphics, data tables, files, etc.

2.1.2 Related simulators

A review of the simulators developed in the first ten years of Membrane Computing is provided in [57]. The evolution of those simulators can be seen as an expression of the evolution of the research itself. In this sense, it is usually to speak about two generations of simulators according to its finality. It is worth to mention that both generations are overlapped on time.

In the first generation, the simulators were used for checking the correctness of the hand constructed P systems. To this end, they usually show to the user the whole computations, including information about which and how many times the rules are applied. These simulators are very good assistants for the design of membrane solutions to computationally hard problems, since it is difficult to ensure the correctness of hand-made simulations when the number of compartments and objects are large. Good examples of this generation are *Malita*'s simulator [95], the first transition P system simulator that only reproduces one computation; *Balbontin*'s simulator [24], another transition P system simulator, but that calculates the whole computation tree; and, increasing the number of membranes: *Ciobanu*'s simulator [48] and two simulators written in CLIPS and Prolog [126, 55] by the *Research Group of Natural Computing of the University of Seville* in their first stage of developing simulators.

In the second generation, P systems are not the object to study: they have become the tool for studying real-life processes, especially from biology. The results of the simulation core must be processed and filtered, showing only relevant information to the user related to the modeled real-life process. One of the first simulators in this line is *Romero-Campero*'s simulator, which implements the multi-compartmental Gillespie algorithm [121] in SciLab and C. This simulator has been successfully used in addressing several real-world problems as the simulation of a signaling pathway associated with the Epidermical Growth Factor [127]; simulation of FAS-induced apoptosis [47]; modeling gene expression control [144]; or the first computational model of Quorum Sensing [143, 152] in *Vibrio Fischeri.* Other relevant simulators in the second generation are: *Cyto-Sim* [148], a software that can simulate micro and macroscopic biological processes using arbitrary kinetic laws; *Meta P-lab* [37], a virtual laboratory which aims at assisting modelers both to understand the internal mechanisms of biological systems and to forecast, in silico, their response to external stimuli, environmental condition alterations or structural changes; and *Infobiotics Workbench* [28], an integrated software suite incorporating model specification, simulation, parameter optimization and model checking for Systems and Synthetic Biology.

2.2 P-Lingua, and the pLinguaCore framework

Each P system model displays semantical constraints that determine the way in which rules are applied, as explained above. Hence, software simulators have to be capable of supporting different scenarios when simulating P system computations. Moreover, simulators have to receive a precise definition of the specific P system to be simulated. Along this thesis, the term *simulator input* will refer to this definition.

One approach could be defining a specific input file format for each simulator (or directly implement it on the source code), but it would require a great redundant effort. A second approach could be to standardize the simulator input, so all simulators will process inputs specified in the same format. Nevertheless, each approach raises up a trade-off: on the one hand, specific simulator inputs could be defined in a more straightforward way, as the used format is closer to the P system features to simulate; on the other hand, although the first approach involves developing a different standard format for each P system model, a common standard format can avoid to completely develop a new simulator for each new P system to be simulated. In fact, it is possible to use a common software library for parsing the standard input format. Moreover, users would not have to learn a new input format each time they use a different simulator. They would also require not to change the way to specify P systems when they move on to another model, as they would keep on using the standard input format.

This second approach is the one considered in P-Lingua [124, 16], which is a definition language able to describe P systems within several P system models. The P-Lingua project also provides free software tools under GNU GPL license [2] for compilation, simulation and debug tasks. These tools are integrated in a Java library called *pLinguaCore*, including a parser to handle P-Lingua input files and check possible programming errors (both lexical/syntactical and semantical); several built-in (sequential) simulators to reproduce P system computations for the supported models; and the ability to export the P-Lingua definition file to other file formats in order to get interoperability between different software environments. This is the main method to obtain the input for the simulators implemented in this thesis. Figure 2.1 illustrates this approach to define the simulator input by using the P-Lingua framework. Such inputs are free of programming errors since the parser inside pLinguaCore has already checked them.

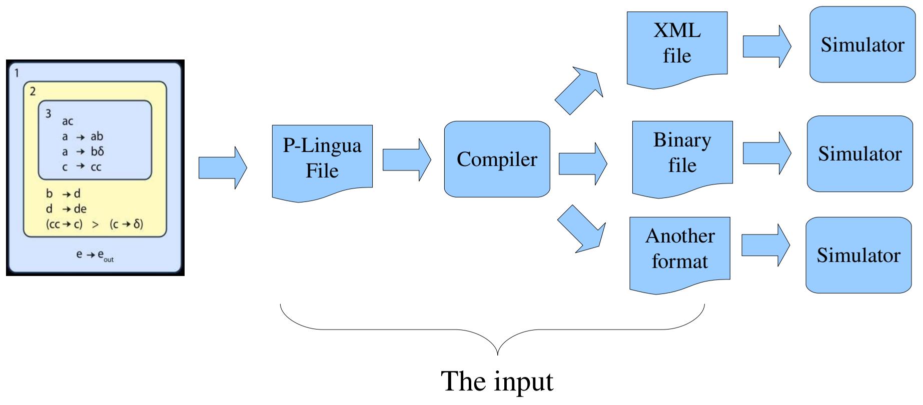


Figure 2.1: The P-Lingua approach to define simulators inputs

As mentioned above, the pLinguaCore library includes several built-in simulators for the supported models. The current version of pLinguaCore is 3.0 and can be downloaded from the P-Lingua website [16]. Each version of P-Lingua and pLinguaCore adds new supported models and implements new simulation algorithms, the syntax definition of the language and more details can be found on the related papers and the website. Next, a chronological enumeration of the different versions of P-Lingua and pLinguaCore library is presented, including the features of each version and related papers.

1. **P-Lingua 1.0** [58, 8]: this initial version is able to define and simulate tractable instances of P systems with active membranes. The simulator only reproduces one possible computation and, therefore, the simulated P system must be confluent to obtain a useful answer.

2. **P-Lingua 2.0** [70, 69, 16]: several cell-like P system models are incorporated, together with one or more built-in simulators for each:

    - Transition P systems.
    - Symport/antiport P systems.
    - Active membranes.
    - Active membranes with creation rules.
    - Stochastic P systems.
    - Probabilistic P systems.

3. **P-Lingua 2.1** [107, 16]: tissue-like P systems with division rules are also supported, including its built-in simulator and some fixed bugs.

4. **P-Lingua 3.0** [98, 16]: Population Dynamics P systems (PDP Systems) are also supported, and several built-in simulators for this model are added. It includes sequential implementations of the simulation algorithms presented in this thesis (DNDP and DCBA). Furthermore, some general bugs are fixed and the support of stochastic P systems is discontinued, encouraging the use of Infobiotics Workbench [28] in this respect.

There are three major advantages for which it has been decided to use P-Lingua and pLinguaCore in this thesis. First, P-Lingua is a standard format to define P systems that can be traduced to syntactically free of errors outputs, since the included parser in pLinguaCore checks them. Secondly, a natural way for us to implement the new simulation algorithms presented in this thesis is to first develop a built-in simulator in pLinguaCore, since this process is relatively straightforward. They implement sequential versions, making use of the functionality provided in the pLinguaCore software library. Thirdly, the same P-Lingua input files can be the input for the GPU simulators and also for the built-in pLinguaCore simulators. This is a good method to validate the GPU simulators by comparing the results for small instances of P systems.

2.3 Improving the efficiency of P systems simulators

In this section we discuss the importance of improving the performance of P systems simulators. In Chapter 3, the concepts of efficiency and performance will be detailed, but here they refer to reducing time and/or memory of the simulation. Other concepts and technologies here mentioned are also explained in Chapter 3. Moreover, an analysis of parallel platforms for simulating P systems is provided, together with a survey of parallel simulators for P systems.

2.3.1 P system parallelism implementation

In [80], the *Sevilla team*[1] said: "*the next generation of simulators may be oriented to solve (at least partially) the problems of storage of information and massive parallelism by using parallel language programming or by using multiprocessor computers*". Indeed, although some simulators and software applications have been produced [57], most of them were developed for sequential architectures using non parallel-oriented programming languages such as Java, CLIPS, Prolog or C, where performance is slightly compromised.

The performance of sequential P systems simulators is dramatically decreased as they serialize the natural double massively parallelism of P systems : execution of rules within each membrane, and the evolution of each membrane. Thus, the sequential simulation time proportionally increases as long as the quantity of parallelism presented in the P system. Furthermore, most of the developed simulators are designed for pedagogic purposes (especially those from the first generation), so they devote more resources to output processes than to the simulation core.

However, the last generation of commodity PCs is able to support the fast execution of sequential simulators. They can manage problem instances of enough size for current research lines [124], but very large instance sizes are still unfeasible for them. These large instances are increasingly demanded and, therefore, they require more optimal and efficient simulators. For example, real ecosystems models based on P systems require high throughput simulation tools, so that model designers and other end users can interact in real time with them. It requires a fast and reliable answer from the simulators. This can be achieved by developing simulators that apply parallelism similarly as the theoretical model does. Thus, the massively parallel nature of P systems

[1]Research Group on Natural Computing, University of Seville (Spain).

computations points out to look for a parallel technology where the simulator can run faster.

Although the capacity of existing electronic computers is limited by the physical laws of silicon, simulators making use of all the available parallel resources today would be sufficient to handle instances of considerable size instances concerning relevant real-life problems. In this regard, High Performance Computing (HPC) is the field which studies the set of necessary techniques to accelerate the execution of applications using parallel platforms. We can find these techniques in modern supercomputers [18] and new parallel architectures (GPUs, FPGAs, CellBE, etc.). This opens a new way with many possibilities for the development of efficient simulators. More details about HPC and new parallel architectures are depicted in Chapter 3.

2.3.2 Parallel platforms for P systems

A good computing platform for simulating P systems should provide a good balance between performance, flexibility, scalability, parallelism and cost [76]. The first three features are considered as the most important quality attributes of computing platforms for Membrane Computing applications [115]:

- *Performance*: it refers to the speed at which a platform executes P systems. A suitable measure for this can be the number of rule applications performed per unit time.
- *Flexibility*: it means the ability of the platform to support the execution of a wide range of P systems. A high grade of flexibility involves a great diversity of P systems supported by the platform (regarding the variants of P systems designated to be simulated).
- *Scalability*: it is the ability of the platform to execute P systems with increasing size without reducing the ability to perform its functions or a reduction in the performance. The size of a P system can be determined by the number of membranes, the number of defined rules and the initial multisets.

It is difficult to ensure that a computing platform conforms the three attributes. A factor promoting one attribute can demote another. In fact, there are two main connections between them [115]:

- *Flexibility vs performance*: the performance of a platform can be improved by adapting it to a restricted set of specific properties of a P systems. That way, there are less elements to reproduce, and it is more easy to carry out optimizations. However, the support of a large diversity of P systems hampers the process of tailoring the implementation to them. Therefore, increasing the performance of a platform comes at the cost of reduced flexibility, and vice versa.

- *Flexibility vs scalability*: increasing the flexibility means supporting additional P systems features. This fact involves the implementation of additional data structures and algorithms, and these data structures also requires more memory. As more flexibility leads to increase resource (memory and processors) consumption, the scalability is then reduced.

Concerning the types of computing platforms for simulating P systems, there are mainly three to mention [115]:

- *Sequential computing platforms*: they are common computers with software programmed microprocessors. In these platforms, the hardware is abstracted by the software it executes. They are a very flexible solution (changes are made on software) at the expense of a low performance (one instruction is executed at a time unit).

- *Software-based parallel computing platforms*: they are based on clusters of processors, being each one a sequential computing platform itself, and interconnected through a fast bus. Because each processor executes in parallel, it can outperform the sequential computing platform. The main problem is the required synchronization process, that can be time consuming. Although more processors can be included to improve the performance, the overhead taken by the synchronization is also increased, what reduces significantly the expected performance. Therefore, the scalability of these systems is supported, but performance is limited.

- *Hardware-based parallel computing platforms*: they execute algorithms directly on hardware. Spatially separated processors run those tasks, what significantly improve performance. However, the use of the spatial dimension of processors constrain the amount of hardware resources available. Therefore, scalability is reduced, as well as the flexibility, since any change made on the model involves a change of the whole hardware design.

When designing parallel simulators for P systems, we have to, first, study the platform and the software to use. Then, we can adopt different approaches to implement real parallelism of P systems, depending on the P system model, and, obviously, on the parallel platform.

2.4 Parallel simulation of P systems

In this section we provide an overview of parallel simulators so far developed for P systems. As parallel simulators require to implement parallelism, they are based on parallel computing platforms. That is, computing platforms, which contain internal parallel architecture. There are many HPC solutions that have been utilized for Membrane Computing applications. Although we will provide a detailed introduction to them in Chapter 3, we also show short definition of each to better understand the functioning of available parallel simulators for P systems.

2.4.1 Cluster based simulators

A *Cluster* [138] can be defined as a set of computers interconnected through a local network. They work together to execute parallel programs. The connection bus is perhaps the system bottleneck, as long as the number of computers increases.

There are several attempts to implement P systems parallelism on such kind of platforms. In fact, clusters were the first platform used to create parallel simulators. In this context, *Ciobanu and Guo* [49] simulates a restricted set of transition P systems using a Linux based cluster, using C++ and *MPI* library. The design of this simulator is to assign each membrane to each computer of the cluster, and each computer executes rules in sequential mode. The communication mechanism is implemented through message passing in MPI. Although performance was improved, authors indicate that the main limitation was in the communication and cooperation between membranes, what consume most of the total execution time. Further research was carried out in this regard by *Fernández et al.* [151] to partially solve the problem in the communication process.

Another approach was introduced by *Syropoulos et al.* [150]. This simulator works under Java, and makes use of *RMI* (Remote Method Invocation) to distribute the workload. Again, each membrane is distributed along the processors, but the scalability was damage because of the communication overhead. However, their aim was not focused on performance, but was to show the usage of P systems as a foundation of distributed computing.

Finally, a novel alternative has been initiated by *Diez Dolinski et al.* [59]. They provide a highly scalable solution to the natural exponential growth of space made by P systems. For this purpose they use *MapReduce* algorithms over distributed environments. It has been categorized here as for clusters, but it has been conceived to work over grids and Internet. Moreover, a new branch of P-Lingua, called *Distributed P-Lingua*, was developed. Finally, the simulator is implemented in Java, using the freely *Hadoop* library. Although no performance analysis was provided, the first results suggest a promising research line.

2.4.2 FPGAs based simulators

FPGAs (Field Programmable Gate Array) [108] are reconfigurable chips that a hardware designer can program. They allow the designer to distribute resources to processors and memory as he wants. The main constraint of these devices resides in the amount of memory, which is normally low. But they are chip technology that can be interconnected to increase scalability.

In this regard, several implementations have been performed to simulate the transitions of P systems. The first attempt was carried out by *Petreska and Teuscher* [136]. Their simulator provides a full support for a particular set of transition P systems, also supporting division and dissolution rules. It was implemented through *VHDL* (VHSIC hardware description language), using only one type of high-level hardware component. However, it has four main limitations [115]: firstly, it does not implement parallelism inside membranes; secondly, it is inflexible; thirdly, it is not extensible, and so, not a flexible platform; and fourthly, it has limited scalability.

Alternatively, *Fernández et al.* [64] presented the design of a circuit that could be implemented using FPGAs. This circuit selects the active rules of a given transition P system configuration.

An evolution of those works was made by *Nguyen et al.* [117, 115, 116]. Their hardware system, called Reconfig-P, together with P-Builder, provides an elegant approach balancing performance, flexibility and scalability. It uses FPGAs, but the systems built on top of them is intelligent enough to adapt the circuit implementation to the P systems features. Such features can be also non-determinism, as an algorithm, called DND, has been designed to support it at the region level.

2.4.3 Microcontrollers based simulators

A microcontroller is an integrated circuit containing a processor, memory and input/output units. Although they are limited by their set of instructions and number precision, they are very chip technology that can also be interconnected.

Gutiérrez et al. [77, 76] utilize this technology to provide an alternative platform. Their aim is to better balance flexibility and performance at a low cost. The work is based on one of the most used microcontrollers, that is, the *PIC* (Peripheral Interface Controller). The implementation is based on the previously proposed algorithms/architectures for improving the communication among units dealing with membranes [151, 30]. They categorize their solution as a "*partially parallel evolution with partially parallel communication*" [76]. Although no performance analysis is provided, their design will be useful for future work.

2.4.4 Cloud computing based simulators

Cloud computing [153] provides a service where performing calculus in an virtual network. They implements a system similar to a cluster, but for a cheap rental price.

In this regard, another novel approach to provide a high number of resources for simulating P systems creating exponential workspace, was initiated by *Nabil et al.* [114]. Authors use the `SAT` problem as a case study, and run an instance with 11 variables (requiring 2^{11} membranes). Although no performance analysis is provided, the design serves as the base for future work.

2.4.5 GPU computing based simulators

The *GPU* (Graphics Processing Unit) [119] is the processor inside graphics cards. Today, they are used as HPC accelerators, since they contain from hundreds to thousands of cores, and are relatively cheap technologies. Programs, not necessary for graphics, are accelerated by means of *GPU computing.* The programming is flexible, but performance depends on how the implementation fits data parallelism. Two major GPU platforms are available: NVIDIA GPUs with CUDA [89, 5] and OpenCL [112, 6], and AMD with OpenCL.

This thesis presents a systematic work on the simulation of P systems with GPUs. After obtaining the first positive results, other branches were initiated. Next, we highlight some of them (to our knowledge):

- *PMCGPU*: it is the project initiated in our work. It consists on several subprojects aiming the simulation of different P systems models:
  - *PCUDA*: it is the first P system simulator based on CUDA (see Chapter 4). It supports P systems with active membranes, with sequential and parallel versions. It conforms a flexible platform to simulate P systems, but performance and scalability are compromised.
  - *PCUDASAT*: it is a branch of PCUDA, on the fast simulation of a family of P systems with active membranes solving `SAT` (see Chapter 5). It is an *ad-hoc* (non-flexible) platform, so that it behaves as a `SAT` solver by means of P systems. However, performance and scalability are increased.
  - *TSPCUDASAT*: it is a branch of PCUDASAT. Here a family of tissue P system with cell division solving `SAT` is simulated at a fast and scalable way (see Chapter 5). However, the performance is demonstrated to be lower than PCUDASAT simulators.
  - *ABCD-GPU*: it is a recent project on the simulation of PDP systems (see Chapter 7). These systems are the base of a modeling framework for Population Dynamics (specially for real ecosystems). It includes C++ based simulators running on both multicore (by the OpenMP [15] library) and manycore (by CUDA) platforms.

- *Simulation of spiking neural P systems*: the first simulators in this regard have been initiated by *Cabarle et al.* [31, 32]. The implemented algorithm uses a matrix representation of the model, introduced by *Zeng et al.* [156]. Therefore, the implementation is straightforward on the GPU. Moreover, they take advantage of simulating non-determinism as a parallel level. That way, each computation path is driven by each multiprocessor within the GPU.

- *Simulation of evolution-communication P systems with energy and without antiport rules*: the initial work on this has been carried out by *Juayong et al.* [86]. As for the spiking neural P systems simulator, the simulator makes use of a matrix based algorithm representing the semantics of the model. Again, the implementation is directly performed through linear algebra operations, that are efficiently executed on the GPU.

- *Simulation of enzymatic numerical P systems*: this work is carried out by *García-Quismondo et al.* [71]. The simulated models are used for modeling robot controllers, so the results can be significant for the Artificial Intelligence field. Versions in Java (inside pLinguaCore) and C programming languages are also provided.

- *Simulation of image processing solutions with tissue P systems*: *Díaz-Pernil et al* [122] present a new work on the simulation of a kind of tissue P systems models on the GPU. These models represent algorithms for processing images (for example, image smoothing). These applications are specific solutions for processing images, where the ad-hoc simulation of P systems is implicit in source code.

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."

Gene Amdahl (1967)

3 High Performance Computing

The basic question of what *High Performance Computing (HPC)* stands for is hard to be concisely answered. Although there is not a unique definition, we can adopt the following one: *"High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problem instances in science, engineering, or business"* [3].

The term HPC has been historically associated to *Parallel Computing*, since it is the most adopted way to improve the performance in computation nowadays. Parallelism is the basis for the acceleration of large and complex real-world applications. Sometimes, HPC is also merged with the term of supercomputing, although it refers to the development of systems with the highest operational rate for current computers (supercomputers). Vast sums of money are invested to develop supercomputers, and they are used by only specialized expertise for special problems. But a High Performance Computing system can be also used and managed without a lot of expense or expertise.

This chapter briefly introduces the required concepts to understand the meaning of High Performance Computing. We explain the concept of parallelism and its application in *Parallel Computing* [120, 73, 1]. We also mention methodologies, parallel algorithm models, and models of parallel platforms. We turn also to make a brief tour of parallel technologies that exist today, going into detail with the massively parallel architecture of graphics cards.

3.1 Parallel Computing

In this section, the concepts of parallelism and Parallel Computing are introduced. Considerations and other aspects of them are also depicted.

3.1.1 Necessity of parallelism in processors

Since the popularization of computers as a way to quickly and automatically solve problems, there has been an increasing demand for computing power [73]. On one hand, software developers have always demanded more processing speed in order to get more functionality. On the other hand, as long as computing power increases, more problems can be considered to be resolved. Applications for modeling, simulation and analysis applied to areas such as microbiology, biochemistry, astrophysics, particle physics, engineering, Internet and social networks, are currently the most demanding of computing power. Without sufficiently efficient computers, these problems will never get to be handled.

This type of demand has led to increase the speed of conventional microprocessors year after year. This improvement has been related, mainly, to the density level of transistors per chip. As the transistors are smaller, their speed is increased. However, transistor downscaling implies to increase the amount of consumed and dissipated energy as heat. In addition, the scale of the transistors also has a known limit given by quantum physics. Although this limit has not been reached yet, the density level has been declining. From 1986 to 2002, processors speed has increased on average 50 % per year. However, since 2002 this rate has been reduced to 20 % [73].

This change in increasing the speed of processors is associated also to the change of the design thereof. In 2005, most chip makers changed their strategy away from monolithic processors to produce multiprocessor systems. Thus, the number of transistors per chip is better utilized by creating redundant computing elements. This strategy, by making much simpler chip designs, allowed to create cheaper technologies that consume less energy [73].

Note also that not only the parallelism is implemented in the processor chips. This strategy is also carried out in other areas of a computer, such as the data storage. The memories of the computers have been also adapted to be accessed in parallel, in order to be able to always feed data to the processor.

3.1.2 Concurrent and Parallel Computing

Normally, we can find in the literature the terms of concurrency, parallelism and distributed, which are often confused. We consider *Concurrent Computing* to the development of programs where multiple tasks can be in progress at any instant [120]. For example, an operating system can be considered a concurrent computer program because it can execute multiple tasks at any instant, even using a single processor (by serially executing them).

On the other side we find *Parallel Computing*, which refers to the study of solving problems through the use of multiple processors working together. It is worth mentioning that the control of the processors is done explicitly by *parallel programming.* We say that a program is *sequential* (or serial) if the execution of all its instructions are carried out successively. A sequential program will not run faster on many processors, but it will have to be rewritten to take advantage of the their inherent parallelism, if any. This conversion process task is called *parallelization.* In this process, the communication, the load balancing, and the synchronization must be taken into account.

In Parallel Computing, we can find several special types of computing. Here we will mention the *Distributed Computing* and *Heterogeneous Computing.* For the first type, the programs are executed in parallel on processors of separate machines (which have their own memories and perhaps are located in different places). Communication between the different machines is performed by specific protocols, such as message passing. The second type refers to the use of Parallel Computing on multiple processors of different features. Normally there is a high end processor, which keeps the control of what happens in the wide range of more simple processors.

The work presented in this thesis is encompassed in Parallel Computing. P systems provide parallel solutions of problems: these machines execute rules (tasks) in different compartments (distributed), all in a parallel and synchronized way. Furthermore, the efficient simulation of these systems is carried out using techniques of conventional parallel computing: simulating the execution of rules distributed in the membranes on platforms that allow to implement a real parallelism.

3.1.3 Performance estimation

In this section we briefly introduce important concepts to understand the main goal of Parallel Computing: improving the execution performance of programs. We also describe how to measure the cost of an algorithm, and the theoretical performance improvement that can be achieved using parallelism.

First, we need to know what is meant for a program to have a good performance. There are several metrics for analyzing an algorithm, according to the resources it uses. Thus, talking about performance depends on the adopted metrics. Among the metrics used by algorithm developers, we highlight the following (in order of the most frequently used):

- Speed or *run-time*,
- Workspace *memory* required during the execution,
- Data transmission size (e.g. required network bandwidth),
- Temporal external memory size (normally, hard disk) to execute the algorithm,
- External memory size required to store the results of the algorithm,
- Quantity of consumed energy by the hardware that executes the algorithm (normally, measured in watts).

The most important metrics when analyzing an algorithm are those corresponding to the speed and memory space, as these two are based on the use of rather limited resources (CPU usage and main memory). This analysis is based on the estimation of the corresponding resource consumption. This allows us to compare the relative cost of different algorithms to solve the same problem. This is usually done using the concept of *growth ratio*, which makes mention of the increased cost of an algorithm as the input size grows.

A common way of estimating the time cost of an algorithm is through the required number of basic operations. As the actual and accurate basic operations of an algorithm are difficult to estimate, an assessment of the best and worst case by the asymptotic analysis of the algorithm is usually carried out. This analysis is the study of an algorithm as the input size becomes large, or reaches a limit. The upper bound of an algorithm indicates the maximum rate of growth. It is indicated by a special notation called *big-O*. There is a similar notation to indicate the minimum amount of resources required by an algorithm. The way to calculate this bound is analogous to the big-O. This notation is called big-Omega with the symbol Ω. If the upper and lower bounds coincide, one can use the average notation indicated by Θ.

Let us recall that if f, g are computable functions over natural numbers, then $g \in O(f)$ if there exist a real number $c > 0$ and a natural number n_0 such tat $g(n) \leq c \cdot f(n)$, for each $n \geq n_0$. We say that $g \in \Omega(f)$ if $f \in O(g)$. Finally, we say that $g \in \Theta(f)$ if $g \in O(f)$ and $f \in O(g)$.

These upper bounds of the growth rate indicates how expensive is an algorithm. The most frequently used orders in the analysis of algorithms, in increasing order, are:

$O(1) < O(log\ (log\ n)) < O(log\ n) < O(\sqrt{n}) < O(n) < O(n \cdot log\ n)\ ) < O(n^c) < O(c^n) < O(n!) < O(n^n)$, where $c > 1$.

In theory, if we use p processors (assuming they have similar features, and that there are no synchronization overhead), the time cost is reduced p times. For example, if the order of an algorithm is $O(n)$, then, using $O(p)$ processors, the order becomes of $O(\frac{n}{p})$. If an algorithm meets that, it is considered *cost efficient*. However, we will study the performance of parallel programs by using the run-time because for the big-O notation, $O(\frac{n}{p}) = O(n)$ (if p is a constant value independent of n). We will consider the run-time as the time elapsed for the algorithm, and not the used for the system's inputs and outputs.

The aim here is to compare a serial program (taking T_{serial} time units) and a parallel program (taking $T_{parallel}$ time units). The value of T_{serial} should be the run-time of the fastest serial program running on one of the same parallel processor (some other authors consider the serial time using the fastest processor available, but the first approach is more common).

We will assume that a *speedup* is linear when $T_{parallel} = \frac{T_{serial}}{p}$, using p processors. However, this is an ideal result. In practice, the performance improvement is lower because of the communication overhead, synchronization processes and portions of non-parallelizable code. Therefore, we will consider that the speedup of a parallel program is given by

$$S = \frac{T_{serial}}{T_{parallel}}$$

As the resulting number has no units, we usually add it an "x". Note that the linear speedup is given when $S = p$. Moreover, as p increases, we expect S to become a smaller fraction. We call $\frac{S}{p}$ the *efficiency* of a parallel program.

As mentioned before, not all the code of a program is parallelizable. However, the larger the parallelizable part, the better the achieved performance. This observation was first made by Gene Amdahl in 1960 [22]. The *Amdahl's law* states that, assuming A is the parallelizable part, and B the rest, the speedup of a parallel program is given by

$$S = \frac{T_{serial}}{A \cdot \frac{T_{serial}}{p}\ +\ B \cdot T_{serial}}$$

The general state we can get from Amdahl's law is that, if a fraction B of our serial program remains unparallelized, we can't get a speedup better than

$\frac{1}{B}$. However, this restriction is not insurmountable. Amdahl's law doesn't take into account the problem size. A normal behavior in parallel programs is that the parallelizable part increases, in time, more with the problem instance size than the inherently serial part. For many problems, the bigger the instance size, the smaller the serial part. This was mathematically stated in *Gustafson's law* [75], as follows:

$$S(p) = p - \alpha(p - 1)$$

where S is the speedup, p is the number of processors, and α is the non-parallelizable part, given by $\alpha = \frac{a}{a+b}$, being a the sequential time and b the parallel time. From such law we can state that, if faster (more parallel) equipment is available, larger problem instances can be solved in the same time.

We finally give some other useful concepts, related with performance, for Parallel Computing:

- *Scalability*: it is a measure of the capacity of parallel platforms to effectively utilize an increasing number of processors [92]. It is commonly used to select the best algorithm-platform combination for a problem under different constraints on the growth of the problem instance size and the number of processors.
- *Parallel slowdown*: it is a phenomenon where the parallelization of a parallel program beyond a certain point causes the program to run slower (using more processors involves working with less data, but the cost of communication becomes bigger).
- *Embarrassingly parallel* problem: it is one for which little effort is required to separate the problem into a number of parallel tasks.

3.1.4 Design of parallel algorithms

Parallel program design and development is a manual process. The developer is responsible for both identifying and actually implementing parallelism. There are two types of parallelism, based on what (tasks or data) is parallelize [83]:

- *Task-parallelism*: execution processes, or tasks, are distributed across different parallel computing nodes, working on the same or different data.
- *Data-parallelism*: the data is distributed across different parallel computing nodes executing the same task.

There are some methodologies that help parallel program developers to start doing their job. We will highlight the *Foster's methodology*, that was introduced by Ian Foster [67], and is based on the following four stages:

1. *Partitioning*: the computation and the data are divided into small tasks. The aim in this stage is to recognize opportunities for parallel execution.

2. *Communication*: the communication required to coordinate task execution is determined.

3. *Agglomeration*: check if there are tasks and communication structures that can be combined into larger tasks to improve performance.

4. *Mapping*: the composite tasks are assigned to processors, maximizing processor utilization and minimizing communication costs. This stage can be specified statically or dynamically at runtime by load-balancing algorithms.

One objective of the Foster's methodology, and of many other methodologies, is to help the developers to select which *parallel algorithm model* fits better to solve problems in parallel. A parallel algorithm model is a way to structure a parallel program by selecting a decomposition and mapping technique, and applying the correct strategy to minimize communication. An appropriate model selection will allow a successfully parallelization process.

Some of the most common parallel algorithm models are listed below [73]:

- The *data-parallel* model: each task applies similar operations on different portions of data. For most problems, the degree of data parallelism increases with the size of the problem instance, what allows to use more processors to effectively solve larger instances.

- The *pipeline* or producer-consumer model: the data is managed as a stream, passing on through a succession of processors that execute some tasks. This model, which works with data stream, is called stream parallelism. The succession (pipeline) of processors can be seen also as a chain of producers and consumers. It involves a static mapping of tasks onto processors.

- The *task graph* model: the computations in any parallel algorithm can be viewed as a task-dependency graph. This model is normally used to solve problems in which the amount of data associated with tasks is large with respect to the amount of associated computation.

- The *work pool* model: a dynamic mapping of tasks onto processors for load balancing is performed. There is no desired premapping of tasks onto processors. It is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with them.

- The *master-slave* model: there are one or more master processors generating the tasks, and taking control of them. It is suitable for both shared and distributed memory systems (explained below).

- Hybrid models of the above.

After an exhaustive analysis of the problem to suit the development of the parallel program, by using the Foster's methodology and choosing a parallel algorithm model, the developer has to start designing the solution. To do that, it is required to know some design aspects and concepts, which are typically used in parallel programming. Below we summarize some of them [1], and the most used along the report:

- Identifying the parts of the program where working in parallel:
  - *Profiling*: identify the parts where the majority of the work is taking place.
  - *Bottleneck*: identify the parts where the execution is slow, and downgrading the whole performance of the program.
  - *Task*: each part to be executed in parallel is assigned to a task. The parallel execution of each task is performed independently of one another, and could be running in an uncoordinated manner.
  - *Granularity*: it is a qualitative measure of the ratio of computation to communication. The range has two extremes: *coarse grain* (relatively large amounts of computational work are done between communication events) and *fine grain* (relatively small amounts of computational work are done between communication events).
  - *Parallel overhead*: it is the required time for coordinating the tasks. It can include factors such as: task start-up time, synchronizations, data communications, some software overhead imposed by compilers, operating system, etc., and task termination time.
  - *Load balancing*: it is the process of distributing work among tasks so that all tasks are kept busy all the time.

- Factors to consider when designing the required communication among the parallel tasks:
  - If there is no communication, the solution can be *embarrassingly parallel.*
  - *Latency*: the minimal necessary time to send a data from the producer to the receiver.
  - *Bandwidth*: the amount of data that can be sent in a period of time (e.g. MBytes/sec.). We can state that, assuming that l is the interconnection latency (e.g. in seconds), and b is the bandwidth (e.g. bytes per second), the time to transmit n bytes is $l + \frac{n}{b}$.
  - *Synchronous communication*: if the tasks need to coordinate in some points of the computation and work in unison with shared data. On the other side, *asynchronous communications* allow tasks to transfer data independently from one another, at any time. Normally, this kind of communication implies to use data *buffers.*
  - *Shared memory*: when all the processors share the main memory system, so they can access the same data.
  - *Message passing*: when the processors has independent (distributed) memory systems, they normally communicate through message passing, either in synchronous or in asynchronous manner.
- When the communication is synchronous, it is necessary to know some more on synchronization processes. Essentially, it is a way to perform task coordination:
  - *Race condition*: if two tasks attempt to update a common resource, the accesses can result in an error. Specifically, the final state of the resource depends on which task "wins the race" (one can overwrite the result of the other), whereas both results has to be added to the resource, independently of the access order.
  - *Critical section*: a piece of code that can only be executed by one task. A parallel programmer has to insure the mutually exclusive access to critical sections.
  - *Lock*: Also called mutex. It is used to protect and serialize the access to global data or a critical section of a program. Only one task can unlock the mutex, and the rest has to wait to the other

to lock it again. Mutex enforces serialization, and the way to implement it is by busy-waiting (the tasks are continuously running), or by blocking (the tasks go to "sleep"). Finally, semaphores and monitors are higher-level types of locks.

- *Barrier*: every involved task works until reaching the barrier. When a task reach a barrier, it stops or "blocks", until the last involved task reaches the barrier.
- *Atomic operations*: they are a special type of locks which implies to perform a specific operation over a global data in a safe manner.

3.1.5 Parallel platforms types

In order to effectively run a parallel program, we must use *parallel platforms.* These platforms make mention of technologies that contain multiple interconnected processors (at least more than one). Some examples of parallel platforms are presented in the next section. We usually call parallel *architecture* the internal structure of these platforms. However, we often use parallel architectures and parallel platforms as similar terms. Moreover, we can also speak about performance on parallel platforms, indicating how good are them handling parallelism, by the communication speed and the computational speed of each independent processor.

From now on, we will consider that the performance of a parallel program depends on the way it has been programmed, and on the parallel platform where it is executed. Normally, in order to get as close as the best speedup, developers study first which is the specific parallel platform where the program is going to run. Then, they adopt the specific design and implementation considerations to optimize the execution of the program.

We will detail below a dichotomy based on the logical and physical organization of parallel platforms [73]. The logical organization is the one that the programmer can handle, while the physical organization is the real hardware. For the latter, we will show some examples in a next section. For the former, we will focus on the two critical components for a parallel program developer: the control structure (expression of parallel tasks) and communication models (interaction between tasks).

Parallel architectures can be classified, depending on the control structure, in several ways. The most used since 1996 [120, 1] is the *Flynn's taxonomy.* The classification is according to the number of instruction streams and the number of data streams simultaneously managed:

- *SISD (Single Instruction, Single Data)*: only one instruction stream is managed by the CPU, using also one stream of data. This corresponds to a serial (non-parallel) computer. This is the oldest and most common model of computer. For example, some modern PCs are still based on this.

- *SIMD (Single Instruction, Multiple Data)*: it is a type of parallel computer that has one instruction stream shared for all the processors. However, each processor may have its own data stream, being able to operate over different data elements in parallel. This type is well suited for data-parallel based programs. We can see nowadays this model, for example, on vector processors and graphics processing units, as mentioned later in this chapter.

- *MISD (Multiple Instruction, Single Data)*: this is a not so common parallel computer, having multiple processors with independent instruction streams, but working over the same data stream. Few actual examples of this have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).

- *MIMD (Multiple Instruction, Multiple Data)*: it is another type of parallel computer. Each processor may have its own instruction stream, working also with different data streams. It is the most common type of parallel computers, and it is noteworthy that typically, MIMD parallel computers require more hardware than SIMD computers (SIMD has only one global control unit). A simple variant of this model, called *SPMD (Single Program, Multiple Data)*, is based on executing several instances of the same program over different data. SPMD model has the same expressiveness as the MIMD model (MIMD can be translated to SPMD using if-else constructions). It is therefore the most widely parallel model used in parallel platforms.

The other classification of parallel platforms that we are going to introduce is the one based on the communication model for MIMD systems. There are two primary forms of data exchange between parallel tasks: by sharing or distributing the memory. The classification is performed depending on the way it is done:

- *Distributed memory* systems: each processor has its own private memory, and the communication is made via an interconnection network. This communication is explicitly done by sending messages or by special functions that access to the memory of other processors. There are several standards that allow to manage distributed memory systems, such as *MPI* [4]. The performance of these systems also relies on the topology of the interconnection network (ring, toroidal mesh, hypercube, butterfly, etc.).

- *Shared memory* systems: the processors are connected to a memory system through an interconnection network, also called here a *bus*. The interaction of the parallel tasks is implemented directly on this shared memory. There exists several standards providing specific implementations for parallel programming over shared memory systems. The most used are the POSIX threads (*Pthreads*) and *OpenMP* [15]. There are also two different implementations of shared memory systems:

  - *UMA (Uniform Memory Access)*: the time taken to access any data element in the memory is the same for all the processors. This is normally achieved by directly connecting all the processors to a same main memory.
  - *NUMA (Nonuniform Memory Access)*: the time taken to access a data element can be longer for some processors compared to others. The reason is that the hardware is implementing a distributed memory system, but this is transparent for the processors. NUMA systems dedicate different main memory banks to different processors, but the communication is made fast through the bus. Some coherence protocols are necessary to insure the same data state for all the processors. The main advantage of this shared memory system is that has the potential to use larger amounts of memory than UMA systems. A typical usage of NUMA systems is the *SMP (symmetric multiprocessing)* architectures, which is the base of the majority of supercomputers today.

3.2 Parallel platforms

Parallel programs must make use of parallel platforms in order to effectively run in parallel. Details on the architecture and current examples of parallel architectures are introduced in this section.

3.2.1 Processes implementation in operating systems

The structure of a parallel program is described in this section to better understand how they are executed on parallel platforms. First of all, we emphasize that current parallel technologies are based on the *Von Neumann*'s architecture model. It consists of a *main memory*, a *CPU (Central Processing Unit)* (also known as processor or core), and an *interconnection* between the memory, the CPU and input/output peripherals [120].

Each position of the memory is addressed, storing both instructions and data. The CPU is divided into a *control unit* and an *ALU* (arithmetic and logic unit), which performs operations over fast-accessed data stored in the CPU (registers). The program counter is a special register used by the control unit in order to know the next instruction to be executed. The CPU works according to a *clock*, so that everything can be measured on clock cycles (accesses to registers, memory, ALU operations, etc.). The speed of a CPU is normally measured by the number of clock cycles per second, or *Hz (Hertz)*. The interconnection between the CPU and the memory is normally implemented by a bus (consisting on a set of parallel wires of fast access), and it allows the CPU to read (or fetch) and write (or store) instructions and data from the memory. The bus normally conforms the bottleneck of the system, but depending on the speed of the CPU and of the main memory.

These basic ingredients appears in the majority of today parallel platforms. However, how are they used? In fact, the *OS (Operating System)* is the software that takes control of the hardware and software levels. It controls the execution of programs, the accesses to memory, and the access to peripheral devices.

We can say that a *program* is a code (collection of instructions and data) stored in hard disk. It is created from a *source code* (in a specific programming language) by using a *compiler*. When a user runs a program, the operating system creates a *process*, which instances the program on main memory. The process contains the following entities: the executable code (the program itself), a block of memory (divided in three parts: the code, a call stack of active functions and a heap of dynamic memory), resource descriptors (e.g. for input/outputs), security information and state information of the process (number identification (ID), is blocked, etc.).

We should explain a bit more precisely how a process uses its block of memory. First of all, all the constants values are directly stored in the program code. The value of each single variable is stored in the process. Finally, the process can also use large amounts of memory associated to array or matrix based variables. This can be performed in two ways: by *static* or *dynamic* memory allocation. The former refers to allocate memory at compile-time before the associated program is executed. The later refers to allocate memory as required at run-time. Sometimes, when using dynamic memory to allocate the required memory for the algorithm just at the beginning of the execution, it is also called static memory.

Most modern operating systems are multiuser (many user can access the operating system at the same time), and multitasking (it can run many processes concurrently). As we mentioned before, parallel programs cooperate in order to solve a problem. By multitasking, it can be done by communicating the processes, through hard disk, networks connections, etc. However, the most common usage of parallelism is by *threading*. A thread is an independent task within a process. Therefore, a process can run several threads in parallel (concurrently and fast cooperating by sharing the same block of memory). They are explicitly encoded in the program by using thread library standards (e.g. Pthreads, OpenMP [15], etc.). Normally, the above mentioned execution of parallel tasks are actually implemented by threads within a process, but they can be also implemented by different processes (running in the same machine by multitasking, or in different machines) communicated through any kind of protocol (e.g. message passing (MPI)).

3.2.2 Memory hierarchy

Another critical part of a parallel platform is the memory system. In the most classical parallel platforms, the memory system is implemented as a hierarchy of subsystems, where the top is the fastest memory (but smaller), and the bottom is the lowest (but larger) memory. Let start detailing the common memory hierarchy, from the bottom to the top.

At the bottom of the hierarchy, we find the called *secondary memory* or *secondary storage* system. It is a large system (nowadays of the order of GigaBytes/TeraBytes), but relatively slow (measured in MegaBytes/second). It consists of one or more rigid (hence "hard") rapidly rotating discs (platters) coated with magnetic material, and with magnetic heads arranged to read-/write data to the surfaces. We can still find other solutions for the secondary memory, such as the magnetic tape (an older and slower solution), and the solid-state drive (newer and faster solution). The programs and all the related data for the execution (configuration and data files) are stored at this level.

The *primary memory*, *primary storage* or *main memory* is placed at the middle of the hierarchy. It is a relatively big system (of the order of some

GigaBytes) and fast access (GigaBytes/second). It is the only one directly accessible from the CPU. The CPU continuously reads instructions from there, and modify the data. The processes of the operating system are stored here. Historically, early computers used delay lines, Williams tubes, or rotating magnetic drums as primary storage. Nowadays, computers make use of modern *random-access memory (RAM)*. It is also volatile, i.e. they lose the information when not powered.

Current operating systems permit a process to require more memory than the available in main memory. To permit this, they use a *virtual memory* system. This system stores the recently accessed parts of a process in main memory, and the rest on secondary memory. The address space of virtual memory can be (and usually is) higher than the main memory. If a position corresponding to an address is not in main memory, then that portion of data is exchanged with the secondary memory. This process is automatically and transparently performed (at the operating system level), and it is commonly named *pagination*.

If we compare the speed of the CPU (e.g. 1 ns clock) and main memory (e.g. 100 ns latency), we see that the latter is not able to give sufficient instructions and data to maintain the CPU always busy: main memory is slow compared to the CPU. A method to solve this bottleneck is the *caching* technique. Cache memory is actually at the second level of the memory hierarchy. It is quickly accessed by the CPU, but is very small (around KiloBytes (KB) or some MegaBytes (MB)). A cache can be divided itself in a hierarchy of levels, again from the fastest but smallest to slowest but largest cache memories. Normally they have at most three levels (level 1 (L1 cache), level 2 (L2 cache) and level 3 (L3 cache)). L1 cache is typically divided into two different parts, one for instructions and other for data. L2 and L3 are also typically mixed memories (instructions plus data), but it depends on the implementation of the manufacturer.

The main idea of caching is to take advantage of *locality*. It is a principle referring that, in most programs, the access of one data element is followed by an access of a nearby element (spatial locality), and also again in a near future (temporal locality). In other words, a cache is a fast but small memory that automatically stores blocks of memory containing recently accessed elements and their nearby ones.

On the other side, there exists many parallel platforms having another kind of memory at the second level of the hierarchy, called *scratchpad* memory. Scratchpad is often confused with cache (it is fast accessed and small sized), but they are not the same. The usage is manually made by the program, and explicitly defined by the programmer. The goal is to allow the programmer to exploit other properties that may exist in its program different from spatial and temporal locality.

The *registers* within the CPU are at the top of the memory hierarchy. A register is a single data element of a static size (from 32 to 64 bits, depending on the implementation) that are accessed very quickly by the CPU (in just one clock cycle). They are a few compared to cache and main memory, but all the data participating in an operation has to be previously stored in a register. The usage of registers, cache and main memory are normally automatically performed by the compiler and the hardware. In other systems, the use of them has to be explicitly done.

3.2.3 Real parallelism implementation

Next we will detail how parallelism is actually implemented in parallel platforms. Historically, it has been gradually implemented in hardware at different levels of abstraction: not all the parallel platforms are implemented only by a set of independent processors. They are based on taking advantage of three parallelism levels: instructions, threads and processes.

The *Instruction-Level Parallelism (ILP)* is a kind of implicit parallelism to achieve better performance in processors. The idea is to replicate multiple processor components (functional units, ALU, etc.) to simultaneously execute instructions. There are two main approaches to ILP, both used in modern processors. We will use two measurement factors in order to clarify the differences of the two approaches. Firstly, the typical way to measure the performance of a single processor is by using the *CPI (Cycles per Instruction)*, which says the number of CPU cycles required to issue an instruction. Secondly, for parallel processors, the performance measurement is based on *IPC (Instructions Per Cycle)*, which indicates the number of instructions that can be issued in a single CPU cycle..

- *Pipelining*: it enables faster execution by overlapping various stages in instruction execution. These stage varies in the implementation, but generally are: fetch (the control unit of the CPU retrieve the instruction information), decode (access to the operands), execute (perform the operation) and store (the result). These stages can be divided into more smaller stages, so that having smaller tasks enables faster clock rates and more stage overlap. The ideal performance is to achieve a CPI equals to 1. However, several issues make it impossible: e.g. every fifth to sixth instruction is a branch (jump instruction). Long instruction pipelines therefore need prediction techniques for branch instructions to feed the pipeline (the penalty of a misprediction can be disastrous). More issues, such as race conditions, appears also at this level.

- *Multiple issue*: it consider a processor with two or more pipelines, with the ability to simultaneously issue and execute several instructions. The goal is to downscale the CPI, so now the IPC is used here. If we have k pipelines in a multiple issue processor, the best performance is to achieve a IPC of k. But again, this is not possible always (the above mentioned issues are also a problem here). There are actually two approaches for this kind of processors:
  - *Dynamic multiple issue*: also called *superscalar* processor. The hardware dynamically search for instructions that can be issued at the same cycle to the different pipelines. Its performance is limited by the available instruction level parallelism. The hardware logic for dynamic dependency analysis is typically in the range of 5-10% [73], and it becomes a bottleneck when increasing the number of pipelines.
  - *Static multiple issue*: also called *VLIW (Very Long Instruction Word)* processor. The idea is to rely on the compiler to resolve dependencies and resource availability at compile time. The compiler creates a code where the instructions are sorted in a way that the hardware has only to issue k consecutive instructions at each clock cycle. Therefore, hardware logic is saved. The performance of VLIW processors depends directly to the compilers' ability to parallelize the instructions.

ILP is usually difficult to exploit, since it depends directly on the independence of instructions of a program. At the next level, we find the *TLP (Thread-Level Parallelism)*. In this case, the hardware (or the compiler) is not the responsible of finding and managing parallelism. Here, the programmer has to rewrite the program in order to allow to execute several flows of instructions (threads). Thus, a TLP processors provide parallelism through the simultaneous execution of different threads. TLP is typically implemented in two ways on hardware multithreading [120]:

- *Fine-grained multithreading*: the processor switches between threads after each instruction, avoiding stalled (blocked) threads. The problem is that it does not allow a thread to execute a full sequence of instructions, when it is possible to do it. An example of this is *SMT (Simultaenous MultiThreading)*, which is a variation exploiting superscalar processors to execute multiple threads.

- *Coarse-grained multithreading*: the processor switches the execution of a thread only when it gets stalled. However, the processor can be idled for shorter stalls. Some multicore processors implement this.

Finally, the coarsest-grained level of parallelism relies between processes. It can be easily noted that processes has their own memory, and they run independently of each other. Therefore, a processor can run a process at a time. There are several ways to do it:

- *Multicore* processors: they are processors that has several cores (CPUs) working in parallel. As they are installed within the same chip, they normally share the low-level of cache (L2 or L3), and the access to main memory. Therefore, they are suitable to both executing processes and threads in parallel. This kind of architecture is also called *CMP* (Chip Multi-Processor).

- *Multisocket* processors: they are several processors that are plugged into the same machine. They can be also multicore. The access to the main memory is normally implemented as separate bus, but they use the same memory system.

- *Network* of processors: the most scalable way to implement parallelism is to use several different machines with their own processors interconnected by a network. The processes can run in parallel in the different machines, and eventually communicate through a protocol, such as message passing.

3.2.4 Current parallel platforms for HPC

3.2.4.1 Current multicore processors

A performance limitation was reached for traditional sequential processor chips. The main reasons are that the clock frequency can not be increased (for energy consumption problems), and it is not possible to extract more ILP from codes. As mentioned before, the way to achieve better utilization of the processor chip is to move from the strategy of incorporating multiple processors (cores) on the same chip [60]. The term ‘processor’ is therefore ambiguous: it can refer to either the chip, or the processor core on the chip. For this reason, we mostly talk about a socket (or connector) for the whole chip and core for part containing one CPU [60].

Current multicore processors implements the three levels of parallelism introduced above: ILP, TLP and PLP. Depending on the implementation, the parallelism level is differently performed. For example, the two main high-end processor manufactures, Intel and AMD, implements TLP by SMT (Hyper-threading) and coarse-grained solutions (multicore), respectively. Moreover, the memory hierarchy, at the cache level, may also be different among the processors.

A dual-core processor has two cores (e.g. AMD Phenom II X2, Intel Core Duo), a quad-core processor contains four cores (e.g. AMD Phenom II X4, Intel’s quad-core processors for i3, i5, and i7 at Intel Core), a hexa-core processor contains six cores (e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X), and an octa-core processor contains eight cores (e.g. Intel Xeon E7-2820, AMD FX-8150). Homogeneous multicore systems include only identical cores, heterogeneous multi-core systems have cores with different features. Just as with single-processor systems, cores in multi-core systems may implement architectures such as superscalar, VLIW, vector processing, SIMD, or SMT.

3.2.4.2 Parallelism in networks

Probably the most adopted way to implement parallelism is the based on computer networks. They likely constitute the most scalable and extensible solution today. These networks can be formed from commodity personal computers to high-end computers. The size of these networks can vary from local networks (located in the same building) to more global networks as the Internet (the global network of networks). The way they are physically interconnected depends on the topology (ring, hypercube, etc.). But we need a protocol defining how computers (nodes) communicate with each other.

Solutions based on global networks (and also for some local networks) are usually based on one of the two next protocol models:

- *Server-client*: a node has the role of the server, which distributes and takes control of the computation among the client nodes.

- *Peer-to-peer*: all the nodes have the same role, and each one takes control of a portion of computation.

Thus, networks has been successfully applied for HPC. An example of this is *Grid Computing.* A grid is a system that "*coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service*" [68]. Grid Computing is therefore a form of Distributed Computing whereby a "super (virtual) computer" is composed of many networked loosely coupled computers acting together to perform very large tasks. It can be an agglomeration of networks from different international enterprises (departmental and enterprise grids), or just a large group of volunteers that "borrow" their computers for the specific purpose of the grid (also known as CPU scavenging for global grids). An example of the latter is the BOINC project, which is operating at 5.945 PFLOPS as of August 6, 2012, as shown in the official website [10]. Research projects, such as Folding@Home, MilkyWay@Home, SETI@Home and Einstein@Home are well-known examples of the BOINC usage for Grid Computing.

Grid Computing is normally used for specific purposes. If one would like to solve its own problem, he might pay to use a grid solution, or to construct his own one. An alternative for this today is to "hire" for computing. *Cloud Computing* denotes a model on which a computing infrastructure is viewed as a "cloud", accessed from anywhere, and offering computing, storage, and software "as a service." [153]. The services provided by Cloud Computing are divided into three classes: Infrastructure as a Service (IaaS, offering virtualized resources), Platform as a Service (PaaS, offering environments for creating and deploying applications), and Software as a Service (SaaS, offering the usage of specific sofware).

At a local level, networks can be used for HPC as *computer clusters.* Clusters are aggregations of processors in parallel configurations [138]. The control of resource allocation, scheduling and users management is made by a centralized node. Each node run its own operating system instance. The network is local, so all the nodes are normally packed into the same place. Moreover, some grids are collections of clusters (e.G. NSF Tera grid). Perhaps the most known cluster management system software is the Sun Grid Engine (SGE), now called Oracle Grid Engine. It offers a complete knowledge of user requests and system status.

Today, clusters are also used for supercomputing. A supercomputer, as mentioned before, is a computer at the frontline of current processing capacity. Most current supercomputers are clusters developed to achieve the best performance. Others are based on grids. The ranking of today most powerful supercomputers is the Top500 [18]. The information is produced bi-annually, and the HPL (a portable implementation of the High-Performance LINPACK) is used as benchmark.

3.2.4.3 Accelerators

A new trend in HPC is the performance improvement of single computing nodes by using *accelerators*. HPC accelerators are inexpensive massively parallel computing chips that perform specific functions faster than using software over general-purpose CPUs. They tend to deliver hundreds of special cores, and they are programmed differently than common CPUs. Accelerators are managed under the Heterogeneous Computing. The CPU is the master device which takes control of the computation, and delivers the work to be performed in parallel by the accelerator.

They therefore act as fast co-processors to help the CPU in certain computations. When applications fit this heterogeneous programming model and the functions implemented in the accelerator, they can run 10 to 100 times faster than on standard multicore processors. A run-time speedup of at least 5 to 10 times is often needed to justify programming heterogeneous architectures. Accelerators are advancing rapidly, in both the architecture and the programming languages.

Many hardware accelerators are built on top of *FPGAs (Field-Programmable Gate Array Chips)*. They are digital integrated circuits (ICs) that contain configurable interconnects between these blocks [108]. They first arrived at the mid-1980s. FPGAs can be configured or programmed to perform a large variety of tasks: digital signal processing, embedded microcontrollers, physical layer communications, and reconfigurable computing (to accelerate more general-purpose applications). The way to program them is through *hardware description languages* (HDL), such as VHDL or Verlilog. There are also high level languages, such as Handel-C o Mitrion-C, that allow to program FPGAs in a easier manner, but not as efficiently as HDLs.

More specific purpose accelerators are those designed for multimedia processing acceleration. The most common example is the vectorial co-processor. It is included in every current commodity CPU. They can operate on vectors or arrays of data using special instructions, while the CPUs operate on scalar data elements [120]. Some examples of them are the technologies MMX and SSE introduced by Intel, or 3DNow of AMD.

Another well-known multimedia accelerator is the *Cell-BE (Cell Broadband Engine)* processor [90]. It was jointly developed by Sony, Toshiba and IBM. The Cell-BE is designed to bridge the gap between conventional processors and more specialized high-performance processors, such as the graphics processors.

Its enhanced SIMD architecture achieves better performance and power efficiency than traditional CPUs. The heterogeneous multicore architecture of the Cell-BE is formed by one Power Processor Element (PPE), which is "the brain" of the chip, and 8 to 9 Synergistic Processing Elements (SPE), which are "the muscles". They are interconnected by the high-speed Element Interconnect Bus (EIB). We can find this processor in many multimedia devices, such as Home Cinemas, Bluray readers and the Playstation 3. It is also used in many supercomputers in order to enhance the performance of the nodes, such as the Blade servers of IBM. It is also going to be the replacement of some nodes in the spanish supercomputer Mare Nostrum.

Finally, a new solution of HPC accelerator is the usage of *GPUs (Graphics Processor Units)* [119, 89], sometimes called VPU (Visual Processing Unit). The term was popularized by NVIDIA in 1999, who marketed the GeForce 256 as "the world's first GPU". It is a specialized chip designed to rapidly manipulate and alter memory for accelerating the construction of frames or images to output in a display. Modern GPUs are very efficient at manipulating computer graphics. Their highly parallel structure is based on hundreds of simple computing cores, making them more effective than general-purpose CPUs for processing large blocks of data in parallel. Moreover, many software tools have appeared until now, providing high level languages for developers. For all mentioned above, the GPU is starting to be consolidated as a fast HPC accelerator.

3.3 GPU computing

With the commercial sector's demands for video and gaming, it was foreseen by Elster [61] and others that graphics processor development would lead to devices suitable for High Performance Computing (HPC). The era of GPGPU (General-Purpose Computing on Graphics Processing Units) truly began with the introduction of NVIDIA's CUDA [89, 62] and AMD's Stream SDK environments in 2007, so that the GPUs became more easily programmable. GPUs can now be considered affordable computing solutions for speeding up computationally demanding applications (*GPU computing*).

In this section the evolution of the GPU is introduced, and its usage for general purpose applications. Then, the CUDA programming model is detailed, which is the one used along the work. Finally, an overview of the GPU characteristics and current graphics cards is provided.

3.3.1 Evolution of the GPU

In the progress of graphics computing, the CPU was the responsible for processing all instructions in order to draw the output on the screen. This entailed an overload with the increased complexity of the graphic calculation. The first attempt to get a CPU with optimized graphics calculations was the vectorial processor in the 70's. This processor worked with stream data (continuous data source), which is the nature of the graphical data. In this way, if an unit A needs the result of an operation performed by B, it is not necessary for A to wait for B computes all elements. Unit A can start working with the elements from the stream that B has already processed. The key is the independence between the elements of a stream, what allow to overlap the functional units in the architecture. This idea is used today in the above mentioned vectorial co-processors, such as MMX, SSE and 3DNow.

The first video cards were dedicated only to control the screen by refreshing and painting *pixels*[1] according to the information received from the CPU. The GPU (Graphics Processing Unit) [56] was first created and placed within the video card to alleviate the bottleneck created by the CPU. The GPU is a special purpose processor, optimized to work with graphics as stream data. Note that the CPU communicates with the GPU through a data bridge (Northbridge), which is also connected to the main system memory. Currently, the technology used by modern GPUs is the PCI Express bus, which runs at 16GB per second with double bandwidth.

The most complex graphics calculations are those based on real-time three-dimensional scenes processing (e.g. 3D video games). Therefore, the graphics pipeline of the GPU has been driven by the 3D-market. The aim is to transform the 3D data (coordinates of the triangles in the model, easily understood by programmers) in pixels drawn on a 2D screen.

Figure 3.1 shows a 3D graphics pipeline overview [56]. The input of the pipeline is the coordinates of 3D objects (mainly composed of triangles). The coordinates are formed from the triangle vertices. They enter the *Geometry* stage, where the 3D coordinates are transformed to 2D coordinates (fragments). In the last stage, called *Rendering*, the final color of each pixel is calculated based on the information associated to each fragment. Some properties associated to fragments are the color, luminescence, texture, alpha effects (transparency) and fog effects. As shown in Figure 3.1, the Geometry stage is divided into two sub-stages. The first one is called *Transformation and Lighting*, which is the process of projecting objects in three dimensions to

[1]A point with a single color in a raster image (bitmap).

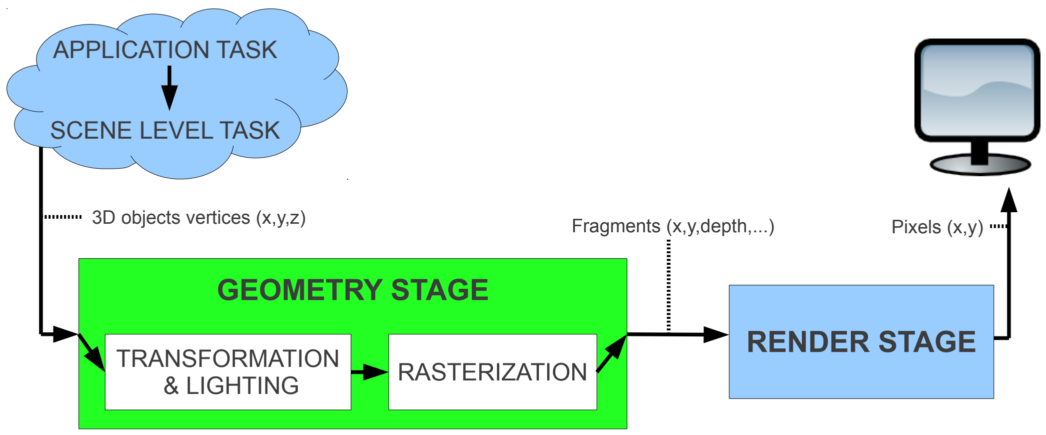


Figure 3.1: General graphics pipeline, from the application to the screen.

two dimensions, and calculate the effects of lights in the scene. The second stage, *Rasterization* (also known as Triangle Setup), set and cut the triangles, making pieces (fragments) for the Rendering stage. Note that a fragment is not a pixel (although sometimes they do not differ, as in the specification of Direct3D). A fragment has attributes such as color and depth. Multiple fragments may correspond to a pixel (for example, when using transparency). The tasks at the application level (artificial intelligence, camera, interaction, etc.) and scene-level tasks (collision, physics, etc.) are always carried out by the CPU, as they are general-purpose tasks and not necessarily graphics.

However, not all the above mentioned stages were initially implemented inside the GPU. The first graphics card drew only lines and dots on the screen, so that the rest of the work was performed by the CPU. By transistor downscaling, it was possible to implement, progressively, stages of the graphics pipeline on the GPU while the CPU was working with more general tasks. Thus, at the beginning, the rendering stage was implemented on the GPU. As a result of this work delegation, the CPU was able to send fragments to the GPU so fast, that the GPU became a bottleneck.

Therefore, two solutions were adopted: pipelined and parallel architecture. Several processors were implemented in parallel for the Rendering stage, since the work to be done with each pixel was the same at any time. Thus, the GPU began to be a SIMD processor (same instructions applied to different data) specialized for graphics. After these changes, the CPU and the access to 3D data in memory became the bottleneck, so the solution was to give more workload to the GPU and improve the memory bandwidth by incorporating specific memory close to the GPU chip.

The GPU finally implemented on-chip all the stages of the pipeline (Geometry and Rendering). This first GPU architecture was a *fixed function pipeline*. Considering Figure 3.1, and the decomposition of each stage, we can consider that the full fixed graphics pipeline is generically formed by the following stages [119] (the stages are still general, and depending on the implementation, each GPU may implement a pipeline with similar stages to these ones):

1. Vertex Operations: it performs transformations (e.g. lighting) to vertices.

2. Primitive Assembly: it creates triangles.

3. Rasterization: it sets and cuts triangles into fragments.

4. Fragment Operations: it calculates colors for fragments according to their properties.

5. Composition: it creates the final image formed by pixels.

Although this was the most efficient solution, the behavior of the GPU was fixed, performing always the same operations on graphical data. The pipeline was not flexible for programmers. The available operations at the Vertex and Fragment stages were configurable but not programmable [119]. For instance, changing the graphics API[2], implementing new light models for vertices or providing new transformations for fragments were not possible.

In 2002, the adopted solution was to develop a programmable pipeline, where the GPU could run small programs called *shaders* on vertices and fragments. A shader is a piece of code that programs certain parts of the graphics pipeline. There are two types: *vertex shader* (which replaces the full Transformation and Lighting stage) and *fragment shader* (replacing the full Rendering stage). Shaders was initially programmed using low level languages, such as ARB Vertex Program, ARB Fragment Program and Direct3D 9 Shading language. However, there was some simple high level languages, such as Cg and GLSlang.

Recall that the pipeline architecture was divided in time by the CPU. However, the GPU divided it in space. Each stage has its own unit. The pipeline has a large latency, but a high throughput [119]. The former refers

[2]Application Programming Interface

that the GPU takes several cycles to perform one operation over a data element. However, by implementing data parallelism inside each stage, the GPU delivers high throughput.

The major disadvantage of the GPU pipeline is load balancing. The slowest stage limits the full performance of the pipeline. Vertex shader is more complex than fragment shader. Moreover, it is shown that vertex shader and fragment shader units are used in different proportions at different times; so there are times when many fragment units are free while there are not enough for vertex, and vice versa. In order to solve this problem, the *unified shader unit* was introduced. These units are able to execute code both for vertex and fragment shaders. An unique language shader (for vertex and fragments) was also introduced to improve the performance [111]. AMD introduced the first unified shader unit in its Xenos GPU for the XBox 360. Today, current GPU architectures consist of a number of unified shader units disposed in parallel, sharing multiple graphics pipelines. This new unit is the key of the rapid development of GPGPU: programmers can now target directly this programmable unit [119].

3.3.2 General-Purpose Computing on the GPU

The evolution of the unified shader units has led to a new graphics-based computing power with a highly parallel nature: a GPU has from 16 to 1000 shader units. The GPU has been therefore considered also to run applications rather than graphics. It is capable of accelerating applications with the following characteristics [119]: large computational requirements (e.g. real-time applications), substantial parallelism (take advantage of the multiple fine-grained programmable compute units), and being the throughput more important than latency (latency of any individual operation is unimportant for human visual system). Given the successes in accelerating applications using the GPU, Mark Harris, now a researcher at NVIDIA, coined the term *GPGPU (General-Purpose computing on the GPU)* [13] in 2002. Today, this effort, also known as *GPU computing*, has positioned the GPU as one of the most powerful and cheap accelerators in HPC.

The GPU, as a HPC accelerator, has its own programming model. It is based on the SPMD programming model: the GPU processes many independent elements (no dependencies with each other) in parallel using the same program [119]. Each element can operate on 32-bit integer or floating-point data, that can be read (*gather*) from a global memory. The newest GPU can also write back to arbitrary locations in memory (*scatter*).

This programming model is well suited for SIMD based parallel programs, but current shader units permit a more general SPMD model. They allow different elements to take different paths. As the GPU demotes more hardware to computation than to control, incoherent branching has a penalty. The solution was to group elements in blocks, so that these blocks are processed in parallel if the take the same branch (blocks are treated in SIMD manner). The size of the block is called "branch granularity", and it has been reducing during the evolution of the GPU. This concept will be used later in the CUDA *warps*.

The GPGPU has evolved together with the GPU programmability [119]. Initially, when programming the GPU for graphics, developers had to think on the geometry of each region, the shaded fragments and graphics buffers. Therefore, a GPGPU programmer had to adapt its problem to graphic data, and apply operations by translating them to shaders. This is also known as the *old* GPGPU. Examples of these first applications were for protein folding, SQL requests and MRI reconstruction. These programs were developed using standard shading languages, such as Microsoft *HLSL*, NVIDIA *Cg* or OpenGL *GLSL*.

The difficulty of writing GPGPU applications was solved by introducing a more natural, direct, non-graphics interface to the GPU hardware. It was performed by abstracting the GPU as a stream processor [119]. *BrookGPU* and *Sh* were the two early academic research projects doing this. They were the starting point of a *new* era of GPGPU. Programmers directly define SPMD general-purpose programs using threads. These threads can compute many math operations, and both gather and scatter to global memory. However, the parallelism has to be defined explicitly in the program, considering some restrictions on the computing elements. The hardware then only cares on exploiting that explicit parallelism to achieve a good performance. This programming model provides a careful balance between generality (general-purpose), and restrictions to ensure performance (SPMD model, branch granularity, data communication, etc.).

Both BrookGPU and Sh, as mentioned above, introduces a stream programming model. A stream program comprises a set of streams (ordered sets of data), and kernels (functions applied to each element of streams). The codes (based on C programming language[88]) were mapped directly to shaders. However, they were quite basic and restrictive. Some more evolved but commercial languages appeared later, based in the same idea, such as Microsoft's *Accelerator* (with just-in-time compilation to shader), *RapidMind* (targeting several platforms, including GPUs, Cell and multicore CPUs), and *PeakStream* (providing profiling and debugging support).

In 2006, AMD introduced its own GPGPU environment, called *CTM (Close-to-Metal)*, which provides a low-level hardware abstraction layer (HAL) and a compute abstraction layer (CAL) to the programmer. They allow to program the GPU in both low and high levels. Moreover, they permit the direct compilation of Brook programs to their hardware.

In 2007, NVIDIA announced *CUDA (Compute Unified Device Architecture)*, a programming model totally abstracted from the hardware of the GPU. The main difference with the previous languages is that CUDA introduces two levels of parallelism (data parallel and multithreading), multiple levels of memory hierarchy, and basic synchronization among threads. Based on C, the programmer only has to think on threads and arrays, together with performance aspects such as a memory hierarchy and branching (explained in the next sections). This easy way to build large applications has led to a rapidly evolution of GPGPU until today.

Finally, it should be noted that the current trend nowadays is the creation of a standard for programming heterogeneous systems [89], to accelerate the codes in any GPU on the market (even intended to be generic for any parallel technology: Multi-CPUs, FPGAs, CellBE, etc.). This trend is being consolidated with OpenCL [112, 6], the first free, open standard for multi-platform parallel programming of modern processors found in PCs, servers and embedded devices. The two major companies of graphics cards, AMD and NVIDIA, supports (compilers, development kits and documentation) OpenCL for their devices.

3.3.3 CUDA programming model

In this section, we will introduce the CUDA programming model [89, 118, 62, 5] which is the one used in our work. CUDA works only for NVIDIA's GPUs, so in the next section, the modern NVIDIA GPU architecture will be also explained.

The CUDA programming model is based on heterogeneous computing: the system consists of a *host*, which is a traditional CPU, and one or more *devices*, which are massively parallel processors, such as a GPU.

In many modern software applications, there are program sections which exhibit a rich amount of data parallelism. CUDA devices take advantage of them, and accelerate their execution harvesting a large amount of data parallelism. A CUDA program therefore consists of one or more phases that are executed either in the host or in device. Sequential and control phases are implemented in the host code, while phases which exhibit a large amount of data parallelism are implemented in the device code.

A CUDA program is a unified source code covering both sides. NVIDIA C compiler (called *nvcc*) separates them during the compilation process. The host code is treated as ANSI C/C++ code [88], so it is compiled by a standard C/C++ compiler (like GNU *gcc*) and runs on an ordinary CPU process. The device code is written using extended C with special keywords to label the data parallel functions, called *kernels*, and their associated data structures. The device code is usually compiled by *nvcc*, and runs on the GPU device.

The kernel functions (or simply kernel) typically generate a large number of threads that exploit the data parallelism. CUDA threads are much lighter than CPU threads. A CUDA programmer can assume that these threads take a few cycles to be generated and scheduled. This contrasts with the threads of the CPU, which normally require thousands of clock cycles to be managed.

The CUDA program execution is illustrated in Figure 3.2. Execution always starts in the host. When a kernel is invoked, execution moves to the device, where a large number of threads are generated to take advantage of abundant data parallelism. All threads that are generated by a kernel are collectively called *grid*. When all threads complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

Typically, a grid is composed of thousands of threads, since the creation of a sufficient number of threads to use all hardware resources requires a large amount of data parallelism. The threads are arranged within the grid in a two-level hierarchy, as illustrated in Figure 3.2. At the higher level, each grid consists of one or more *thread blocks*. At the lower level, each block is organized as a three dimensional array of threads. All blocks in a grid have the same number and organization of threads. Each block is identified by a two dimensional identifier, and each thread within its block by a three dimensional identifier. A thread block can contain, at most, 512 threads.

CUDA allows threads within a block coordinate their activities using a barrier synchronization function. When a thread executes this function, it remains blocked until the other threads of the same block execute it. Threads of different blocks can only be synchronized by terminating the kernel, so that all the threads in the grid finish the execution.

The memory model is also an aspect to consider in the CUDA programming model. This memory hierarchy has to be explicitly and manually managed. In CUDA, the host and the devices have separate memory spaces. This reflects that, in fact, graphics cards are hardware devices that come with their own

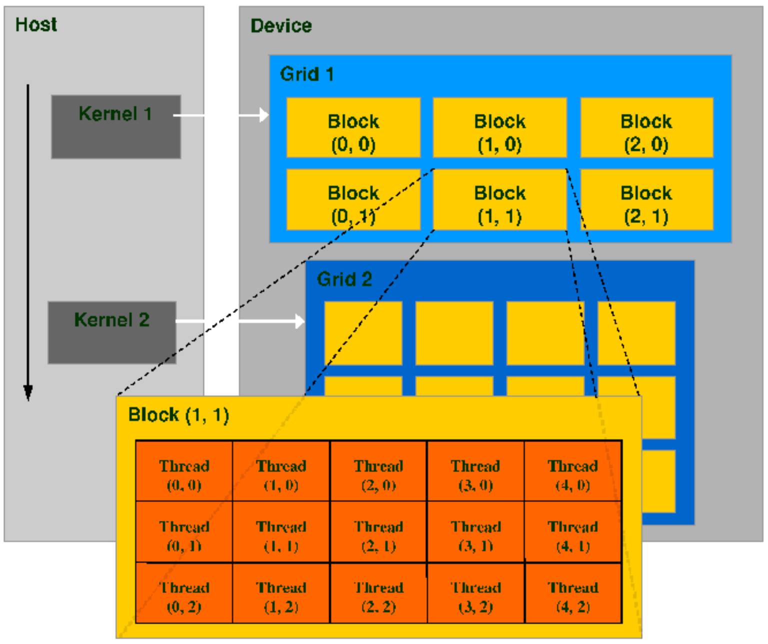


Figure 3.2: CUDA thread execution model

DRAM (Dynamic Random Access Memory). In order to run a kernel on the device, the programmer needs to allocate memory on it, and transfer the relevant data from the host to the reserved memory in device. Similarly, after the execution on the device, the programmer needs to transfer the resulting data from memory to the host device, and release the device memory. The CUDA runtime system API provides functions to perform these activities by the programmer. From this point of view, we can assume that CUDA uses static memory allocations. Dynamic memory is only supported in the newest GPUs of NVIDIA, with some restrictions (the maximum amount of dynamic memory to use has to be previously allocated).

Figure 3.3 shows a summary of the CUDA memory model. At the bottom of the memory hierarchy we found *global* and *constant* memories. These are the memories to which the host can transfer data code in a bidirectional way, as seen in the double arrows in the figure. Constant memory allows read-only access to the device code. This memory is cacheable (i.e., frequent consecutive accesses will be made faster). It is often used to store constant and common variables to all threads. From now on, we focus on the use of global memory.

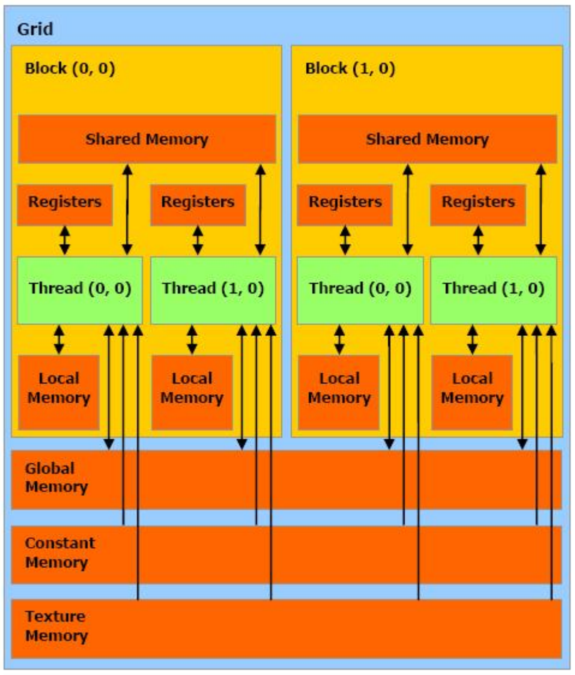


Figure 3.3: CUDA memory model

Threads can both read and write on this memory, but it is not cacheable. The input data and the results of the device computation is normally stored here. Note that the host memory is not shown explicitly in the figure, but is assumed to be in the CPU side.

At the top of the hierarchy, CUDA provides a small portion of *shared* memory for each block of threads. All blocks have access to it, as fast as the access to *registers* (storing only single variables). However, the host can not access to this memory. It is used to hide global memory access latency, which limits application performance. Shared memory is a scratchpad memory, so that the programmer has to explicitly use it. Finally, a small piece of global memory can also be privatively reserved to each thread. It is called *local memory*.

An efficient way to structure an algorithm is as follows:

1. The threads of each block read its corresponding data portion from global memory to shared memory (which is inevitable because the host only can put the data in global memory).

2. Threads work with the data directly on the shared memory.

3. Threads copy these data back to global memory (so the host can retrieve the result).

A well-known strategy in parallel programming, and used also in CUDA, is *tiling*. This strategy seeks to combine the previous structure of three phases with partitioning data, so that the three phases are repeated for each data portion (or *tile*), minimizing accesses to global memory.

As mentioned before, threads from different blocks cannot cooperate directly, but only through the global memory and using a special set of atomic operations. These operations are implemented by implicit locks, so that accesses to desired data elements can be efficiently synchronized through them. However, this is restricted to the use of a small set of operations (see Table 3.1).

The main keywords and language elements added by CUDA into C are summarized in Table 3.1. An example is also attached in an extra column to better understand how they are used.

CUDA comes with a large software support for developers. It is of a huge importance, so that programmers can use them to develop their CUDA based applications:

- Toolkit: the driver, *nvcc* compiler and many other useful tools, such as the debugger (cuda-gdb), the profiler (cuda-prof), etc.
- SDK: many examples and extra tools.
- Libraries: there are libraries to make an easier use of CUDA, and others for specific problem domains (linear algebra, bioinformatics, random number generation, etc.). All of them are developed by the CUDA community.
- Plugins: there are plugins for Microsoft Visual Studio, and for Eclipse (in CUDA version 5).

As stated above, there are tools for debugging and profiling, what GPU computing has lacked so far. Firstly, the *nvcc* compiler handles all parts of the compilation flow, trying to hide the compilation details from developers and giving a wide range of compiler options. There are several compiler flags that are really useful in certain parts of the development process. For the debug purpose there is an emulation mode which is enabled with the compiler flag *-g* (this flag generates debuggable code). Furthermore, the compiler has other flags which are focused on optimizing the CUDA code. These flags are *-ptx -cubin*.

The *PTX* (Parallel Thread Execution) code is an assembly-like representation which is produced by the *nvcc* compiler whenever a CUDA code is compiled with the `-ptx` flag enabled. It is optimized by the CUDA runtime to get hardware-specific binaries for execution. Notice that PTX code is not the code which executes on the GPU, but it gives an approximated idea of the execution.

Additionally, the *nvcc* compiler has the `-cubin` flag which produces a .cubin file. This file contains information about occupancy of each multiprocessor in the GPU. It also shows the number of registers per thread, the amount of shared memory used by a thread block, whether the kernel is using local memory or not, and finally, the binary code of the application.

Other software tools have been created to support CUDA programmers and ease the CUDA development cycle, such as *CudaVisualProfiler* or *decuda* [5]. The former is a quite useful tool to profile your CUDA code. The latter is a disassembler for the NVIDIA CUDA binary (.cubin) format and it helps to identify bottlenecks showing the internal instructions generated for the G8x and G9x architectures.

3.3.4 Modern GPU architecture: NVIDIA's G200 as case study

The GPU used in our work is the NVIDIA Tesla C1060. Therefore, this section introduces the Tesla C1060 (G200) computing architecture, and it shows architecture parameters that can affect the performance. In addition, the threading model of Tesla architectures is analyzed, together with the most important issues in the CUDA programming environment.

The Tesla C1060 [94] is based on a scalable processor array which has 240 *SPs* (streaming-processor) cores organized as 30 *SMs* (streaming multiprocessor) and 4 GB of off-chip GDDR3 memory called device memory. The applications start at the host side which communicates with the device side through a bus, which is a PCI Express x16 bus standard (see Figure 3.4). PCI Express delivers up to 4 GB/sec of peak bandwidth per direction, and up to 8 GB/s of concurrent bandwidth.

The SM is the processing unit and an unified graphics and computing multiprocessor. Every SM contains the following units: eight SPs arithmetic cores, one double precision unit, an instruction cache, a read only constant cache, 16-Kbyte on-chip read/write shared memory, a set of 16384 32-bit registers, and

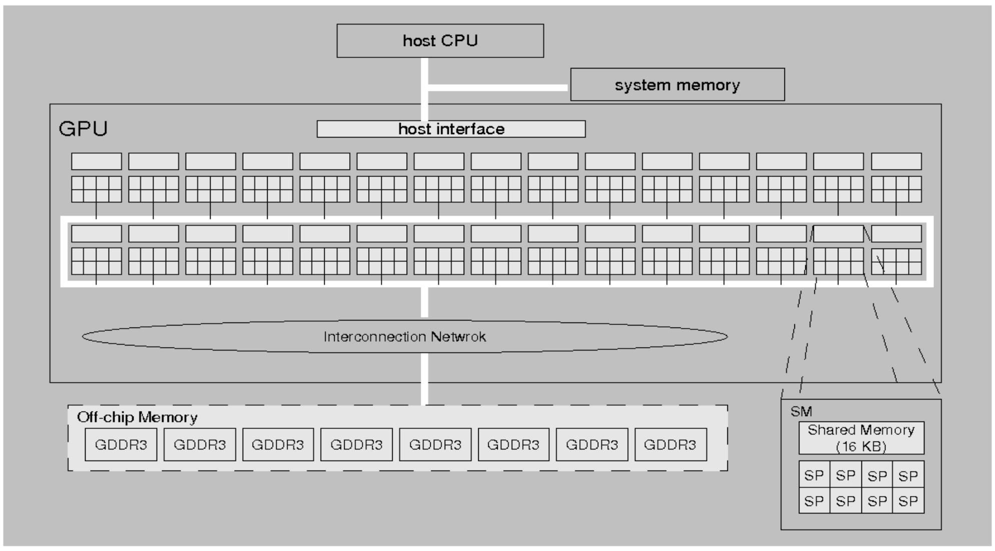


Figure 3.4: Tesla T10 unified architecture, based on G200.

access to the off-chip memory (device/local memory). The SM also has two SFUs that execute more complex floating point operations such as reciprocal square root, sine or cosine with low latency. The arithmetic units are capable to execute three instructions per clock cycle, and they are fully pipelined, running at 1,296 GHz, yielding a peak theoretical 933 GFLOPS[3] (240 SP * 3 instructions * 1,296 GHz).

The local and global (device) memory spaces are not cached, which means that every memory access to global memory (or local memory) generates an explicit memory access. A multiprocessor takes 4 clock cycles to issue one memory instruction. Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency [5], that is more expensive than accessing share memory and registers (only the mentioned 4 cycles).

The Tesla C1060 achieves 102 GB/sec of bandwidth to the off-chip memory (running at 800 MHz). This bandwidth is not enough for the big set of cores and the possibilities to sature it are high. It is needed to coalesce accesses to the device memory for obtaining the maximum bandwidth available. The coalesced accesses are obtained whenever the accesses are contiguous 16-word lines, otherwise a fraction of this bandwidth is obtained. Coalesced accesses will be a critical point in the optimization process.

[3]FLOPS stands for *FLoating-point Operations Per Second.* GFLOPS are giga FLOPS.

In addition, the threads can use other memories like constant memory or texture memory. Reading from constant cache is as fast as reading from registers, as long as all threads in the same warp read the same address. Texture Memory is optimized for 2D spatial locality (see Table 3.2).

A SM is a hardware device specifically designed with multithreaded capabilities. Each SM manages and executes up to 1024 threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. The SMs execute threads in a *SIMT (Single-Instruction Multiple-Thread)* fashion [94]. Basically, in the SIMT model all the threads execute the same instruction on different piece of data. SMs create, manage, schedule and execute threads in groups of 32 threads (which is the branching granularity of NVIDIA GPUs). This set of 32 threads is called *warp*. Each SM can handle up to 32 warps (1024 threads in total, see Table 3.3). Individual threads of the same warp must be of the same type and start together at the same program address, but they are free to branch and execute independently.

The execution flow starts with a set of warps ready to be selected. The instruction unit, which is ready for issue and executing instructions, selects one of them. The SM maps all the threads in an active warp to the SP cores, and each thread executes independently with its own instructions and register state. Some threads of the active warp can be inactive due to branching or predication, and this is a critical point in the optimization process. The maximum performance is achieved when all the threads in an active warp takes the same path (the same execution flow). If the threads of a warp diverge, the warp serially executes each taken branch path, disabling threads that are not on that path. When all the paths complete, the threads re-converge to the original execution path.

3.3.5 Performance considerations

In this section we survey some performance considerations when programming GPUs. It is important to understand that the implemented code for the GPU has to be premeditated. If it is done abruptly, the application will be not adequately accelerated.

First of all, there are data parallel operations that are well suited for GPU computing. In fact, the success on accelerating many applications lies in translating parts of the algorithm to one of these operations. Some algorithms and applications that have been accelerated via this operations are sorting, search and database queries, differential equations and linear algebra. We list below four data parallel operations that are programming idioms long familiar to parallel computing users, and used now as computational primitives in GPU computing [119]:

- *Scatter/gather*: a type of memory addressing. Historically, GPUs allows efficient reading (gather through cached memories), but writing (scatter) is normally more slow (even more when it requires data synchronization through atomic accesses). Sometimes, it is better to let the threads to write each output value through reading many input data (gather), rather than assigning an input element to each thread that will write in several arbitrary positions in memory, what requires synchronization (scatter). Variants of these types are the *stencil* and *transpose* patterns.
- *Map*: apply an operation to all elements in a collection. It is typically expressed as a for loop in sequential code, and it can be performed easily in parallel. It is a good example of data parallelism.
- *Reduce*: repeatedly apply a binary associative operation to reduce a collection of elements to a single value (e.g. sum, average, minimum, etc.). It can be performed in parallel by iteratively reduce the number of threads working over the collection. It also requires to synchronize the threads in every step.
- *Scan* (also known as parallel-prefix-sum): take an array A and return an array B of the same length, such that each element $B[i]$ represents a reduction of the sub-array $A[1 \dots i]$. Applications of this type can be applied to, for example, quicksort or sparse matrix operations.

Finally, a characterization of what GPUs do well would allow us to define more efficient algorithms [119]. Next we provide four characterizations that we will consider in our algorithms, and that are of a large importance in GPU computing. It is the best way to understand how the GPU works:

- *Emphasize parallelism*: GPUs prefer to run thousand of lightweight threads to maximize opportunities to mask memory latency and blocking. The algorithms should permit divide the computation into many independent pieces. For this purpose, it is necessary to reduce the resources assigned to each thread, and try to avoid synchronization.
- *Minimize branch divergence*: in CUDA, the warp is the unit of parallelism. A warp is executed in parallel, but the warps belonging to the

same thread block are serially executed. If a warp is broken because divergence, then there is no real parallelism (only along the different thread blocks). Note that a large branching granularity has more possibilities to be broken, but a small one leads to a low real parallelism.

- *Maximize arithmetic intensity*: computation is relatively cheap for today GPUs, but bandwidth is precious. It is better to maximize the computational operations per memory transaction. Shared memory or registers help for this purpose.

- *Exploit streaming bandwidth*: however, GPUs and their on-board memory (GDDR3) have a peak bandwidth 10x faster than CPUs and DRAM. It is achieved by streaming memory access patterns: coalesced access to aligned memory positions. A good way to maximize the bandwidth in an algorithm is by the scatter/gather strategy.

3.3.6 GPUs today and in future

Given the importance of GPU computing today, the GPU hardware architecture and software are continually evolving. For example, both main GPU manufactures, AMD and NVIDIA, are working on providing full double precision floating-point hardware. The bandwidth is also improving by technologies such as PCI Express 2 and HyperTransport. AMD has also developed Fusion, a microprocessor technology having the CPU and the GPU together in the same chip. This technology has less bandwidth with memory (it uses DDR3 DRAM, and not GDDR3), but avoids the transfer of memory. Other authors also propose the creation of a fully programmable graphics pipeline, being the pipeline itself as programmable as shader units.

We have presented the NVIDIA Tesla C1060, since it is the graphics card we have used in our work. This card appeared in the market in 2008. However, we can find newer GPU technologies today, such as NVIDIA Fermi and Kepler, providing a huge amount of cores (up to 512 for Fermi G400, and 1536 for Kepler G600), cacheable global memory and better bandwidth. See Table 3.4 for more detailed hardware specifications about the three generations of the NVIDIA Tesla cards brand. Moreover, CUDA is (at the date of this work) in its 5^{th} version, but we have used CUDA version 4.

AMD also provides today high-end GPUs, such as the Southern Islands based systems, having up to 2048 cores. They are now programmed by using OpenCL, which is standard and high level language, very similar to CUDA.

Table 3.1: CUDA language elements and keywords

Keyword	**Usage**	**Meaning**	**Example**
__global__	function qualifier	A kernel function (it is called from the CPU and executed on the GPU)	__global__ void myKernel(int a) {...}
__device__	function qualifier	The device function (it is called from the GPU and executed on the GPU)	__device__ int myDeviceAux-Function(int a) {...}
<<<a,b,c>>>	function call	The kernel function call, configuring the number of blocks (a), number of threads within blocks (b) and, optionally, a reserved number of bytes in shared memory (c)	<<<dimGrid,dimBlocks>>> myKernel(0);
__shared__	variable qualifier	The variable is stored in the shared memory	__shared__ int a;
__constant__	variable qualifier	The variable is stored in the constant memory	__shared__ int a;
__local__	variable qualifier	The variable is stored in the local memory	__shared__ int a;
threadIdx.x threadIdx.y threadIdx.z	built-in variable	Coordinate x of the thread identifier Coordinate y of the thread identifier Coordinate z of the thread identifier	myArray[threadIdx.x] [threadIdx.y]=0;
blockIdx.x blockIdx.y	built-in variable	Coordinate x of the block identifier Coordinate y of the block identifier	if (blockIdx.x==0) z=0;
blockDim.x blockDim.y blockDim.z	built-in variable	Size of the block in the x dimension Size of the block in the y dimension Size of the block in the z dimension	myArray[threadIdx.y* blockDim.x+threadIdx.x]=0;
gridDim.x gridDim.y	built-in variable	Size of the grid in the x dimension Size of the grid in the y dimension	myArray2[blockDim.y* gridDim.x+blockIdx.x]=0;
__syncthreads();	built-in function	Barrier synchronization of threads belonging to the same block	__syncthreads();
cudaMalloc(...);	built-in function	Executed in the host, it reserves memory in device memory	cudaMalloc(&ptr,numBytes);
cudaMemcpy(...);	built-in function	Executed in the host, it copies memory bidirectionally between the host and device memories. It takes 4 parameters: destination pointer, source pointer, number of bytes to copy, and the involved memories (cudaMemcpyHostToDe-vice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, cud-aMemcpyHostToHost)	cudaMemcpy(d_ptr,h_ptr,...);

atomicAdd(...)
atomicSub(...)
atomicExch(...)
 	
atomicMin(...)
atomicMax(...)
atomicInc(...)
atomicDec(...)
atomicCAS(...)
 	
atomicAnd(...)
atomicOr(...)
atomicXor(...)
	built-in function	Atomic operation for addition
		Atomic operation for subtraction
		Atomic operation performing value exchange
		Atomic operation for minimums
		Atomic operation for maximums
		Atomic operation for incrementation
		Atomic operation for decrementing
		Atomic op. performing compare and swap
		Atomic operation for bits and
		Atomic operation for bits or
		Atomic operation for bits xor
			atomicAdd(g_mem_ptr, 4);
			atomicSub(g_mem_ptr, 4);
			old=atomicExch(g_mem_ptr, newValue);
			atomicMin(g_mem_ptr, 0);
			atomicMax(g_mem_ptr, 99);
			old=atomicInc(g_mem_ptr);
			old=atomicDec(g_mem_ptr);
			old=atomicCAS(g_mem_ptr, 0, 5);
			atomicAnd(g_mem_ptr, 0xFF);
			atomicOr(g_mem_ptr, 0x10);
			atomicXor(g_mem_ptr, 0xC1);

Table 3.2: Memory system on the Tesla C1060

Memory	**Location**	**Size**	**Latency**	**Access**
Registers	On-Chip	16384 32-bits Registers per SM	$\simeq$ 1 cycles	R/W
Shared Memory	On-Chip	16 KB per SM	$\simeq registers$	R/W
Constant	On-Chip	64 KB	$\simeq registers$	R
Texture	On-Chip	Up to Global	> 100 cycles	R
Local	Off-Chip	4 GB	400-600 cycles	R/W
Global	Off-Chip	4 GB	400-600 cycles	R/W

Table 3.3: Major hardware and software limitations programming on CUDA

Configuration Parameters	**(Maximum) Value**
Threads/SM	1024
Thread Blocks/SM	8
32-bit Registers/SM	16384
Shared Memory/SM	16 KB
Threads/Block	512
Threads/Warp	32
Warps/SM	32

Table 3.4: Hardware features for the Teslas C1060, M2050 and K10 GPUs.

GPU element	**Feature**	**Tesla C1060**	**Tesla M2050**	**Tesla K10**
Streaming processors (GPU cores)	Cores per multiprocessor	8	32	192(SMX)
	Number of multiprocessors	30	14	2x8(16)
	Total number of cores	240	448	2x1536(3072)
	Clock frequency	1296 MHz	1147 MHz	745 MHz
Maximum number of threads	Per multiprocessor	1024	1536	2048
	Per block	512	1024	1024
	Per warp	32	32	32
SRAM memory available per multiprocessor	32-bit registers	16 K	32 K	64 K
	Shared memory	16 KB	16 KB or 48 KB	16 KB or 48 KB
	L1 cache	No	48 KB or 16 KB	48 KB or 16 KB
	Total SRAM (shared + L1)	16 KB	64 KB	64 KB
Global (video) memory	Size	4 GB	3 GB	2x4(8) GB
	Speed	2x800 MHz	2x1500 MHz	2x2500 MHz
	Width	512 bits	384 bits	256 bits
	Bandwidth	102 GB/sc	144 GB/sc	320 GB/sc
	Technology	GDDR3 DRAM	GDDR5 DRAM	GDDR5 DRAM

Part II

Parallel Simulation applied to Efficient Solutions of Computationally Hard Problems

"The advent of the accelerators in High Performance Computing offers fresh avenues for developing new and efficient simulators for P systems and Systems Biology."

J.M. Cecilia, G.D. Guerrero, J.M. García, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez

4 Parallel Simulation of P systems with Active Membranes

In order to experimentally validate a P system based model, it is necessary to have simulators running on electronic computers. They would help researchers to compute, analyze and extract results from a model [57]. These simulators have to be as efficient as possible to handle instances of large size. This is one of the main problems with current simulators for P systems. Software applications for Membrane Computing normally implement sequential (or parallel with relatively few threads) simulation algorithms adapted to common CPU architectures [57], so they do not take advantage of the massively parallel nature that P systems present by their definition.

This parallel computation model leads us to look for a highly parallel computational technology where a parallel simulator can run efficiently. The newest generation of GPUs, as mentioned before, are massively parallel processors which can support several thousand concurrent threads. To date, many general purpose applications have been ported to these platforms obtaining good speedups compared to their corresponding sequential versions [146, 147].

In this chapter, we highlight the necessity to use a parallel architecture which improves the efficiency of P systems simulators. For this purpose, we present a parallel simulator for the class of recognizer P systems with active membranes using CUDA [41, 39, 44, 105, 102, 74, 45, 106], due to the fact that in this theoretical model, the creation of an exponential workspace, expressed in terms of number of membranes and objects, in linear time takes place in a natural way. The simulator receives as input a P system which is defined and translated into a binary file using the pLinguaCore [70, 69]. The simulation algorithm is divided into two main stages: selection stage and execution stage. Both phases are implemented on the GPU, so the entire simulation executes all the computations in different membranes in a parallel way.

We also provide a performance test of the simulator with a family of P systems that exploit the intrinsic parallelism of P systems and demonstrate that GPUs are better suited than CPUs to simulate P systems as long as the problem instance size increases.

4.1 Simulation algorithm

The simulator we have developed is based on the sequential simulator for P systems with active membranes provided in pLinguaCore [70]. In this design, the simulation process is a loop divided into two stages: *selection stage* and *execution stage* (see Figure 4.1). The selection stage consists in the search for rules to be executed in each membrane of a given configuration. The selected rules are executed at the execution stage, what finalizes the simulation of a computation step (or transition).

The input data for the selection stage contains the description of the membranes with their multisets (strings over the working alphabet of objects, labels associated with the membrane, etc.), and the set of defined rules. The output data of this stage are the multisets of selected rules. Only the execution stage changes the information of the configuration. It is the reason why execution stage needs synchronization when accessing to the membrane structure and the multisets.

At the end of the execution stage, the simulation process restarts the selection stage in an iterative way until a halting configuration is reached. This stop condition is twofold: a certain number of iterations or a final configuration is reached. On one hand, we define a maximum number of iterations at the beginning of the simulation. On the other hand, a halting configuration is obtained when there are no more rules to select at selection stage. As previously explained, the halting configuration is always reached since it is a simulator for recognizer P systems.

Non-determinism affects the selection stage, when there are more than one selectable rule but only one can be executed. For example, two evolution rules

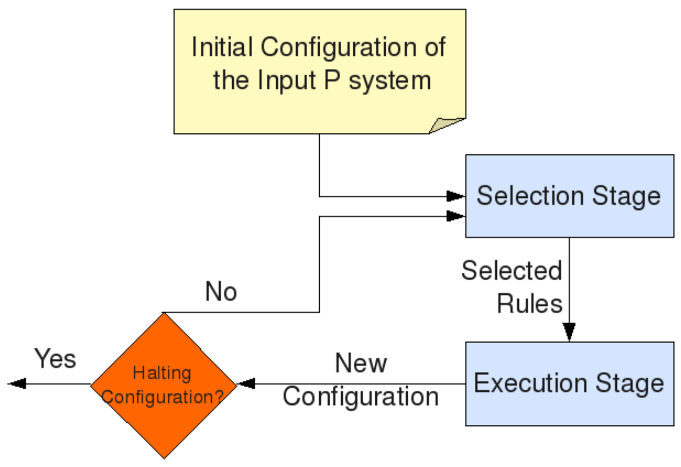


Figure 4.1: Iterative process of the simulation algorithm for P systems with active membranes.

that can be executed using the same object, a division rule and a send-in rule that can be selected in the same membrane at the same time, etc. In order to avoid non-determinism somehow, the simulator assumes only confluent P systems. Thus, instead of working with the entire tree of possible computations, the simulator selects and simulates only one computation path, since all paths are guaranteed to give the same answer. We can take advantage of this property by selecting path using the lowest cost rules. We will measure this cost in number of membranes and synchronization operations. These are the conditions that could damage the simulation performance the most. In this context, we introduce the following priorities among rules in the selection stage:

1. *Dissolution rules*: they decrease the number of membranes (highest priority);

2. *Evolution rules*: they do not need any communication among membranes (which avoids synchronization);

3. *Send-out rules*: they do need communication between the given membrane and its parent (adding one object to its parent);

4. *Send-in rules*: they do need communication between the given membrane

and its parent (reserving one object from its parent and adding the object to itself);

5. *Division rules*: they increase the number of membranes (lowest priority).

During the execution stage, the information of the system can vary by including new objects inside membranes, dissolving membranes, dividing membranes, etc., obtaining a new configuration. This new configuration will be the input data for the selection stage of the next iteration.

Finally, note that this two-staged algorithm allows to keep a coherence in the simulation. If we perform selection and execution of rules, one by one, it would be difficult to ensure the semantic constraints of the system. Moreover, the selected and executed rules in a step of the simulator may not correspond to the rules applied in a computing step of the theoretical model. An alternative solution might be to take two copies of the configuration, one to be updated with the right-hand sides of the rules, and another to select rules (subtracting the left-hand sides of rules). As this involves a bigger use of memory, our simulator uses the two stages, and a temporary data structure to store information of the selection of rules.

4.2 Sequential simulation in C++

As previously mentioned, CUDA programming model [5] is based on the C/C++ language [88]. Therefore, the first recommended step when developing applications in CUDA is to start from a baseline algorithm written in C++, identifying the parts that can be susceptible to be parallelized on the GPU. In this work, we have based on the simulator for P systems with active membranes developed in pLinguaCore [70]. This sequential (or single-threaded) simulator is programmed in JAVA, so the first step was to translate the code to C++.

Our first sequential simulator implements the structure of membranes by using C++ pointers and dynamic memory allocations. Each membrane stores a pointer to its parent, a pointer to the first of its children, another pointer to one of its brothers (having the same parent membrane), the charge, and the multiset of objects. The multiset of objects is also implemented by a (dynamic) linked list based on pointers. Each object in the multiset stores its multiplicity (if zero, it is deleted to save memory space) and a pointer to the next object. Therefore, memory spaces for membranes and objects are created and deleted "on demand". The rules of the system are statically stored, so that we can

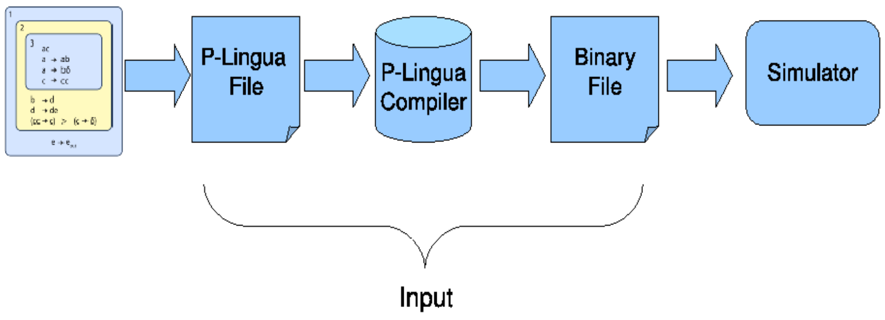


Figure 4.2: Generation of the input for the simulator

easily access to the rules associated to each membrane, by using its label and charge. Furthermore, the multiset of selected rules is also implemented using a dynamic linked list.

The common problem here is the competition for objects between different membranes. In this case, internal membranes applying send-in rules are competing for the objects in the parent. We loop the tree from the top to the bottom, so the top level membranes have more priority using its objects than internal membranes using send-in rules.

The input of the simulator (the P system with active membranes to simulate) is given by a binary file. It is a file whose information is encoded in Bytes and bits (not understandable by humans like plain text), which is suitable for compressing data. This binary file contains all the information of the P system (alphabet, labels, rules, etc.) which is the input of our simulator. The format is depicted in Section 4.3. pLinguaCore 2.0 [70] is able to translate a P system written in P-Lingua language into a binary file. We therefore use a pipeline of applications to simulate P systems, as shown in 4.2. First, we define our P system into P-Lingua. pLinguaCore translate it to a binary file, which is used as the input of the simulator. The output is a plain text generated with a format similar to the provided in pLinguaCore.

After experimentally validating our sequential simulator with different examples, we have developed the parallel simulator as an extension of it. The simulation always starts with the sequential simulator, and when a threshold number of membranes is created, the execution is changed to the parallel version. In this way, we make use of the parallel simulator only when it worths it.

After creating the parallel simulator, we also have developed an adapted sequential simulator for the CPU (called *fast sequential* simulator), which has the same constraints as the CUDA simulator explained in the next subsections, to make a fair comparison among them. This simulator achieves much better performance than the original sequential simulator, since it uses only static memory (saving time in memory management). It is noteworthy that this simulator has been improved in successive versions. The current version is faster than the one presented in [41, 44], since it has been optimized to better utilize cache memory by internalizing some loops over the membranes.

The full framework of simulators for P systems with active membranes, including the sequential, fast sequential and CUDA parallel simulators, is called *PCUDA*. It is a subproject of the PMCGPU project, and can be downloaded from the website: `http://sourceforge.net/p/pmcgpu` [17]. More information about the simulator, how to install and use it, refer to Appendix A.

4.3 Input binary file format

In this section we show the definition of the binary format of the input file for our simulator. It specifies the number of bytes is used for each required element, and comments with explanations are given by '#'. Note that it is thought for compressing the information of the P system.

The format is structured in 3 main parts:

- A header storing general information about the file (version, etc.).
- The P system general information: alphabet, membrane structure (plus initial charges) and initial multisets.
- The rules of the P system.

```
#########################
## Format of the binary input file for the simulator (revision 13-12-2008)
#########################

Header (4 Bytes):
0xAF
0x12
0xFA
# Last Byte: 4 bits for P system variant, 4 bits for file version
0x11

Number of objects in the alphabet (2 Bytes) # if greater than 0xFF, 2 Bytes
                                            # for objects identifiers (ID)
# For each object (identifier implicitly given by the order)
Text string representing the object (finished by '\0')

Number of different labels of membranes (2 Bytes) # if greater than 0xFF,
                                                  # 2 Bytes for labels ID
# For each label (identifier implicitly given by the order)
Text string representing the label (finished by '\0')

Number of membranes (2 Bytes) # if greater than 0xFF, 2 Bytes for
                              # membranes ID
# For each membrane (identifier implicitly given by the order)
# The first membrane is the skin, with identifier 0
ID of parent membrane (1 or 2 Bytes)
ID of label (1 or 2 Bytes)
Charge (1 Byte, 0 = neutral, 1 = positive, 2 = negative)

Number of initial multisets (2 Bytes)
# For each multiset:
ID of the membrane (1 or 2 Bytes)
Number of different objects in the membrane (2 Bytes)
# For each object:
ID of the object (1 or 2 Bytes)
Multiplicity (2 Bytes)

Number of evolution rules (2 Bytes)
# For each rule:
ID of the label (1 or 2 Bytes)
Charge (1 Byte)
ID of the object in the left-hand side of the rule (1 or 2 Bytes)
Number of different objects in the right-hand side of the rule (2 Bytes)
# For each object:
ID of the object (1 or 2 Bytes)
Multiplicity (2 Bytes)

Number of send-in rules (2 Bytes)
# For each rule:
ID of the label (1 or 2 Bytes)
Charge (1 Byte)
New charge (1 Byte)
ID of the object in the left-hand side of the rule (1 or 2 Bytes)
ID of the object in the right-hand side of the rule (1 or 2 Bytes)

Number of send-out rules (2 Bytes)
# For each rule:
ID of the label (1 or 2 Bytes)
Charge (1 Byte)
New charge (1 Byte)
```

```
ID of the object in the left-hand side of the rule (1 or 2 Bytes)
ID of the object in the right-hand side of the rule (1 or 2 Bytes)

Number of dissolution rules (2 Bytes)
# For each rule:
ID of the label (1 or 2 Bytes)
Charge (1 Byte)
ID of the object in the left-hand side of the rule (1 or 2 Bytes)
ID of the object in the right-hand side of the rule (1 or 2 Bytes)

Number of division rules (2 Bytes)
# For each rule:
ID of the label (1 or 2 Bytes)
Charge (1 Byte)
New charge (1 Byte)
New charge for new membrane (1 Byte)
ID of the object in the left-hand side of the rule (1 or 2 Bytes)
ID of the object in the right-hand side of the rule (1 or 2 Bytes)
ID of the object in the right-hand side of the rule for new membrane (1 or
                                                                   2 Bytes)
```

4.4 Parallel simulation on CUDA

Whenever we design algorithms in the CUDA programming model, our main effort is dividing the required work into processing pieces, which have to be processed by *TB* thread blocks of *T* threads each. Using a thread block size of *T=256*, it is empirically determined to obtain the overall best performance on the Tesla C1060 [147]. Each thread block access to one different set of input data, and assigns a single or small constant number of input elements to each thread.

As mentioned in Chapter 3, each thread block can be considered independent to the other, and it is at this level at which internal communication (among threads) is cheap using explicit barriers to synchronize, and external communication (among blocks) becomes expensive, since global synchronization can only be achieved by the barrier implicit between successive kernel calls. The need of global synchronization in our designs requires successive kernel calls even to the same kernel.

Figure 4.3 shows the overall design of the simulator that we have implemented on the GPU. We distribute the thread blocks and threads as follows. Each membrane of the simulated P system is attributed to each thread block. In this way, we identify the parallelism between membranes by using the par-

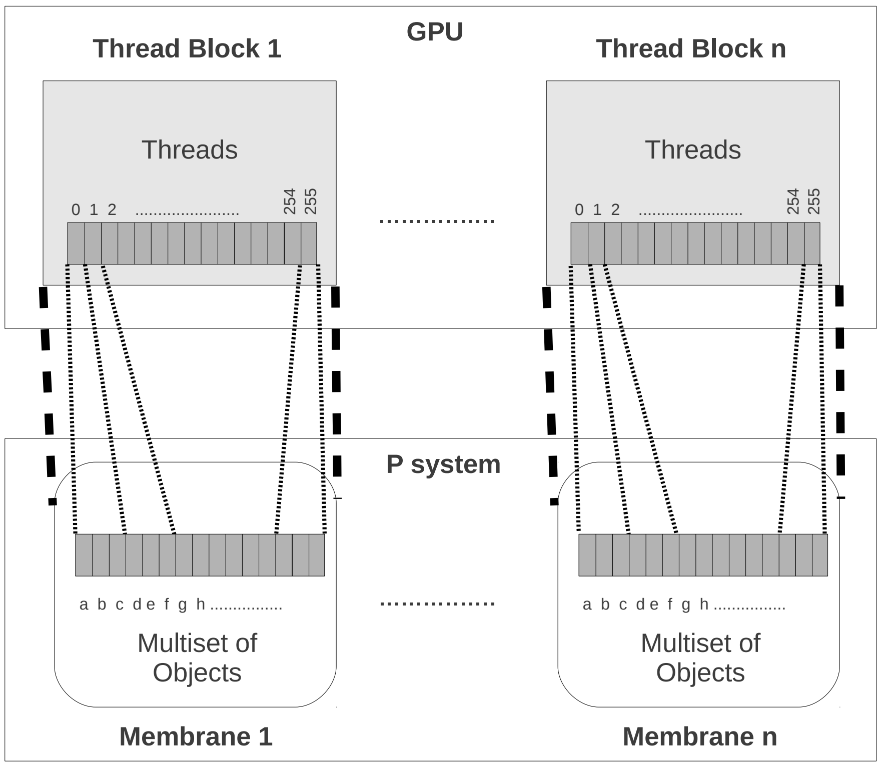


Figure 4.3: Basic design of the parallel simulator on the GPU.

allelism between thread blocks. We must take precautions with this design decision. Membranes can communicate accordingly to the hierarchical tree structure, while thread blocks are all independent. Communication through send-out and dissolution rules (down-up direction) is controlled by globally synchronizing the selection and execution stages. This is implemented by using different kernels. However, send-in rules (up-down direction in the tree) are more complicated to control. In this case, different membranes can compete for single objects. The sequential simulator controls this issue by looping the tree from the top to the bottom. However, the parallel simulator has to run all the membranes in parallel. Therefore, for the sake of simplicity, the parallel simulator can handle only two levels of membrane hierarchy: the skin (controlled by the host) and the rest of elementary membranes (controlled by the thread blocks in device). This is the tree structure we can find in the literature for the majority of solutions based on P systems with active membranes (note that division rules enlarge the tree widthwise) [133].

Furthermore, each individual thread is assigned to each object within a membrane (corresponding to its thread block). It is responsible for identifying the rules that can be executed using the corresponding object. That is, rules that have that object in their left-hand sides. Since all blocks must have the same number of threads, and each membrane can contain a different multiset of objects in every time step, we identify as common for all membranes the whole alphabet. Note that threads can work with many objects that do not really exist in the membrane, as all the alphabet of objects is usually not present within a membrane at a given instant. In fact, the simulator assigns multiple objects to the same thread for not restricting the number of objects in the alphabet. However, the number of objects in the alphabet must be divisible by a number smaller than 512 (the maximum number of threads per thread block), in order to equally distribute the objects among the threads.

The simulator contains five kernels to implement the selection and execution stages. The first kernel implements the selection stage and also the execution stage for evolution rules. The other four kernels implement the other execution rules (dissolution, division, send-out and send-in rules). All the kernels follow this basic design. The selection kernel starts with the selection stage. After the selection stage, we also execute in this kernel the evolution rules. These rules are executed inside this kernel for three main reasons: the evolution rules do not imply communication (and therefore, synchronization) among membranes; they are executed in a maximal way, and this decision allows us to use less global memory because it is not necessary to store the selected evolution rules for the execution stage. The rest of the rules to be applied are executed in four different kernels, one kernel per each kind of rule (dissolution, division, send-out, send-in).

Algorithm 4.4.1 shows the pseudo-code of the simulator. First of all, we move the data needed for the computation to the GPU. Then, the code calls the selection kernel which returns the selected rules for the current configuration of the P system. Among the possible selected rules there will be different kinds of rules to be executed. Therefore, the type of those rules is identified for launching only the required kernels to accomplish the execution stage. As we explained before, we iterate on this process until the maximum number of steps is reached or the system returns an answer. Finally, we copy back the result data to CPU.

Algorithm 4.4.1 Parallel simulator of P systems on the GPU

```
 1: configuration ← initialConfiguration
 2: selectedRules ← $\emptyset$
 3: step ← 0
 4: isFinalConfiguration ← false
 5: CopyDataFromCPUtoGPU(configuration)
 6: CopyDataFromCPUtoGPU(rules)
 7: while step $<$ maxStep $\wedge$ NOT isFinalConfiguration do
 8:     kernelSelection(rules,configuration,selectedRules)
 9:     if DISSOLUTION $\in$ selectedRules then
10:         kernelDissolution(rules,configuration,selectedRules)
11:     end if
12:     if DIVISION $\in$ selectedRules then
13:         kernelDivision(rules,configuration,selectedRules)
14:     end if
15:     if SEND-OUT $\in$ selectedRules then
16:         kernelSendOut(rules,configuration,selectedRules)
17:     end if
18:     if SEND-IN $\in$ selectedRules then
19:         kernelSendIn(rules,configuration,selectedRules)
20:     end if
21:     step ← step + 1
22:     isFinalConfiguration ← checkFinalConfiguration(configuration)
23: end while
24: CopyDataFromGPUtoCPU(configuration)
```

The parallel simulator implements the following data structures to store the P system information. Let us assume that the simulated P system with active membranes is of the form $\Pi = (\Gamma, H, \mu, \omega_1, \ldots, \omega_q, R)$, and creates at most M elementary membranes:

- *multisets*: an array storing the multisets of objects related with elementary membranes. The size is of order $O(|\Gamma| * M)$, and it is indexed by using the function $multisetsIndex(m, o) = |\Gamma| * m + o$, being m and o membrane and object identifiers.

- *skinMultiset*: an array storing the multiset of objects associated with the skin membrane. It is of size $O(|\Gamma|)$, and it is indexed using the object identifier.

- *environmentMultiset*: it stores the multiset in the environment.

- *rodRuleSet*: an array storing rules information. It is indexed by using an object o, a label l and a charge c: $rodRuleSet(o, l, c) = o{*}|H|{*}3{+}l{*}3{+}c$. Each position of the array is composed of four elements, so the size is of order $O(|\Gamma| * |H| * 3 * 4)$. These elements are:

  - Evolution rule identifier. It also stores the indexes for the *multisetRuleSet*, because evolution rules are the only one involving a multiset in their right-hand side.
  - Send-out, division or dissolution rule identifier. Only one of them is stored, and it is done by following the commented priorities.
  - Send-in rule identifier. We have to consider that, when focusing on an active membrane, the object for the send-in rule is the one in the parent. Therefore, we need to differentiate it with the rest, because, for example, a division and a send-in rule can be associated to the same label and charge, but the division rule cannot be applied because there is no objects available, and for the send-in rule yes.
  - Rule information. It stores some more information about the rules codified in the array position.

- *multisetRodSet*: an array storing all the objects appearing in the right-hand side multisets of evolution rules. It is indexed by a direct index (stored in the *rodRuleSet*) and an offset. The size is of the order $O(S)$, where S is total size of the right-hand side multisets of all evolution rules. It stores two elements, the object identifier and a multiplicity.

- *rsiodd*: an array which rule of send-in, send-out, division or dissolution has been selected for each membrane. It is of size $O(M)$. Note that only one element is enough for each membrane, since evolutions rules are executed directly in the selection kernel, and there is not need of storing them. These kinds of rules are selected which each thread blocks using a shared lock.

4.5 Performance comparative analysis

In this section we compare the performance of the developed simulators. We made two performance analysis based on two different case studies. The first one is a very simple example, with the aim of studying the behavior of the CUDA kernels. The second one is based on a real example from the literature. It shows a profiling of the simulators to better understand the structure and complexity of them.

4.5.1 Case study A: simple test example

In order to evaluate the performance of the simulator, we have designed a family of P systems, named test P system, where it is easy to vary the number of membranes as well as the number of objects. This test P system also fits the behavior of the GPU since only evolution and division rules are defined (without communication and dissolution rules), and every object in every membrane will evolve according to a given rule. The defined P system is of the following form $\Pi = (O, H, \mu, \omega_1, \omega_2, R)$, where:

- $O = \{d, o_i \ / \ 0 \leq i \leq n\}$,
- $H = 1, 2$,
- $\mu = [[]_2]_2$,
- $\omega_1 = \emptyset$, $\omega_2 = O$,
- $R =$

  (i) Evolution rules: $[o_i \rightarrow o_i]_2^0, 0 <= i < n$

  (ii) Division rule: $[d]_2^0 \rightarrow [d]_2^0[d]_2^0$

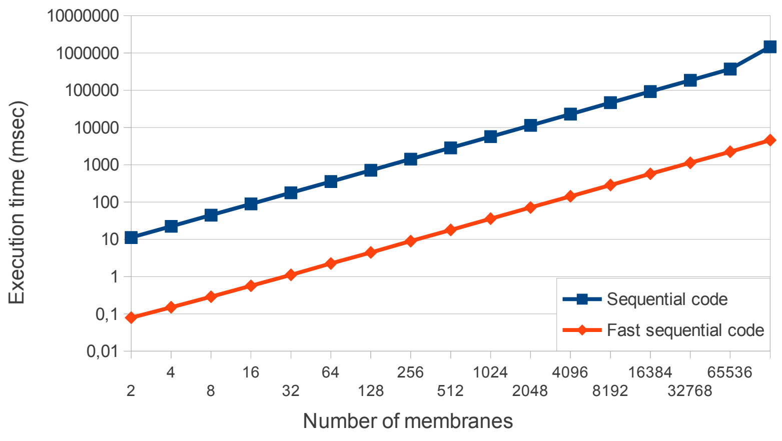
Thus, the test P system allows us to take control of the number of objects in the system by modifying the n parameter. Furthermore, the number of rules changes along with the number of objects, and the number of membranes in every step of the computation is equal to 2^s, where s is the step number. Lastly, the number of evolution rules selected and executed per membrane in every step is invariable, since they are defined one per object and all the objects of the alphabet are presented in every membrane labeled with 2.

First of all, we compare the performance of both sequential simulators. Note that they run on the CPU, but they mainly differentiate on the management of memory. The sequential simulator (let designate it *amp-seq*) is the most flexible of them all, but the slowest one. The fast sequential simulator (let call it *amp-fast-seq*) supports only two levels of membrane hierarchy, and the maximum amount of membranes has to be previously declared. It has been designed to be the CPU counterpart of the parallel (GPU) simulator (let call it *amp-gpu*), since for performance comparison, we should consider the run-time of the fastest serial program (see Chapter 3).

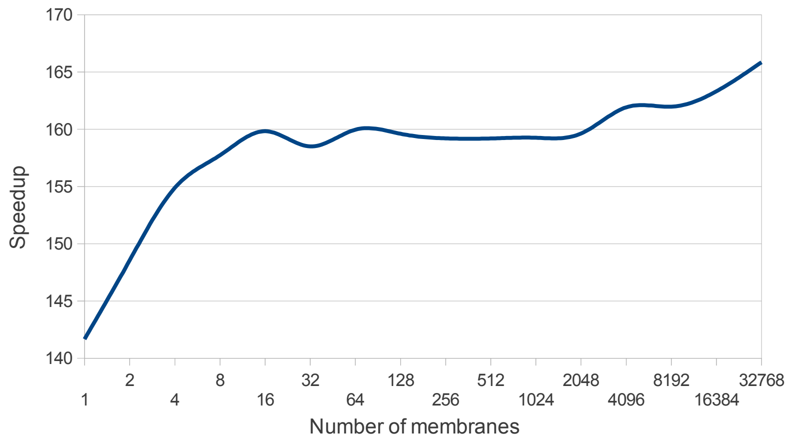
Figure 4.4(a) shows the run-time for *amp-seq* and *amp-fast-seq*, for only one step and for different amount of membranes, having 2560 objects in the alphabet. The corresponding speedup is also attached (Figure 4.4(b)). The amount of membranes is exponentially increased by applying division rules, and the Y-axis is showed in logarithmic way. We can observe that *amp-fast-seq* outperforms *amp-seq* for any instance. It saves both time and memory, because it avoids to allocate and deallocate memory on demand by using a previously allocated array for objects. The obtained speedup is 160x, what is a good number. Moreover, we can see that the speedup tends to increase for bigger instances. The main reason for this is the pagination process, because *amp-seq* requires more memory. We therefore take the *amp-fast-seq* for performance comparison from now on.

Figures 4.5 and 4.6 present the results we have obtained for the simulator between the sequential version developed in the C++ language and our simulator developed in CUDA. Notice that in both graphs the Y-axis is also represented in an logarithmic form. These benchmarks cover both ways of parallelism that P systems naturally have by its definition. The first one tests the parallelism between membranes, exponentially increasing the number of membranes, and the second one tests the parallelism between objects exponentially increasing the number of objects within each membrane.

Figure 4.5 shows the results for the benchmark which increases the number of membranes exponentially, having a fixed number of objects per membrane (2560 objects). The CPU simulator also increases its time exponentially from

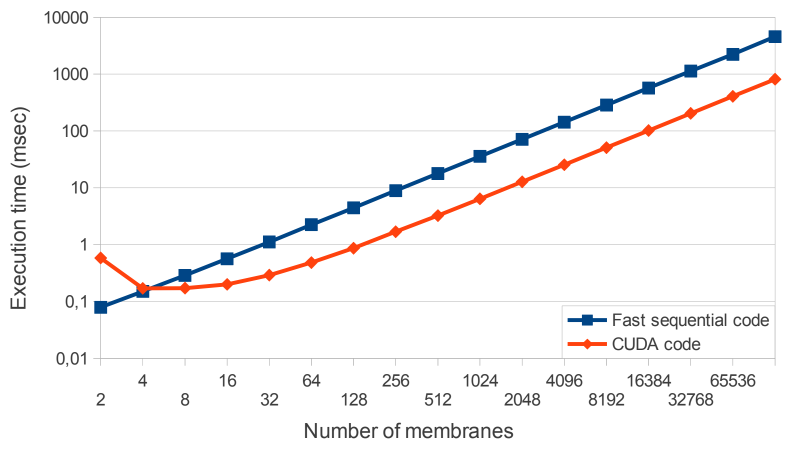


(a)

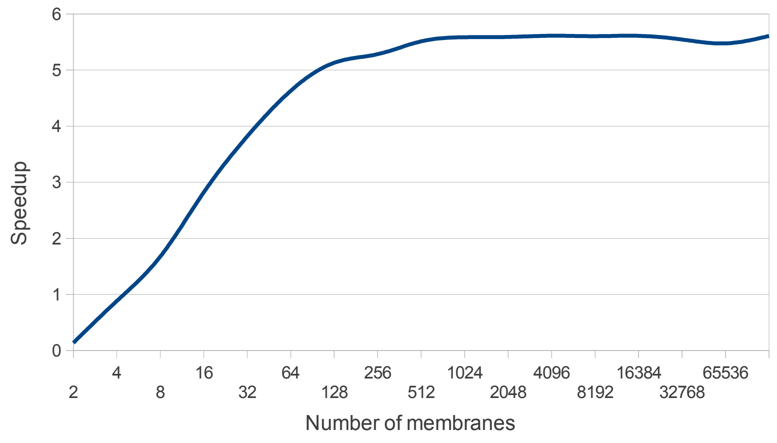


(b)

Figure 4.4: Comparing the execution time (a) and speedup (b) for one step of *amp-seq* and *amp-fast-seq*, by increasing the number of membranes in the system and using a total of 2560 objects in the alphabet.

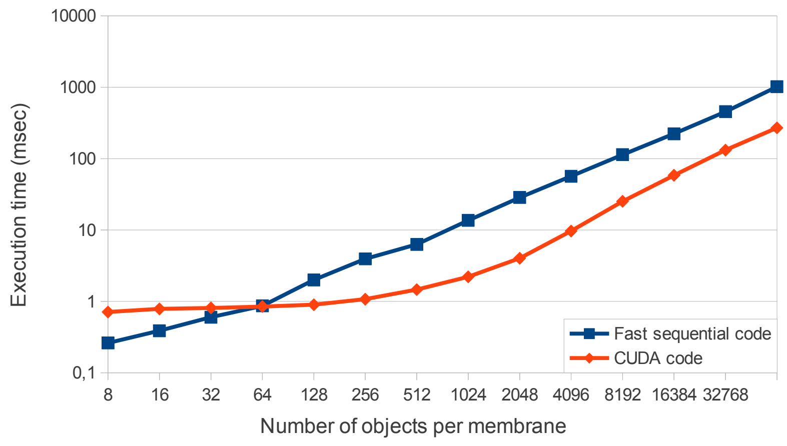


(a)

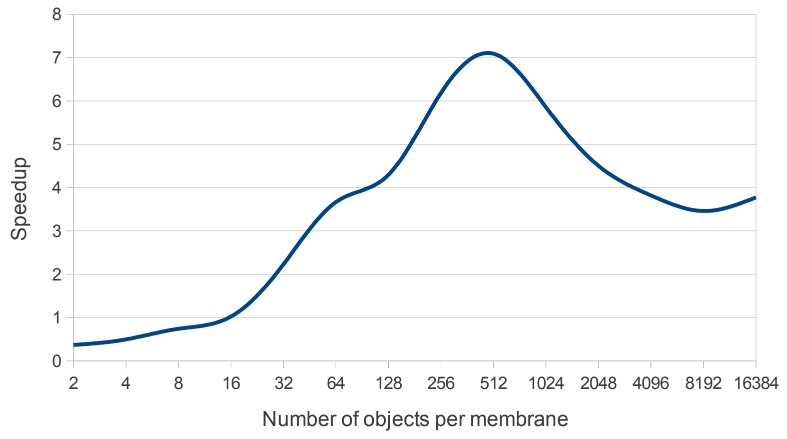


(b)

Figure 4.5: Comparing the execution time (a) and speedup (b) for one step of the fast sequential and parallel simulators, by increasing the number of membranes in the system and using a total of 2560 objects in the alphabet.



(a)



(b)

Figure 4.6: Comparing the execution time (a) and speedup (b) for one step of the fast sequential and parallel simulators, by increasing the number of objects in the system and using a total of 1024 membranes.

the beginning (with four membranes) until reaching the final configuration (with 32768 membranes). Our CUDA simulator, which assigns 256 threads per block (each thread handles 10 elements per membrane), also increases its execution time in a near exponential way, but the performance difference is about 5.7x, and this difference enlarges with the number of membranes (from 1024), because the resources of the GPU are fully utilized.

Figure 4.6 shows the behavior of both simulators executing the benchmark which increases the number of objects per membrane. In this case, the number of membranes is fixed to 1024, which implies to have enough blocks to distribute the work among multiprocessors (as seen in Figure 4.5). Our simulation starts with only few objects per membrane, which implies just few threads per block in the CUDA code. Figure 4.6 shows that *amp-fast-seq* initially obtains better performance than *amp-gpu* until the simulations reach 32 elements per membrane. Less than 32 elements per membrane implies less than 32 threads per block in *amp-gpu* which does not fill a warp; hence GPU resources are badly used. The sequential version increases its simulation time along with the number of objects since just one thread has to deal with all the objects in each membrane.

The simulation time remains flat using the CUDA version until reaching a 256-object configuration. The simulation time increases a little bit faster from this configuration onwards because the following configurations have more objects per membranes than threads per block (it uses 256-thread blocks). Therefore, objects in a membrane are equally distributed across all the threads in a block: 512-object per membrane implies two objects per thread; 1024-object per membrane implies 4 objects per thread, and so on. Otherwise, it implies to have an overloaded thread which reduces the performance of our simulator, and leads us to conclude that it is better to have lightweight threads.

Overall, we have obtained a reduction in the simulation time, reaching for 512 objects and 1024 membranes an improvement of 7x in the execution time between *amp-fast-seq* and *amp-gpu*.

4.5.2 Case study B: an efficient solution to SAT by a family of P systems

In this second case study, we analyze the performance of the simulators by running a real example from the literature: a well-known family of P systems with active membranes solving `SAT` in linear time (see Sections 1.6 and 1.6.1).

Table 4.1 shows the results of both fast sequential and parallel GPU simulators. It provides the run-times of each simulator, for the whole computation.

Note that in the previous case study, we only considered the run-time for one computational step to show if the parallel execution outperforms the sequential version. This time, we consider the full run-times of the simulators.

We consider only the full computation time (given in milliseconds) for *amp-fast-seq*. For *amp-gpu*, we consider the full computation time (also given in milliseconds) taken by each kernel, plus the overhead caused by the CUDA memory management, which is an extra to consider when using the GPU. Since all the data structures of the sequential simulator are also used by our CUDA simulator, we do not take into account their management time in both simulators.

Moreover, we also provide a profiling of the computation for both simulators. We use for this the percentage of time consumed by each part of the simulators:

- For the sequential simulator (*amp-fast-seq*): the time percentages for selection and execution phases of the simulator.
- For the parallel simulator (*amp-gpu*): the time percentages for kernels computation (selection phase plus execution phase), memory allocation (malloc) and memory transferences.

For each simulated instance, we show the `SAT` instance size (number of variables n and number of clauses m of the encoded CNF formula), and the size of the simulated P system, in terms of maximum number of created membranes and total amount of objects of the alphabet.

At first glance, we can see that selection phase is the most time consuming, for both simulators. Indeed, selection phase takes more than the 90% of the computation time for both simulators. Execution phase is actually the less complex one, since it simply adds objects to the multisets according to the decisions taken in selection phase.

We can also see a huge overhead caused by memory allocation in the parallel simulator. For small instances the overhead takes around the 90% of the time. However, it is static (always the same value for any instance size), so for larger instances the overhead is hidden by the time consumed by kernels. For the largest instance we could run, this overhead is completely hidden. Furthermore, the time for memory transferences is always the same but small, so it is negligible.

Instance size			**Fast sequential**			**Parallel (GPU)**						**Speedup**	
SAT (n,m)	Membranes	Objects	Time(ms)	% Sel.	% Exec.	Time(ms)	% Kernel	% Sel.	% Exec.	% Malloc	% Transf.	Kernel	Total
(6,33)	64	920	18.34	90.3%	9.7%	1215.1	1.37%	1.13%	0.24%	98.14%	0.49%	1.1x	0.01x
(8,32)	256	1155	83.44	93.13%	6.87%	1286.99	4.89%	4.56%	0.33%	94.61%	0.5%	1.32x	0.06x
(10,16)	1024	729	161.35	88.85%	11.15%	1341.5	8.95%	8.28%	0.67%	90.63%	0.42%	1.34x	0.12x
(12,17)	4096	914	867.88	90.45%	9.55%	1725.39	30%	28.24%	1.76%	69.68%	0.32%	1.67x	0.5x
(14,17)	16384	1056	4082.2	91.14%	8.86%	3762.81	68.09%	65.07%	30.02%	31.75%	0.16%	1.59x	1.08x
(16,16)	65536	1131	17757.4	91.56%	8.44%	12840.6	90.4%	87%	3.4%	9.54%	0.06%	1.53x	1.38x
(18,32)	262144	2465	215177	93.47%	6.53%	145497	99.22%	97.05%	2.17%	0.77%	0.01%	1.49x	1.48x

Table 4.1: Profiling the simulators *am-fast-seq* and *am-gpu*, running several instances of the solution to `SAT`.

The execution of kernels always outperforms the sequential code. The speedup is increased with the instance size, achieving the maximum of 1.67x for 4096 membranes and 914 objects. However, the full parallel simulation time only outperforms the sequential counterpart when the overhead is hidden by the computation time of kernels. We report the maximum speedup of 1.48x for the largest instance.

Nevertheless, note that the speedups for this example are very small compared with the simple test example of the previous case study. The performance is downscaled around 4 times for this example (from 5.7x to 1.38x for 65536-membrane P systems). The speedups does not go beyond the 2x, what is a bad result. The main reason is the small rule intensity of this P system. We will talk more about this in the characterization (next) section, so that we can better understand which are the P systems characteristics that are well suited to be accelerated on the GPU.

4.6 Characterizing the simulation on the GPU

In this section we characterize the simulation of P systems with active membranes on the GPU, considering the results obtained during the performance analysis.

First of all, as mentioned before, our simulator presents some limitations, constrained by some peculiarities in the CUDA programming model and design decisions. The main limitations are shown in table 4.2, and the following stand out among them:

- The simulator can handle only two levels of membrane hierarchy for simplicity in synchronization (the skin and the rest of elementary membranes), which is enough for solving a wide range of **NP**-complete problems. This is a design decision which aims to simplify the problem of object reservation by send-in rules in the parent membrane, which means concurrent access from different thread blocks to the same memory space, what has to be avoided in CUDA.

- The number of objects in the alphabet must be divisible by a number smaller than 512 (the maximum thread block size), in order to equally distribute the objects among the threads.

The CUDA memory model is statically managed, so the programmer should make an estimation of the memory usage by the kernel in order to statically

Table 4.2: Main limitations in the parallel simulator

Parameter	**Limitation**
Levels of membrane hierarchy	2
Maximum alphabet size	65535
Maximum label set size	65535
Maximum multiplicity of an object in an elementary membrane	65535
Alphabet size	Divisible by a number smaller than 512

allocate in advance the memory required by the kernel execution. However if this estimation is not enough to completely execute the kernel, the GPU execution should finish, back the control to the CPU, reallocate resources on it, and then start again the kernel execution. Definitely, this is expensive in time. This is one of the key aspect to take into account when designing solutions on CUDA. A naive solution is to allocate enough memory in order to store the worst case of a problem.

In the case of P systems with active membranes, the worst case is presented when all different objects of the alphabet are presented in the multiset of a membrane in a given configuration. Although this is the worst case that the simulator has to deal with, it does not take place in the majority of P systems to be simulated. Thus, the performance of the simulator totally depends on the simulated P system.

Indeed, in the previous section we have shown that the performance can vary depending on the case study. We see that the test P system (case study A), where all the rules are applied to each membrane in all the steps, is better accelerated (up to 5.7x for 65536 membranes) than a real example from the literature, an efficient solution to `SAT` (case study B, up to 1.53x for kernels).

Looking at the results, and the general CUDA design of the kernels, there are some properties that makes the simulation on the GPU faster than on the CPU:

(a) Density of objects per membrane: more different objects should appear in the multiset (having multiplicity greater than 0) of every membrane to increase the CUDA threads usage.

(b) Rule Intensity: more evolution rules are defined to evolve the objects within membranes in parallel. Since the objects evolve according to the evolution rules, the density of objects can affect the rule intensity, but it definitely depends on the amount of defined rules. For example, an strategy can be to reuse objects defined in a P system.

(c) Communication among membranes: if less objects are sent to or retrieved from the skin, the communication through PCI express bus is reduced (the skin is executed on the CPU).

For example, the density of objects in the test P systems is of 100% (every object in the alphabet is present in all membranes, and they evolve by the high rule intensity). The obtained performance by the CUDA simulator is therefore much higher. However, for the family of P systems solving `SAT`, the density of objects is an average of 15%, so that the 85% of threads are wasted in the simulation process. This leads to a lower performance in general. Moreover, the sequential simulator has several optimizations that avoids to process the 85% of inexistent objects. This improvements are expensive to implement in CUDA, since it is based in the ordered insertion of elements in the array.

Thus, the above mentioned properties can be considered for defining several strategies for P system definitions, so that the simulation of the defined P systems can be better accelerated on the GPU. Finally, recall that the strategies from the properties (a) and (b) are directly related with the key point of designing solutions in Membrane Computing by using massive parallelism [81]:

Intuitively, a bad design of a P system consists of a P system which does not exploit its parallelism, that is, working as a sequential machine: in each step only one object evolve in one membrane whereas the remaining objects do not evolve. On the other hand, a good design consists of a P system in which a huge amount of objects are evolving simultaneously in all membranes. If both P systems perform the same task, it is obvious that the second one is a better design than the first one.

The fact of this relationship comes out from the idiosyncrasy of the GPU and our design: the double massively parallelism of P systems has been directly mapped with the double highly parallelism of GPUs. Therefore, the performance of the GPU simulator depends directly on the design of P system (if it is bad or not), as pointed in [81].

In this respect, a good way to previously study if the simulation of a P system can be well accelerated on the GPU is by using *Sevilla Carpets* [79]. Sevilla Carpets provide on one hand quantitative information, and on the other hand a fast glimpse on the complexity of the computation thanks to their graphical representation. We can assume that Sevilla Carpets can be used to analyze a priori the achieved simulation acceleration of a given P system. They can help to identify the properties listed above, and to search for many others. Some future work can be considered in this respect (as described in Chapter 8).

4.7 Conclusions

In this chapter we have introduced the PCUDA framework (a subproject of PMCGPU [17]). It includes simulators for confluent recognizer P systems with active membranes and elementary division:

- *A sequential simulator*: it is written in C++, and supports any kind of P systems of this class. C++ pointers are used to represent the membrane hierarchy and the multisets. The simulation algorithm is based on the one implemented in the simulators of the pLinguaCore simulation framework. The algorithm is divided into two stages: selection (how many times each rule is going to be executed) and execution (update the state of the P system according to the output of selection stage). This permit to synchronize the way to apply rules in every membrane.

- *A parallel simulator*: it is based on CUDA. The double parallelism of these P systems is represented in the double parallelism of GPUs. That is, each CUDA thread block works with an elementary membrane, and each CUDA thread is assigned to a set of objects within the corresponding membrane. Some constrains are also required to the type of P systems to simulate, that mainly are: only two levels of membrane hierarchy, and an alphabet size divisible by a number smaller than 512.

- *A fast sequential simulator*: it is based on the parallel simulator design, but run on the CPU. The same constrains than the CUDA simulator are presented here. Moreover, it uses static memory management (as the parallel simulator), so the performance is much better than the sequential simulator (that uses dynamic memory allocation, loosing time on the simulation process).

The provided simulation framework is therefore flexible enough to run many P systems from the literature, but performance and scalability are compromised. Experiments showed that the best acceleration is achieved when the P system to simulate intrinsically presents a high degree of parallelism. That is, a high degree of rule intensity (number of rules executed per each step) and density of objects (number of objects appearing in the multisets).

"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps", no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss..."

Scott Aaronson, MIT

5 Parallel simulation of P systems solving SAT

The use of powerful supercomputers has been proposed over the past years to tackle certain instances of Natural Computing methods, among which we may cite ant colony [149], particle swarm [113] and even genetic algorithms [137]. Following this trend of good alliance between applications and hardware, we contribute with novelties on both sides:

- At hardware level, we propose the use of commodity graphics hardware (GPUs) as a low-cost and emerging parallel architecture to accelerate the simulations. The newest version of programmable GPUs provide a compelling alternative to the traditional parallel environments such as cluster of computers, delivering extremely high floating point performance and also a massively parallel framework for scientific applications which fit their architectural idiosyncrasies.
- At application level, we focus on Membrane Computing, an emergent research area which abstracts computing ideas (like data structures, operations or computing models, among others) from the structure and behavior of single cells, ultimately grouped into complexes of cells. P systems are distributed parallel and non deterministic computing devices, and some models have been successfully used for designing polynomial time

solutions to NP-complete problems by trading space for time. Specifically, these models were inspired by the capability of cells to produce an exponential number of new membranes in linear time, through *mitosis* (membrane division) and/or *autopeosis* (membrane creation) processes. Major challenges on P systems simulations are (1) a dynamic handling of memory space and (2) an exponential workspace growing as our code increases the number of variables involved to run the simulation.

Our previous attempt to design simulators on GPUs for P systems has demonstrated that a parallel architecture is better positioned in performance than traditional CPUs to simulate P systems, due to the inherently parallel nature of them, and specifically GPUs obtain very good preliminary results simulating P systems.

In this chapter, we analyze the behavior of GPUs simulating a family of P systems with active membranes solving the `SAT` problem in linear time [133] (Section 1.6.1). Unlike the simulator described in Chapter 4 (which is a flexible simulator for the P systems with active membranes), this one is optimized for this particular family of P systems, achieving a more efficient simulation in spite of decreasing the flexibility. This work was first published in [40], and will be detailed in Section 5.1. It has been improved by better adapting the simulator to new GPU architectures and multi-GPU systems [43, 46] and to supercomputers [42]. These two extensions are summarized in Section 5.5 and 5.6, respectively.

Moreover, we also analyze the parallel simulation of another efficient solution to `SAT` based on a family of tissue P systems with cell division (Section 1.6.2). This simulator was presented in [123], and it is detailed in Section 5.2. The aim of this simulator is to further study which ingredients of different P systems models are well suited to be managed by the GPU. It is carried out by comparing the performance with the GPU simulator for cell-like P systems. The performance analysis and simulation characterization are explained in Sections 5.3 and 5.4, respectively.

5.1 Parallel simulation of the solution with cell-like P systems on the GPU

In this section we describe the simulator for the family $\mathbf{\Pi_{am-SAT}}$, described in Section 1.6.1 (Chapter 1), of recognizer P systems with active membranes solving `SAT` in linear time. First, we explain the previous work to prepare the development of the parallel simulator on the GPU. Then, we introduce the simulator design that fully simulates a P system computation to solve instances of the SAT problem. Finally, we describe an adapted simulator to the GPU to accelerate the simulation.

The framework of all these simulators is named *PCUDASAT*, and it can be downloaded from the website: `http://sourceforge.net/p/pmcgpu` [17]. More information about the simulator, how to install and use it, refer to Appendix A.

5.1.1 Design of the baseline simulator: Sequential Simulator

As mentioned before, the first recommended step when developing applications in CUDA is to start from a baseline algorithm written in C++, where some parts can be susceptible to be parallelized on the GPU. The sequential simulator design is based on the four main phases of a P system computation from $\mathbf{\Pi_{am-SAT}}$, as it is depicted in Section 1.6.1: Generation, Synchronization, Check-out, and Output. All these phases are sequentially executed in this simulator, reproducing the computation of the P system.

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

It is important to remark that the semantics of the P system is reproduced by the simulation algorithm, so the simulator is specific for this solution. Thus, we don't need to receive a binary file describing the P system. It is enough to receive an instance of the SAT problem through a CNF formula. We can assume therefore that the simulator behaves as a SAT solver, receiving a propositional formula and giving the corresponding *yes* or *no* answer. However, this solver is implemented following a solution by means of P systems. For this purpose, the input of this simulator is a standard DIMACS CNF file, where the CNF propositional formula is encoded. The output is read from the environment of the P system, where the result is stored.

5.1.2 Reading the input file

The input file, as mentioned above, is a text file matching the DIMACS CNF format. It has become a standard for SAT solvers. This file is read character by character.

Normally, this format file has comments at the beginning. These comments start by the character 'c'. The comments finish when a line starting by 'p' is given. After this character, a blank space plus the word 'cnf' has to be placed. If all of these is present, then we can assume that the file is in DIMACS CNF format. After 'p cnf', the number of variables 'n' and the number of clauses 'm' are defined. These values must be greater than 0, otherwise it is not encoding a correct formula.

Next, we read the set of literals and clauses. We have one line per clause, ending by character '0'. Each line is composed by a list of numbers separated by spaces, representing the literals (from 1 to n). Such numbers can be negative or positive: negative numbers mean negating the corresponding variable, and positive the variable itself. In our implementation we use a matrix of $n \times m$ dimension, placing 0 if a variable (row) is not present in a clause (column), 1 if it appears as the variable, or -1, if it appears negated. Moreover, we also count the total number of literals ($|cod(\varphi)|$).

Finally, we provide an example for the better understanding of the file format we use for our simulator. Let us assume the following propositional formula in CNF: $(x_1 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_1})$. A file representing this formula, using the DIMACS CNF format, will be as follows:

```
c
c  This is the DIMACS CNF file representing the example formula
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

5.1.3 Data structures

The main goal of this specific simulator is to optimize the simulation of the family $\mathbf{\Pi_{am-SAT}}$. The first challenge was to increase the object density in the implementation, since this parameter is around 15% in the theoretical model, as seen in Section 4.6. Increasing the object density means to avoid extra work over inexistent objects. The solution was to find an upper bound for the number of different objects that can simultaneously exist within any membrane.

After an exhaustive analysis of the computation, the upper bound was fixed to the size of the input multiset (the number of literals in the input propositional formula). Indeed, one can observe that the size of the right-hand side of evolution rules is always 1. Thus, every object in the input multiset always evolves to another, but it never increases the number of objects (it can be decreased). The same also for the division rule. For the send-in rules, it can be also observed that, before them, send-out rules are applied, so they can be managed as an exchange of objects through transitions.

The representation of the P system is made by an array storing the multisets of objects for every membrane labeled by 2. The amount of elements per membrane equals to, as mentioned above, the size of the input multiset (total number of literals in the formula, $|cod(\varphi)|$). This array is initially allocated for the maximum amount of membranes 2 that the P system will create, which is 2^n (note that n is defined in the input file). Only the first one is initialized by storing the full input multiset. Division rules will initialize each membrane later on.

We encode the objects for the input multiset in the mentioned array at bit-level within integers of 32 bits. Each integer stores the following (8 bits for each field):

1. The name of the object (x or $\overline{x}$)
2. Reserved space.
3. Variable (index i).
4. Clause (index j).

It is noteworthy that the membrane charges are not stored anymore. From the computation of $\mathbf{\Pi_{am-SAT}}$, we can observe that a partition of membranes having positive and negative charges can be done over the array, that is, the first half of membranes are positive, and the other half (new ones) negative. The skin membrane is not represented, since its purpose is to store objects sent out from membranes, those which are sent in to the same membranes in the next step. This process is therefore simulated within each membrane, avoiding to store the information for the skin membrane. Other objects, such as *yes*, *no* and c counter, are also placed as variables in source code.

5.1.4 Design of the GPU Simulator: Parallel Simulator

The objective of this parallel simulator is to fully simulate the behavior of a P system computation, doing this in a parallel way whenever is possible. To

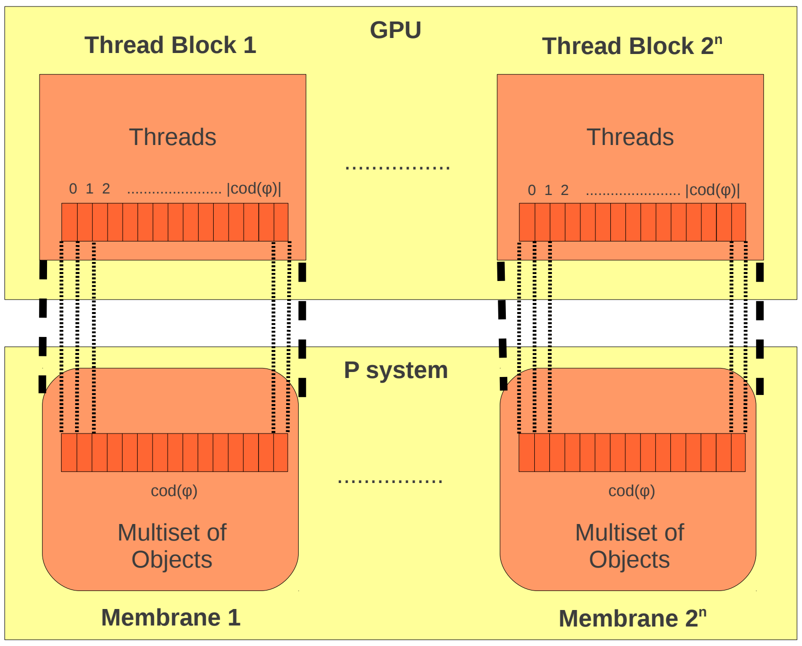


Figure 5.1: General design of the parallel simulator for $\mathbf{\Pi_{am-SAT}}$.

do that, we use the baseline design based on the four main phases of the P system computation. The first three phases are developed as CUDA kernels in this simulator, and the last one (Output phase) is developed on the CPU.

Similarly to the design of the simulator presented in Chapter 4, this simulator assigns a thread block to each membrane, as shown in Figure 5.1. In this way, the parallelism among membranes is represented. Moreover, each thread is assigned to each object of the input multiset, which is a literal of the input formula (with the exception of object d_1). This mapping is common to all the defined kernels.

Algorithm 5.1.1 shows the pseudocode for this version of the simulator. The Generation phase is simulated by using three kernels which computes the rules previously explained in Section 1.6.1. This is an iterative process of n steps where the kernels are called n times. In each iteration, the simulator adjusts the number of thread blocks before calling the kernel, since new membranes are created. That is, the membranes are distributed along the 2-dimensional grid of thread blocks.

When the exponential workspace is created, the Synchronization and Check-out phases are executed (following the rules showed in Section 1.6.1). Both phases are performed in the same kernel, and so, in parallel to each membrane.

Algorithm 5.1.1 Parallel Simulator, reproducing the P system computation

```
1: $Threads \leftarrow |cod(\varphi)|$        ▷ The number of literals in the CNF formula
2: $Blocks \leftarrow 1$
3: repeat                                     ▷ Generation phase
4:     $Division\_kernel <<< Blocks, Threads >>> (numMembranes)$
5:     $numMembranes \leftarrow numMembranes \times 2$
6:     $Blocks \leftarrow AdjustBlocks(numMembranes)$        ▷ Distribute membranes
                                               ▷ into the 2-dim grid
7:     $Send\_out\_kernel <<< Blocks, Threads >>> (numMembranes)$
8:     $Send\_in\_kernel <<< Blocks, Threads >>> (numMembranes)$
9:     $d \leftarrow d + 1$
10: until $d < n$                 ▷ Repeat n times (number of variables)
                                  ▷ Synch. and Check-out phases
11: $Syn\_Check\_kernel <<< Blocks, Threads >>> (numMembranes)$
12: $Output(numMembranes)$        ▷ Output phase (executed on the CPU)
```

Global synchronization is not necessary because there is no communication among the internal membranes at this phase. Finally, the Output phase is developed on the CPU, checking the conditions and launching the result of the computation.

5.1.5 Adapting the simulator to the GPU architecture: Hybrid Simulator

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

GPUs are basically a graphics accelerator which are designed to mainly accelerate graphics applications. Graphics applications presents huge data parallelism, that is, developing the same computation over different set of data. Then, the communication and synchronization requirements among processing elements should be drastically limited to enhance performance [94].

The objective of this simulator is to reduce the communication and synchronization overheads presented in the previous simulator by using several heuristics as shown in Algorithm 5.1.2. For instance, some objects that con-

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

Algorithm 5.1.2 Hybrid Simulator. Adapting the P system computation

```
1: $Threads \leftarrow |cod(\phi)|$                    ▷ The total number of literals in the CNF formula
2: $Blocks \leftarrow 1$
3: repeat                                                  ▷ Generation phase
4:     $Generation\_kernel <<< Blocks, Threads >>> (numMembranes)$
5:     $numMembranes \leftarrow numMembranes \times 2$
6:     $Blocks \leftarrow AdjustBlocks(numMembranes)$          ▷ Distribute membranes
                                                           ▷ along the 2-dim grid
7:     $d \leftarrow d + 1$
8: until $d < n$                             ▷ Repeat n times (number of variables)
                                 ▷ Synchronization, Check-out and Output phases
9: $Syn\_Check\_kernel <<< Blocks, Threads >>> (numMembranes)$
```

trol the timing of the theoretical computation, depicted in Section 1.6.1, are replaced by statical variables instead of dynamic variables. Doing this, this simulator can join CUDA kernels, and therefore, reduce the synchronization overhead produced by launching kernels onto the GPU, since only one kernel can be launched at the same time on the GPU.

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

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

5.2 Parallel simulation of the solution with tissue-like P systems on the GPU

In this section we depict the simulator for the family $\mathbf{\Pi_{tsp-SAT}}$ of recognizer tissue P systems with cell division, described in Section 1.6.2. For this simulator, we have only constructed a full simulator (there is no a hybrid version), so that we can perform comparison with the cell-like version. We first explain the data structures and the phases that compounds the simulation algorithm, then the sequential version, and after that, the parallel one based on CUDA, describing the different optimizations taken for each phase of the simulator.

This simulation framework is named *TSPCUDASAT*, and it can be downloaded from the website: `http://sourceforge.net/p/pmcgpu` [17]. More information about the simulator, how to install and use it, refer to Appendix A.

5.2.1 Sequential simulation

For an easier implementation, the simulation algorithm has been divided into five (simulation) phases. Note that they are different in number than the denoted phases of the theoretical model. Each of these simulation phases are implemented in code as separated functions whenever is possible. They corresponds to the application of certain rules, as explained below:

- *Generation phase*: it performs the application of rules from (a) to (e) of systems from $\mathbf{\Pi_{tsp-SAT}}$ (Section 1.6.2). Therefore, it comprises the two first phases of the theoretical model: valuations generation phase and counters generation phase.
- *Exchange phase*: it simulates the application of rules (f) and (g). It comprises the first part of the checking preparation phase.
- *Synchronization phase*: it applies the rules from (h) to (m), so comprising the second part of the checking preparation phase.
- *Checking phase*: it performs the application of rules from (n) to (p). Thus, it is the checking clauses phase we identified in the theoretical model.
- *Output phase*: it applies rules from (q) to (t). It then performs both the formula checking phase and the output phase identified in the theoretical model.

The sequential simulator implements these five simulation phases directly in code, which is in C++. Each one works directly with the data structures depicted in the next Subsection 5.2.2. The input of the simulator is the same than the one used in the simulator for the cell-like solution. A DIMACS CNF file is provided, and the simulator outputs the response of the computation. Therefore, it acts merely as a `SAT` solver, but the implementation follows the computation of the systems from the family $\mathbf{\Pi_{tsp-SAT}}$.

Furthermore, we have adopted a set of optimizations to improve the performance of the sequential simulator. After several tests, we show that the best optimizations are [123]:

- As the Exchange phase is very simple, it is then implemented after the Generation phase loop, within the same function.
- We apply the full Synchronization phase to one cell before going to the next one. This allows us to exploit data locality in cache memories.
- In the Checking phase, we orderly insert the objects r_j, for $1 \leq j \leq m$, in the corresponding array whenever they are created. Thus, the Output phase can be easily performed, in such a way that it is not necessary to loop all the objects coming from the input multiset (literals). Now it is enough to check if there exists the m objects r_j.

5.2.2 Data structures

For this solution, the representation of the tissue P system is twofold. As the model differentiates between cells labeled 1 and 2, the design decision was to also have a different data structure representing each type of cell in the system.

First, the cell 1 is represented as an array having a maximum dimension of 5 elements. That is, the multiset for cell 1 has the maximum amount of 5 objects. These 5 objects are the three counters, b, c and d (which are initially in this cell), and the two objects yes and no (that will final answer to the problem). Note that the size of the array for cell 1 is always constant, as it is independent of the input parameters of the simulator.

Second, the cells labeled by 2 are also represented by a one-dimensional array. All of them are stored inside this large array, since it is initially allocated to store the maximum amount of cells (2^n). By studying a computation of the systems $\mathbf{\Pi_{tsp-SAT}}$, we conclude that the maximum number of objects appearing in a cell 2 is $(2n)+4+|cod(\varphi)|$, where (see Section 1.6.2 for further information about the alphabet):

- $|cod(\varphi)|$ elements for the initial multiset,
- n elements for objects $T_{i,j}$ and $F_{i,j}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Note that an object $T_{i,j}$ and an object $F_{i,j}$, for any i, cannot be simultaneously placed within a cell 2. Moreover, the index j is used sequentially in the computation steps of the system, i.e. replacing objects in the evolution process of incrementing the second index. For all of this, n elements are enough to store those objects.
- n elements for objects t_i and f_i, for $1 \leq i \leq n$. Note that objects f_i and t_j, for $i = j$, cannot be simultaneously placed within a cell 2, so n elements are enough to store those objects.
- 4 elements for counter objects a, b, c and d. They will be replaced for counter objects f and g.

The objects are represented similarly to the simulator for $\mathbf{\Pi_{am-SAT}}$. In this case, we recover the reserved space utilized to store the multiplicity of the object, inasmuch as it can exceeds 1. In summary, they are encoded at bit-level within integers of 32 bits, that store the following (8 bits for each field):

1. The name of the object (x or $\overline{x}$)
2. Multiplicity of the object. As there are objects whose multiplicity can exceed 2^8, this field can eventually be joined to the next one (variable).
3. Variable (index i).
4. Clause (index j).

5.2.3 Design of the parallel simulator

The parallel simulator is designed to also fully reproduce a computation of the systems from the family of tissue P systems $\mathbf{\Pi_{tsp-SAT}}$. That is, there is no a hybrid solution providing simulation shortcuts to the computation. The design of this parallel simulator is driven by the same structure of phases we have used for the sequential one. Separate CUDA kernels are utilized to speedup the execution of each phase.

Similar work distribution in CUDA than in the other simulators (PCUDA, presented in Chapter 4, and PCUDASAT, presented in Section 5.1) is again

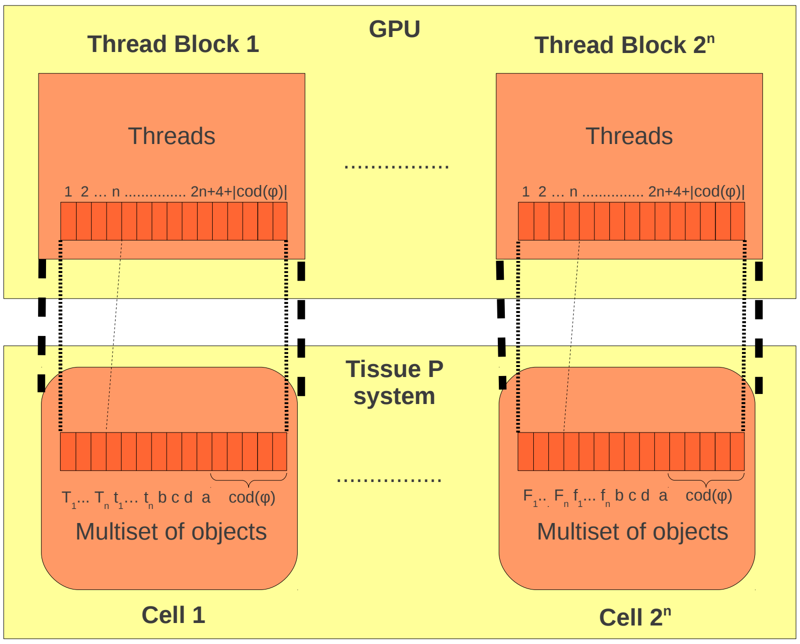

GPU
Thread Block 1
Thread Block 2^n
Threads
Threads
1 2 ... n .... 2n+4+|cod(φ)|
1 2 ... n .... 2n+4+|cod(φ)|
Tissue P system
$T_1 \ldots T_n\ t_1 \ldots t_n$ b c d a cod(φ)
$F_1 \ldots F_n\ f_1 \ldots f_n$ b c d a cod(φ)
Multiset of objects
Multiset of objects
Cell 1
Cell 2^n

Figure 5.2: General design of the parallel simulator for $\mathbf{\Pi_{tsp-SAT}}$.

applied. The general assignment of work for threads and thread blocks is summarized in Figure 5.2. Each thread block corresponds to each cell labeled by 2 created in the system. However, unlike the previous simulator for the cell-like solution, we do not assign a thread per literal. The assignment of each thread, this time, is different for each simulation phase. The work mapping per phase is therefore as follows:

- *Generation phase*: the number of thread blocks is iteratively increased together with the amount of cells created in each computation step. We distribute cells along the two-dimensional grid through successive kernel calls. Each thread block contains $(2n) + 4 + |cod(\varphi)|$ threads. That is, the amount of elements assigned to each cell in the global array storing multisets. Threads are then used to copy each individual elements of the corresponding cell when it is divided.

- *Exchange phase*: it is executed at the kernel for Generation phase, using the same amount of thread blocks, but only the corresponding threads perform the exchange.

- *Synchronization phase*: the thread blocks are assigned to the cells labeled by 2, like in the last step of the Generation phase. For this phase, the number of threads is n (number of variables). If we use the same amount of threads than in Generation phase, most of them will be idle. So it is preferred to launch less threads, but performing effective work. We have experimentally corroborated this fact.

- *Checking phase*: the number of thread blocks is again assigned to be the number of cells labeled by 2. However, for this phase we use a block size of $|cod(\varphi)|$. That is, each thread is used to execute, in parallel, rules of type (n) and (o). The result at the SAT problem resolution level, each thread checks if the corresponding literal makes true its clause, depending on the truth assignment encoded by the cell assigned to the thread block.

- *Output phase*: rules of type (q) are sequentially executed in a separate kernel, again using $|cod(\varphi)|$ threads per block, and 2^n thread blocks (2^n is the number of cells labeled by 2).

For this solution, we have applied a small set of optimizations, focused on the GPU implementation, to improve the performance of the parallel simulator. As analyzed in [123], we identify that the simulator runs twice faster than the simulator without these optimizations. We will use the optimized version of the parallel simulator to perform the comparisons.

These optimizations are oriented to improve two performance aspects of GPU computing, what leads us to consider two kind of optimizations (see Chapter 3). The first one is to *emphasize the parallelism.* This optimization aims to increase the number of threads per block (to the recommended amount form 64 to 256), so it allows to fulfill warps and hide latencies. The second is to *exploit streaming bandwidth.* To do this, the data is loaded first to the shared memory, and operated there, avoiding global memory (expensive) accesses.

Next, we show the specific optimizations we have carried out for each phase:

- *Generation phase*: no optimizations were implemented here, since the implementation already satisfies the first optimization type. The second type will require a more sophisticated implementation, like the one presented in Section 5.5.

- *Exchange phase*: this phase, as it is joined with the generation phase, has no optimizations.

- *Synchronization phase*: the two optimization types are implemented here. The second optimization type is carried out by using shared memory to avoid global memory accesses. The first type is performed by increasing the number of threads per block. For our simulator, we can assume that n (number of variables, and the number of threads per block) is a small number, since the number of cells grows exponentially with respect to it. For example, let be $n = 32$. Then, 2^{32} cells will be created, what require $2^{32}(68+|cod(\varphi)|)$ bytes (in gigabytes: $272+4|cod(\varphi)|$). This number obviously exceeds the amount of available device memory. We therefore need to increase the number of threads per block, since $n < 32$ means to not fulfilling a CUDA warp. A solution here is to assign more than one cell to each thread block. This amount is $\frac{256}{n}$, being 256 the optimum number of threads per block. It allows us to reach a number of threads close to the optimum one. However, we have to take care also of having enough shared memory to load the data of every assigned cell.

- *Checking phase*: since $|cod(\varphi)|$ can be greater than 32, we then keep this number as the number of threads per block. However, we use shared memory to speedup the accesses to the elements of the array.
- *Output phase*: as in the previous phase, we also use shared memory, and the number of threads per block is kept to $|cod(\varphi)|$.

5.3 Performance analysis

In this section we test the performance of both simulators for the families $\mathbf{\Pi_{am-SAT}}$ (cell-like solution to SAT) and $\mathbf{\Pi_{tsp-SAT}}$ (tissue-like solution to SAT). Moreover, we provide a comparison of both simulators to extract conclusions of which properties are better suited for GPUs. The GPU used for the experiments is our NVIDIA GPU Tesla C1060, which has 240 execution cores and 4 GB of device memory, plugged in our computing server with two Intel i5 Nehalem processors (8 cores) and 12 GB of RAM, and using a 64-bit Ubuntu Server 10.04 as operating system.

We have developed two benchmarks (called *test 1* and *test 2*, respectively) to analyze the performance behavior of our simulators in two ways: increasing the number of threads per thread block, and increasing the number of thread blocks per grid. Both benchmarks have been generated by WinSAT program [19]. WinSAT is able to generate random SAT instances in DIMACS CNF format file by configuring several parameters: the number of variables (n), the number of clauses (m) and the number of literals per clause (we fix k for our experiments). As mentioned in Section 5.1, the number of threads per block is associated to the number objects in the input multiset, which is the same as the number of literals of the CNF formula φ (that is, $m * k$). The number of thread blocks (2^n) is equal to the number of membranes in the system, which depends on the number of variables in the CNF formula (n).

5.3.1 Cell-like simulator

In this subsection, we analyze the performance of the three simulators above presented for the family of cell-like P systems $\mathbf{\Pi_{am-SAT}}$: the sequential simulator developed in C++ (from now, *am-sat-seq*), the parallel simulator on CUDA (*am-sat-gpu*) and the hybrid simulator on CUDA (*am-sat-gpu-hyb*).

Figure 5.3 shows the experimental performance of the cell-like simulators (in a log scale) for test 1. The benchmark test 1 increases exponentially the number of literals in the CNF formula (and so, the number of objects in the P system and threads per block in the GPU) until reaching a configuration with 512 literals. It also has fixed the number of thread blocks (and membranes) to 2048 ($n = 11$). When the number of threads per block is low, the performance of GPU codes is not substantial compared with the sequential code. That is, the data parallelism is low, and we cannot take advantage of the resources available on the GPU. However, as long as the number of threads per block increases, the data parallelism of the application also increases, and therefore, the performance of our GPU codes improves notably compared to the sequential code, obtaining up to 64x of speedup between *am-sat-seq* and *am-sat-gpu*. Furthermore, the *am-sat-gpu-hyb* accelerates the simulation on the GPU, being this up to 9.63 times faster than *am-sat-gpu*. Hence, the hybrid simulator is better adapted to the GPU architecture than the parallel simulator of the P system, because it presents more data parallelism in its computation as it is described in Section 5.1.

Figure 5.4 shows the experimental performance of the simulators (in a log scale) for test 2. The benchmark test 2 increases the number of variables in the CNF formula (and so, the number of membranes in the P system and the number of blocks in the GPU in an exponential manner) until reaching a configuration with 2^{11} membranes. The number of simulated membranes is constrained by the available memory of the system. The number of literals in the formula is fixed to 256, which means 256 threads per block.

The behavior of the GPU simulators, as showed in Figure 5.4, is similar in both. This is because the execution time in the GPU codes increases exponentially depending on the number of blocks running at the same time. Once all the GPU resources have been fully occupied, the execution time increases linearly with the number of blocks. In this case, we report up to 94x of speedup

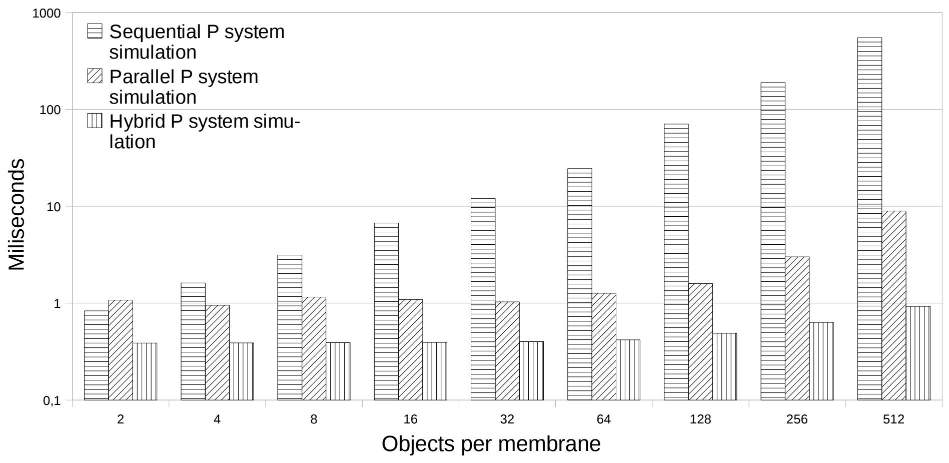


Figure 5.3: Simulation performance for *am-sat-seq*, *am-sat-gpu* and *am-sat-gpu-hyb*: Test 1 (2048 membranes)

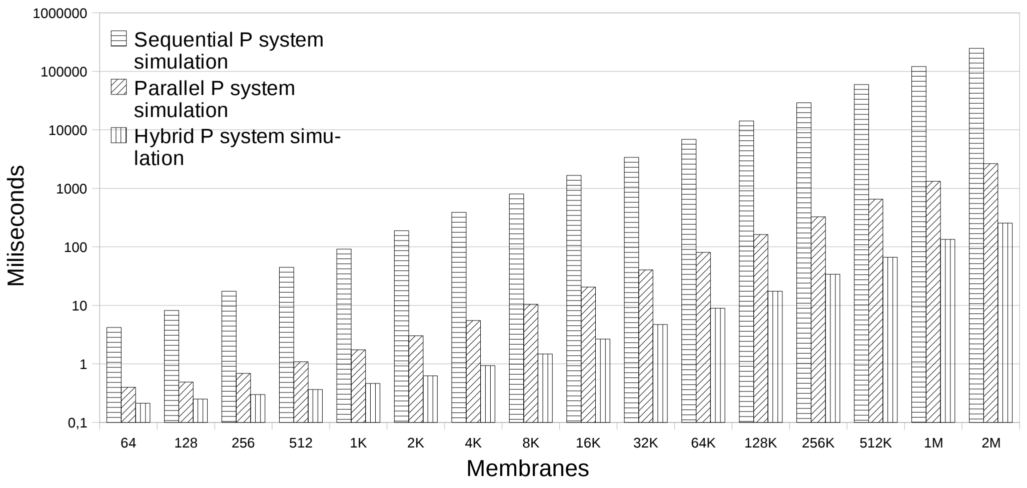


Figure 5.4: Simulation performance for *am-sat-seq*, *am-sat-gpu* and *am-sat-gpu-hyb*: Test 2 (256 Objects/Membrane)

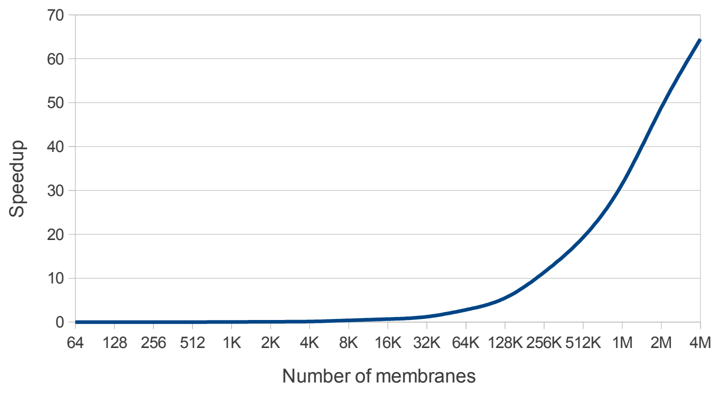


Figure 5.5: Achieved speedup running Test 2 (256 Objects/Membrane) using *am-sat-gpu* against *am-sat-seq*. GPU data management is also considered.

between *am-sat-seq* and *am-sat-gpu*. However, Figure 5.4 shows the speedup becomes a constant number of 10x when the number of membranes is greater than 128 K[2]. This is the number of blocks that fills the pipeline of the GPU in this case, having the hybrid simulator better overall performance than the parallel one.

We finalize the performance analysis of the cell-like simulators by also considering the data management (allocation and transfer) time of the GPU. Figure 5.5 shows the speedup achieved by comparing *am-sat-gpu* (with data management) and *am-sat-seq*, using Test 2. We can see that for small amounts of membranes, the speedup is below 1, what means a worst performance. However, after 32 K membranes, the speedup is 1.23x, and it is increased along with the number of membranes until 64x for 4 M membranes. This is caused by the decrease in the kernels time, and the time of handling the data is almost constant for any system size. Note that the data management performed by *am-sat-gpu* is the following: data allocation, initial configuration (only 1 membrane) transfer, and answer (object yes or not) transfer. The information of the P system during the computation is always kept on the GPU memory.

5.3.2 Tissue-like simulator

In this subsection, we analyze the performance of the two simulators developed for the family of tissue-like P systems $\mathbf{\Pi_{tsp-SAT}}$: the sequential simulator

[2]Note that we use here "K" and "M" for binary prefixes "kilo" and "mega", respectively. Therefore, 128 K=2^{17}=131072.

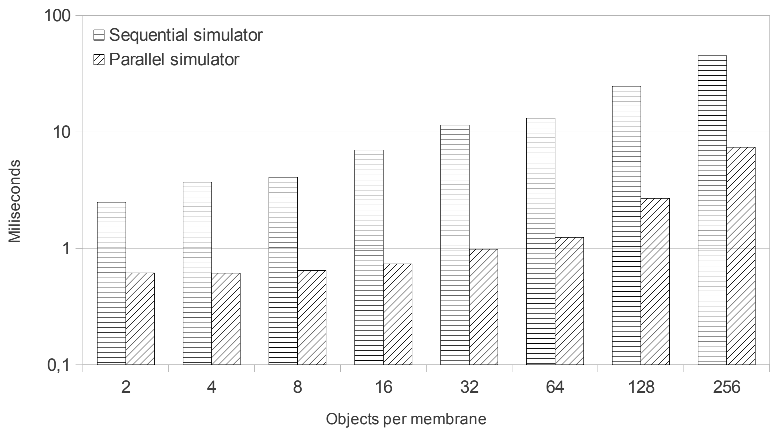


Figure 5.6: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 1 (2048 membranes)

developed in C++ (from now, *tsp-sat-seq*), and the parallel simulator on the GPU (*tsp-sat-gpu*).

For this analysis we will use also the two tests utilized for the cell-like simulators: the first one increasing the number of objects (fixing membranes to 2048), and the second increasing the number of membranes (fixing objects to 256).

Figure 5.6 shows the performance behavior of the tissue-like simulators for test 1. Only the time employed by kernels are considered for *tsp-sat-gpu*. We can see that, even for small number of objects per membrane, *tsp-sat-gpu* runs faster than *tsp-sat-seq*. A different number of objects does not produce a great impact into the performance of the parallel simulator. Note that in Section 5.2, we have introduced a different CUDA design for each phase. In this sense, the synchronization phase has been optimized to assign more cells to a thread block in order to increase the number of threads. However, the speedup is increased together with the number of objects per membrane. This means that the resources of the GPU are better utilized (e.g. 4 objects/threads does not fulfill a warp). We report the maximum speedup for 32 objects (a warp), which is of 11.6x. For 2 objects is 4x, and for 256, 6.1x.

Figure 5.7 shows the results for test 2, considering only kernel runtime for *tsp-sat-gpu*. For this case, we can observe that again, the kernels of *tsp-sat-gpu* runs faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cells 2 created by the system. For 64 membranes, the speedup is of 2x, but for 2 M cells it is of 8.3x.

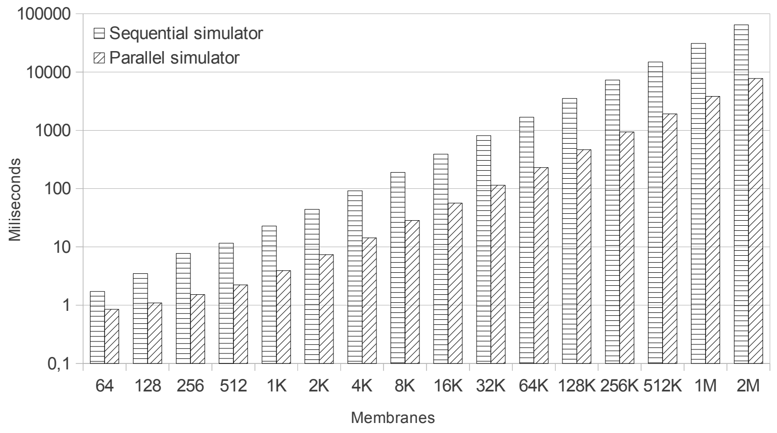


Figure 5.7: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 2 (256 Objects/Membrane)

Finally, we show the speedup achieved by using test 2 of the simulator *tsp-sat-gpu*, taking into account also the amount of time consumed by the data management (allocation and transfer). It is observed that, since the data management time is fixed for all the sizes (copy the initial multiset and retrieve the final answer), the speedup exceeds 1 only after 128 K membranes. Systems with smaller number of cells are executed slower than in the CPU, because of the data management. However, for very large systems, the speedup is as large as with only kernels. The maximum speedup we report for this simulator is given for 4 M cells, up to 10x.

5.3.3 Cell-like vs tissue-like

Next, we compare the two simulators developed for the two solutions to the `SAT` problem, using different Membrane Computing model types: cell-like (P systems with active membranes, family $\mathbf{\Pi_{am-SAT}}$) and tissue-like (tissue P systems with cell division, family $\mathbf{\Pi_{tsp-SAT}}$). The aim here is to study which model is better suited to be simulated on the GPU.

First of all, we should analyze the differences between them, to better understand the different behaviors. We highlight the following:

- Computational steps: for a given pair $< m, n >$, $m, n \in \mathcal{N}$, representing the number of clauses and variables respectively, the P systems of $\Pi_{am-SAT}(\langle m, n\rangle)$ take $5n + 2m + 3$ steps, and the tissue P systems of $\Pi_{tsp-SAT}(\langle m, n\rangle)$ require $2n + 2m + nm + 1$. We can state that the

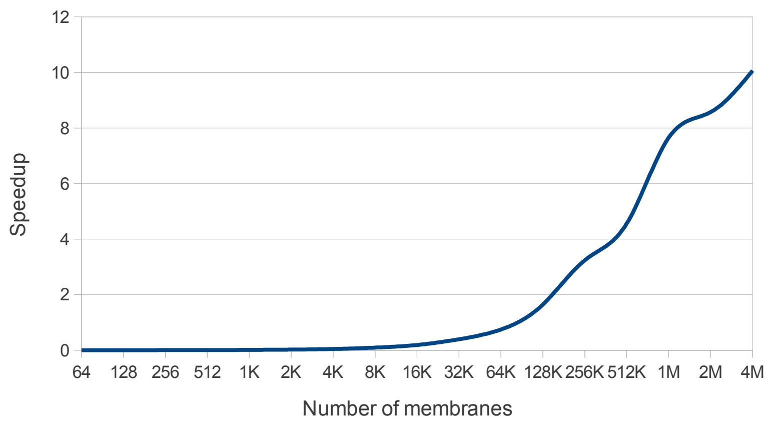


Figure 5.8: Speedup achieved running Test 2 (256 Objects/Cell) for *tsp-sat-gpu* and *tsp-sat-seq*. GPU data management is also considered.

computation of $\Pi_{tsp-SAT}(\langle m, n\rangle)$ is longer (in number of steps) than $\Pi_{am-SAT}(\langle m, n\rangle)$, if $m > 3 + \frac{2}{n} \simeq 3$.

- Phases: the cell-like simulators are based on 4 phases (implemented in 3 kernels), whereas the tissue-like simulators uses 5 phases (implemented in 4 kernels).

- Memory requirements: each membrane in the cell-like simulators is represented by a number of 32-bit integers equals to $|cod(\varphi)|$, but the tissue-like simulators use for them $2n + 4 + |cod(\varphi)|$. Thus, the tissue-like simulators use, in total, $(2n + 4)2^n$ bytes more.

Our first comparison is made for the sequential simulators. Figure 5.9 shows the speedup achieved by comparing *tsp-sat-seq* against *am-sat-seq*. We can note that the *tsp-sat-seq* simulator outperforms the cell like version, for any instance. The speedup gets frozen in 3.2x for more than 2k cells/membranes (from now, we will refer to membranes for both simulators). Although *tsp-sat-seq* has been optimized, this result is still interesting.

Finally, we compare the GPU simulators for both solutions. We can see, in Figure 5.10, that the kernels of *am-sat-gpu* outperforms *tsp-sat-gpu*, even using optimizations for the last one. This improvement implies a speedup of 2.9x. However, if we take into account the data management in the GPU, we can see that the behavior of them is almost similar. The simulator *am-sat-gpu* runs just a bit faster, but for 2M membranes, the speedup is almost 2x. This makes us to think that the data implementation of *am-sat-gpu* can be

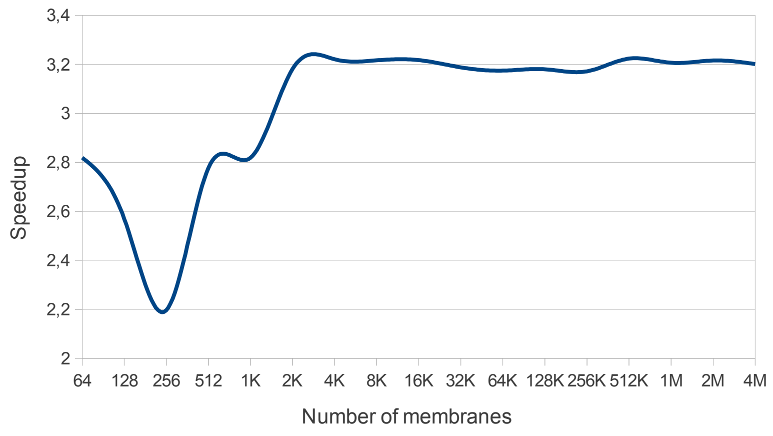


Figure 5.9: Achieved speedup running Test 2 (256 Objects/Membrane) for the sequential tissue-like (*tsp-sat-seq*) against the cell-like (*am-sat-seq*) simulator.

improved, since it requires an inferior amount of data. Recall that *am-sat-gpu* has not any GPU oriented optimizations, as *tsp-sat-gpu*.

We finish this comparison by reporting their corresponding maximum speedup, which is of 63x and 10x for the cell-like and tissue-like simulators, respectively. Therefore, using the GPU for the cell-like solutions allows to get better performance gain.

5.4 Characterizing the simulation on the GPU

In this section, we characterize the simulations described in this chapter. We will analyze two aspects introduced here: ad-hoc (less flexible) simulations, and simulation of cell-like and tissue-like models. These aspects are depicted below:

- *Ad-hoc simulation*: we achieve better performance by using a specialized (less flexible) simulator for the family of P systems $\mathbf{\Pi_{am-SAT}}$, compared with the general simulator *am-gpu* (Chapter 4). The general simulator has to check all situations that may occur in a P system with active membranes. For example, it has to control that a division rule and a dissolution rule cannot be selected at the same time in a membrane. However, the specific simulator can avoid generalities and improve the performance of the simulator by adapting the simulation algorithm, since it only works with a family of P systems. For example, it can be easily

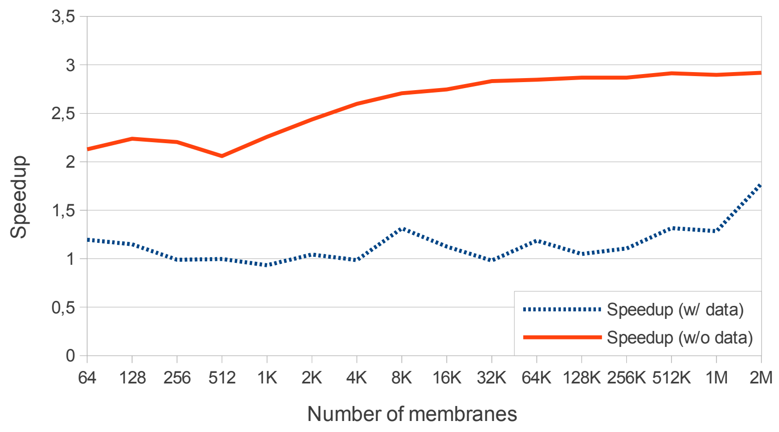


Figure 5.10: Speedup achieved running Test 2 (256 Objects/Membrane) for both parallel tissue-like (*tsp-sat-gpu*) and cell-like (*am-sat-gpu*) simulators, considering or not considering the data management.

demonstrated that, for the family $\mathbf{\Pi_{am-SAT}}$, a division rule and a send-in rule cannot ever be selected simultaneously in any membrane. In addition of adapting the algorithm, the data structures can be also optimized. For example, in this simulator, it is not necessary to maintain a data structure for rules, since the simulation algorithm already implements them in code. This way, we report up to 94x of speedup between the simulator *am-sat-gpu* and *am-sat-seq* (only considering kernels runtime), and up to 10x between both GPU codes (*am-sat-gpu* and *am-sat-gpu-hyb*). For *tsp-sat-gpu* and *tsp-sat-seq*, we report up to 10x.

- *Simulation of cell-like vs tissue-like models*: from the comparison of the simulators for the cell-like and the tissue-like solutions, we have observed that the cell-like simulations are better carried out by the GPU. Thus, we have identified two properties that have helped to improve the performance of these GPU simulators:

  – *Charges*: the model of P systems with active membranes associates charges to the membranes. They can be used to store information over the computation as well. If they are considered, and well used, for a given solution (e.g. for `SAT` to encode the truth assignment), less memory would be required (remind that the tissue-like simulator requires $2^n(2n+4)$ bytes more). Actually, having charges can help to improve the object density within membranes (described in Chapter 4). The information encoded by charges can save objects that may or not may appear simultaneously in membranes, what saves at the same time memory to represent them, and so, the number of threads to launch, working with much less objects.

– *Rules with no cooperation*: the model of P systems with active membranes defines rules with no cooperation, that is, the number of objects appearing in their left-hand sides is always 1. This property helps threads to be assigned to each rule, what also means to work with each object in parallel. Rules permitting cooperation (as in tissue P systems) require to take care of which objects are accessed by the rules (and threads). However, it is interesting to study each type of rule separately. Recall that for the general simulator *am-gpu*, the constraints of send-in, send-out, division and dissolution rules have to be considered for each membrane, what degrades parallelism on the GPU (it implies using local locks). While in the model of tissue P systems these restrictions are not presented, a general simulator for tissue models (*tsp-gpu*) can be implemented in a future to study what is better: usage of charges but restricting types of rules, or not using charges (i.e. more objects per membrane) and more (but less restrictive) parallel rules.

In this chapter we show two different results. On one hand, we demonstrate that GPUs are well suited to simulate P system due to the highly parallelism that they present in its architecture. Although the GPU is not a cellular machine, its features help the researches to accelerate their simulations. On the other hand, if the P systems based solutions are redesigned to be adapted to the GPU programming model, the performance of the simulations can be also improved.

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

5.5 Optimizing the parallel simulator on GPUs

Our simulator for the family $\mathbf{\Pi_{am-SAT}}$ has been further optimized to take advantage of modern GPU architectures [43, 46]. This was conducted through a close collaboration with specialized parallel computer architecture researchers. Next we summarize the major improvements developed, and the performance comparison.

We start by introducing a new design for the first cell-like simulator (detailed in Section 5.1). This helps to solve some bottlenecks by using the *tiling* technique. Then, we show that using a newer GPU architecture, NVIDIA's Fermi, we can achieve also a bit better performance.

5.5.1 A new design of the parallel simulator

We recall that the family $\mathbf{\Pi_{am-SAT}}$ gathers all computational features of recognizer P systems with active membranes. Among them, we highlight the theoretical double level of parallelism and non-determinism that makes P systems a computational tool to solve **NP**-complete problems in polynomial (often linear) time.

The first level of parallelism is found among membranes, that is, by executing rules inside each membrane in parallel along the computation (see Figure 5.11). The second level of parallelism is found within each membrane (see Figure 5.12). That way, the first level is coarse-grained and can be characterized by an inter-task parallelism and exploited by the number of processors available in a parallel system, whereas the second level of parallelism is fine-grained and intra-task to be exploited by the number of cores within each processor, either on multi- or many-core architectures.

The main idea of this design is to distribute the exponential amount of membranes among the processors of a parallel system. This distribution is performed in the Generation phase, in such a way that each processor will execute Generation, Synchronization, Check-out and Output phases over a portion of membranes in parallel.

Figure 5.11 shows the membrane parallelism for the execution of the Generation phase in a sequential as well as in a parallel architecture with four Compute Elements (CE). In a parallel architecture, a set of membranes is initially created by the master process, whose size is equal to the number of CEs available during the execution. Then, a membrane is sent to each CE by the master processor. This step is called *Parallel Preprocessing (PP)*, and it is developed just before the generation phase starts the computation on each

CE. This CE is represented by a processor which can later be eventually decomposed into multi- or many-cores when exploiting intra-task parallelism.

Furthermore, Figure 5.11 shows that it is known which membrane generates each one and also in which computational step. For instance, membrane two is always generated by membrane one in the first computational step, membrane three is always generated by membrane one in the second step, and so on. Finally, each node sends the partial response back to the master in order to produce the final result of the P system.

Figure 5.12 shows the second level of parallelism, internal to membranes. Once the initial data has arrived to the CE after the *Parallel Preprocessing* step, it starts the simulation of the phases. Then, resources on each CE can be exploited at its peak to cooperate for speeding up the computation of the generation and check-out phases. These resources are essentially hardware cores, but fortunately GPUs are many-core which can handle this level of parallelism at large scale using hundreds of streaming processors.

5.5.2 GPU optimization by tiling and dynamic queuing

In our first version of the GPU simulator (*am-sat-gpu*), the Generation phase is encoded as a CUDA kernel, and it starts right after the *Parallel Preprocessing* step. Therefore, each membrane is assigned to each block (which is considered to be a CE). Once membranes have been generated, the Check-out phase starts its execution in a different kernel. Each thread block loads a membrane from global memory, and then each thread checks the rules associated with this phase (Figure 5.1). For these phases, all threads within a CUDA thread block cooperate with coalesced access to device memory (threads of the same warp access the same memory segment either for reading or writing).

Blocking can be exploited on GPUs, taking advantage of the on-chip shared memory by using tiles and dynamic queues with the aim of increasing the bandwidth to device memory. Tiles decompose the computational domain into a number of independent chunks whose size fits within the shared memory, and they are implemented using the concept of CUDA blocks. This way, the whole data structure can benefit from this high-speed and low-latency memory even though it represents just a tiny fraction of the algorithm requirements. Furthermore, dynamic queues schedule jobs at run-time depending on the ever changing workload supported by each GPU multiprocessor, always assigning the next computational block to the less overloaded one.

The simulation has to perform a *Block Preprocessing* (BP) step before starting the generation phase itself, which is implemented through a CUDA

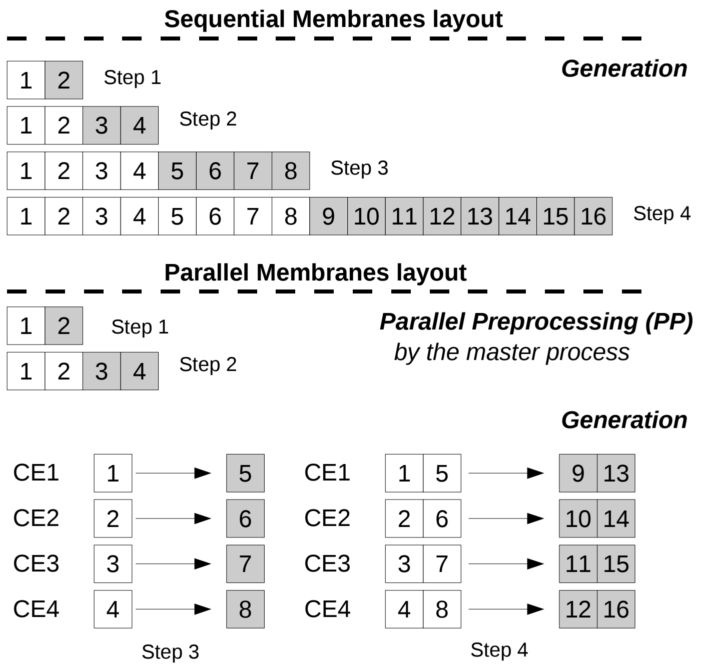


Figure 5.11: Sequential and parallel membranes generation on four Compute Elements (CE). The *Parallel Preprocessing (PP)* is required to set up the parallel execution prior to its starting on each CE. This CE is represented by a processor (die) which can later be eventually decomposed into multi- or many-cores depending on target architecture.

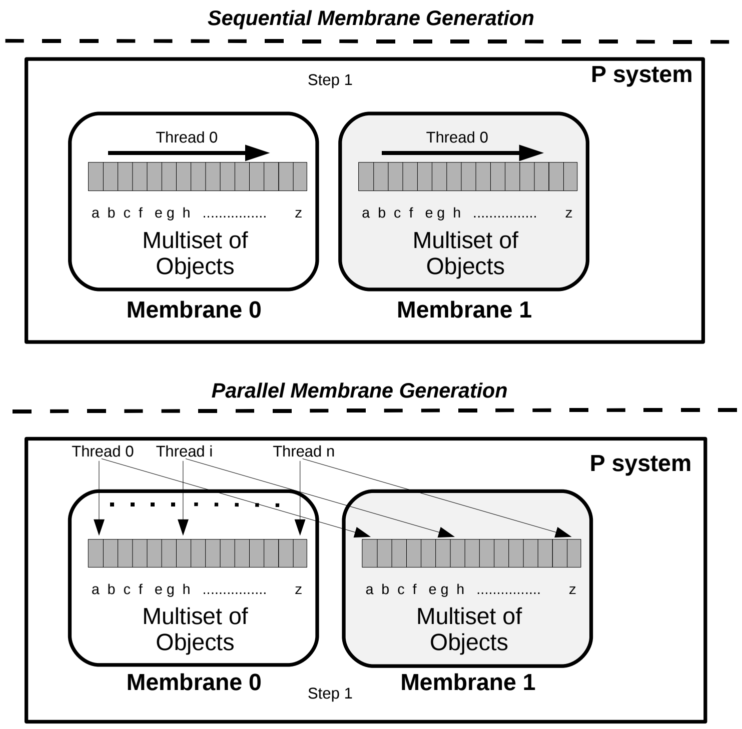

Sequential Membrane Generation
P system
Step 1
Thread 0
Thread 0
a b c f e g h .... z
a b c f e g h .... z
Multiset of Objects
Multiset of Objects
Membrane 0
Membrane 1
Parallel Membrane Generation
Thread 0
Thread i
Thread n
P system
a b c f e g h .... z
a b c f e g h .... z
Multiset of Objects
Multiset of Objects
Membrane 0
Step 1
Membrane 1

Figure 5.12: Sequential and parallel execution when creating the exponential workload shows the second level of parallelism in P systems, that internal to membranes. P system rules are applied running on hardware cores, and many-core GPUs translates this into massive parallelism using hundreds of cores.

kernel where a set of membranes are partially created, placing them apart from each other at a block size distance. This can be considered the GPU implementation of the PP. An additional kernel is created at the end of the simulation to perform the generation locally to each block, followed by the Check-out phase. Each thread on a thread block cooperates for an efficient load from global memory to shared memory of the initial membrane generated by the *Block Preprocessing* step.

5.5.3 Parallel simulation on multi-GPU systems

Within a GPU-based cluster, GPUs cannot interact with each other, and a CPU process has to be created to monitor each GPU independently. The master process creates four CPU threads or contexts (for four GPUs plugged in a system) to invoke the execution on each GPU and manage its resources (i.e allocate device memory, move data to/from the GPU, and so on). Resources created on each CPU thread are not accessible by any other thread, and there is no explicit initialization function for the runtime API [5], which makes hard to measure time in a reliable manner, particularly on multi-GPU environments.

For the GPU case, the master process performs the *Parallel Preprocessing* step as usual, generating as many membranes as GPUs are involved in the simulation, and performing the assignment. At a starting point, the simulation barely exploits GPU resources because the computation begins with a single CUDA thread block (which represents the membrane generated by the *Parallel Preprocessing* step). However, the number of CUDA thread blocks grows exponentially along with the number of membranes, and GPU resources are fully utilized at early stages of the simulation.

5.5.4 Performance analysis

Table 5.1 summarizes the performance for all software implementations and hardware enhancements exploited through parallel strategies deployed along this section. For the smallest benchmark, GPU performance achieves an impressive speedup factor which exceeds three orders of magnitude, and this factor even increases with the problem size. Acceleration reaches its peak for the highest number of membranes that can fit into video memory given our hardware constraints (that is, the $n = 21$ variables case).

In general, all speedups increase with the problem size, being more remarkable the scalability shown by the multiprocessor version: 3.87x when moving to 4 GPUs means that are barely 3% below the optimal line. This outstanding

Table 5.1: Summary for the execution times (in milliseconds) and speed-up attained by the set of implementations outlined in this section.

Code version	Number of membranes (problem size)				
	2^{13}	2^{15}	2^{17}	2^{19}	2^{21}
1. CPU baseline simulator	800.47	3382.49	14211.80	59521.80	247467.00
2. Running on Tesla C1060 GPU	0.82	2.90	11.16	44.69	171.04
3. Running on Tesla M2050 GPU	0.62	2.30	8.71	33.16	127.85
4. With tiling on Tesla M2050	0.37	1.24	4.65	18.27	73.23
5. With tiling on 4 Tesla M2050	0.17	0.39	1.26	4.86	18.89
Departure GPU speed-up (2 vs. 1)	976.18x	1166.37x	1273.45x	1331.88x	1446.83x
Speedup on M2050 GPU (3 vs. 2)	1.32x	1.26x	1.28x	1.34x	1.33x
Speedup with tiling (4 vs. 3)	1.67x	1.85x	1.87x	1.81x	1.74x
Speedup on 4 GPUs (5 vs. 4)	2.17x	3.17x	3.69x	3.75x	3.87x
Overall speed-up factor (5 vs. 1)	**4708x**	**8673x**	**11279x**	**12247x**	**13100x**

behavior can find a good rationale in the overall amount of cache available for running the code, which is multiplied by a factor of four in a memory-bound algorithm like ours. On the down side, one might expect more from the new Tesla M2050 GPU: 448 cores running at 1.147 GHz deliver 513 GFLOPS, while those 240 cores of the Tesla C1060 running at 1.296 deliver 311 GFLOPS. Analytically, we have a 1.65x speed-up factor in raw processing power; in practice, however, our improvement fluctuates around 1.30x, confirming the theory that:

1. *P systems simulations are not that demanding on arithmetic intensity.*
2. *The bottleneck lies more on memory accesses.*

Considering the largest problem size and amount of parallelism we were able to expose, we reach a minimum execution time of 18.89 milliseconds on four Tesla M2050 GPUs, each endowed with 448 cores for a total of 1792 GPU streaming cores. This represents an improvement factor of 13100x with respect to the departure time given by the original simulator, that is, more than four orders of magnitude. In other words, a P system simulation which would take an entire day on the Intel Xeon CPU can be processed in 6.5 seconds within our Tesla S2050 GPU Computing System.

The multi GPU environment shines with a linear speed up along with the number of GPUs. This result is expected as the computational workload is evenly distributed on GPUs. Furthermore, there is more room on each GPU memory space, so higher workloads may be executed.

5.6 Parallel simulation on supercomputers

In [42], an alternative parallel simulator for the family $\mathbf{\Pi_{am-SAT}}$ is introduced. This simulator is implemented for clusters, which normally contains shared memory machines conforming distributed memory systems. The specific cluster used for the experiments was the supercomputer installed at the Supercomputing Center of the Region of Murcia (Spain). The developed simulators, one focused on shared memory, and the other on distributed memory, were designed following the strategies explained in Section 5.5.

The simulator on the shared memory system was implemented using OpenMP. The shared memory space is equally distributed among the processes considered, and the master process performs the *Parallel Preprocessing* step by creating as many membranes as threads are involved in the computation.

The simulator on the distributed memory system was programmed using MPI. In this case, each process allocates memory on its own and private memory space. The master process also performs the *Parallel Preprocessing* step, creating as many membranes as number of processors involved in the computation. Then, membranes are distributed along the processors by using MPI. Once the initial data arrives to each node, the P system computation is developed as in the shared memory case.

Using this technique, the parallel efficiency of the shared memory architecture is improved, but the OpenMP simulator reaches the lowest performance as the pressure on shared resources increases with the number of cores. On the positive side, this was the only platform able to execute all benchmarks due to a higher memory availability. The distributed memory system exhibits good scalability with the number of processors, which can be partially explained by the low number of communications required by the simulations.

This work concludes that GPUs constitute a good platform to simulate P systems solving `SAT` in terms of execution time. The two levels of parallelism that P systems exhibit, one at region level and another one at system level, were exploited by the GPU implementation to reach speedup factors around 10x versus distributed memory and around 40x versus shared memory when four processors are used on a given platform.

5.7 Conclusions

In this chapter we have described the PCUDASAT framework (a subproject of PMCGPU [17]). It includes simulators specially designed for a family of P systems with active membranes solving `SAT` in linear time (Chapter 1):

- *A sequential simulator*: it is written in C++, and it is optimized for this family of P systems. The rules of the P systems are represented directly in source code, so the given input is only the P system input multiset; that is, the code of the formula to be processed. Therefore, the simulator acts as a `SAT` solver but with a Membrane Computing based engine. The simulation algorithm is based on the four phases identified in these P systems computation: Generation, Synchronization, Check-out and Output. Each phase is executed separately.

- *A parallel simulator*: it is based on CUDA, and works similar than the sequential simulator. Again, the double parallelism of these P systems is represented in the double parallelism of GPUs; that is, each CUDA thread block works with an elementary membrane, and each CUDA thread is assigned to an object of the input multiset. Looking at the `SAT` solution level, each thread block is assigned to a truth assignment, and each thread to each literal of the CNF formula. Each phase of the simulation algorithm is implemented on separated CUDA kernels.

- *A hybrid parallel simulator*: based on CUDA, this simulator is similar to the previous parallel simulator. However, the Check-out phase kernel is substantially different. The solution made by the P systems is sequential regarding the clauses (each one is checked at a time). The solution in this simulator is parallel, using GPU techniques. Therefore, the simulator does not fully reproduce the P system computation at this phase, but the speedup is much improved. Moreover, further optimizations were made in the other phase kernels.

The provided simulation framework is therefore inflexible (only supports one family of P systems), but thanks to this, the performance and scalability is much increased. Moreover, further development was made for new GPU architectures (Fermi), multi-GPU systems and clusters. Experiments showed that P systems simulations are not that demanding on arithmetic intensity, and that the bottleneck lies more on memory accesses.

Part III

Parallel Simulation applied to Computational Models in Biology

"Essentially, all models are wrong, but some are useful."
George E. P. Box

"Wolves directly affect the entire ecosystem, not just moose populations, their main prey, because less moose equals more tree growth."
Rolf Peterson, Purdue University

6 Simulation Algorithms for Population Dynamics P Systems

Membrane Computing covers both the study of the theoretical basis for the models as well as the applications of the model to various fields including computational Systems Biology [47, 152, 128, 121, 51], and Ecosystem Dynamics [51, 34, 53]. *Population Dynamics P Systems*, or PDP systems, is a P system based framework for modeling population dynamics [50, 33, 99]. It enables simultaneous evolution of a high number of species, as well as the management of a large number of auxiliary objects. It also facilitates model development that can be easily interpreted by simulation software.

So far, several algorithms have been developed in order to capture the semantics defined by the modeling framework. A comparison on the performance of some of these algorithms can be found in [52]. These algorithms select rules according to their associated probabilities, while keeping the maximal parallelism semantics of P systems.

In this chapter we introduce the formal framework for Population Dynamics modeling, or PDP systems (Section 6.1). We then discuss about the simulation algorithms for this framework, and also provide the description of each one developed to date. The first simulation algorithm was called *BBB (Binomial Block Based algorithm)*, and it is described in Section 6.4. The two new algorithms we have defined are called *DNDP (Direct Non-Deterministic* *distribution with Probabilities)* (Section 6.5) and *DCBA (Direct distribution based on Consistent Blocks Algorithm)* (Section 6.6). Finally, they both are experimentally validated through using a model for a real ecosystem related to the Bearded Vulture in the Pyrenees 6.7.

6.1 Population Dynamics P systems

Population Dynamics P systems are a variant of *multienvironment P systems with active membranes* [51], a model with a network of environments, each of them containing a P system where features such as electrical charges associated with membranes which describe specific properties in a better way, are used. All P systems share the same *skeleton*, in the sense that they have the same working alphabet, the same membrane structure and the same set of rules. Nevertheless, in this framework each rule has associated a probability function which can vary for each environment.

Definition 6.1. *A Population Dynamics P system (PDP) of degree (q, m), $q, m \geq 1$, taking $T \geq 1$ time units, is a tuple*

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

where:

- *$G = (V, S)$ is a directed graph. Let $V = \{e_1, \dots, e_m\}$.*
- *Γ and Σ are alphabets such that $\Sigma \subsetneq \Gamma$.*
- *T is a natural number.*
- *$\mathcal{R}_E$ is a finite set of rules of the form $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$, where $x, y_1, \dots, y_h \in \Sigma$, $(e_j, e_{j_l}) \in S, 1 \leq l \leq h$, and $p_r : \{1, \dots, T\} \longrightarrow [0, 1]$ is a computable function such that for each $e_j \in V$ and $x \in \Sigma$, the sum of functions associated with the rules of the type $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ is the constant function 1.*
- *μ is a rooted tree labeled by $1 \leq i \leq q$, and by symbols from the set $EC = \{0, +, -\}$.*
- *$\mathcal{R}$ is a finite set of rules of the form $u[v]_i^{\alpha} \rightarrow u'[v']_i^{\alpha'}$, where $u, v, u', v' \in M_f(\Gamma)$, $u + v \neq \emptyset$, $1 \leq i \leq q$ and $\alpha, \alpha' \in \{0, +, -\}$, such that there is no rules $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ and $u[v]_i^{\alpha} \rightarrow u'[v']_i^{\alpha'}$ having $x \in u$.*

- *For each $r \in \mathcal{R}$ and $1 \leq j \leq m$, $f_{r,j} : \{1, \dots, T\} \longrightarrow [0, 1]$ is a computable function such that for each $u, v \in M_f(\Gamma)$, $1 \leq i \leq q$, $\alpha, \alpha' \in \{0, +, -\}$ and $1 \leq j \leq m$, the sum of functions $f_{r,j}$ with $r \equiv u[v]_i^{\alpha} \rightarrow u'[v']_i^{\alpha'}$, is the constant function* 1.
- *For each i, j $(1 \leq i \leq q, 1 \leq j \leq m)$, $\mathcal{M}_{i,j}$ is a finite multiset over Γ.*

A Population Dynamics P system defined as above can be viewed as a set of m environments $e_1, \dots, e_m$ interlinked by the edges from the directed graph G. Each environment e_j only can contain symbols from the alphabet Σ and all of them also contain a P system skeleton, $\Pi_j = (\Gamma, \mu, \mathcal{M}_{1,j}, \dots, \mathcal{M}_{q,j}, \mathcal{R})$, of degree q, where:

(a) Γ is the working alphabet whose elements are called objects.

(b) μ is a rooted tree which describes a membrane structure consisting of q membranes injectively labeled by $1, \dots, q$. The skin membrane (the root of the tree) is labeled by 1. We also associate electrical charges from the set $\{0, +, -\}$ with membranes.

(c) $\mathcal{M}_{1,j}, \dots, \mathcal{M}_{q,j}$ are finite multisets over Γ, describing the objects initially placed in the q regions of μ, within the environment e_j.

(d) $\mathcal{R}$ is the set of evolution rules of each P system. Every rule $r \in \mathcal{R}$ in Π_j has a computable function $f_{r,j}$ associated with it. For each environment e_j, we denote by $\mathcal{R}_{\Pi_j}$ the set of rules with probabilities obtained by coupling each $r \in \mathcal{R}$ with the corresponding function $f_{r,j}$.

Therefore, there is a set $\mathcal{R}_E$ of communication rules between environments, and the natural number T represents the simulation time of the system. The set of rules of the whole system is $\bigcup_{j=1}^{m} \mathcal{R}_{\Pi_j} \cup \mathcal{R}_E$.

The *semantics* of Population Dynamics P systems is defined through a non deterministic and synchronous model (in the sense that a global clock is assumed). Next, we describe some semantics aspects of these systems.

An evolution rule $r \in \mathcal{R}$, of the form $u[\,v\,]_i^{\alpha} \rightarrow u'[\,v'\,]_i^{\alpha'}$, is applicable to each membrane labeled by i, whose electrical charge is α, and it contains the multiset v, and its parent contains the multiset u. When such rule is applied, the objects of the multisets v and u are removed from membrane i and from its parent membrane, respectively. Simultaneously, the objects of the multiset u' are added to the parent membrane i, and objects of multiset v' are introduced in membrane i. The application also replaces the charge of membrane i to α'. In each environment e_j, the rule r has associated a probability function

$f_{r,j}$ that provide an index of the applicability when several rules compete for objects. In this model, the cooperation degree is given by $|u| + |v|$.

A rule $r \in \mathcal{R}_E$, of the form $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$ is applicable to the environment e_j if it contains object x. When such rule is applied, object x passes from e_j to $e_{j_1}, \dots, e_{j_h}$ possibly modified into objects $y_1, \dots, y_h$ respectively. At any moment t $(1 \leq t \leq T)$ for each object x in environment e_j, if there exist communication rules of the type $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$, then one of these rules will be applied. If more than one such a rule can be applied to an object at a given instant, the system selects one randomly, according to their probability which is given by $p_r(t)$.

For each j $(1 \leq j \leq m)$ there is just one further restriction, concerning the consistency of charges: in order to simultaneously apply several rules of $\mathcal{R}_{\Pi_j}$ to the same membrane, all the rules must produce the same electrical charge in the membrane in which to be applied. Thus, we will say that the rules of the system, in this computational framework, are applied in a *non-deterministic*, *maximally consistent* and *parallel* way.

An *instantaneous description* or *configuration* of the system at any instant t is a tuple of multisets of objects present in the m environments and at each of the regions of each Π_j, together with the polarizations of the membranes in each P system. We assume that all environments are initially empty and that all membranes initially have a neutral polarization. We assume a global clock exists, synchronizing all membranes and the application of all the rules (from $\mathcal{R}_E$ and from $\mathcal{R}_{\Pi_j}$ in all environments).

In each time unit we can transform a given configuration in another configuration by using the rules from the whole system as follows: at each transition step, the rules to be applied are selected in a non–deterministic way according to the probabilities assigned to them, and all applicable rules are simultaneously applied in a maximal way. In this way, we get *transitions* from one configuration of the system to the next one.

A *computation* is a sequence of configurations such that the first term of the sequence is the initial configuration of the system, and each non-initial configuration of the sequence is obtained from the previous configuration by applying rules of the system in a maximally consistent and parallel manner with the restrictions previously mentioned.

Remark: PDP systems verify four properties (relevance, understandability, extensibility and computability) that are desirable to have any computational model [142].

- *Relevance:* a computational model must be relevant, capturing the essential features of the investigated system. It should present a unifying specification of the different components that constitute the system, the interaction between them, their dynamic behavior as well as the physical structure of the system itself.

- *Understandability:* the abstract formalisms used to model complex systems should correspond to the informal concepts and ideas which are used by experts in the population under study.

- *Extensibility:* in a computational model, we should be able to easily identify the different components and characteristics of the systems that are essential in the context of the management or scientific problem to be solved or comprehended [85]. So they can be rearranged, duplicated, composed, etc. in an easy way to produce other models. Models of complex systems should also be extensible to higher levels of organizations.

- *Computability and mathematical tractability:* it should be possible to implement or simulate a model in a computer so that we can run simulations to study the dynamics of the system by manipulating experimental conditions in the model. In this manner we can experimentally validate the model and also study the behavior of the system under different scenarios of interest. The computability of the model also allows us to perform model checking and similar techniques to infer and study qualitative and quantitative properties of the system in an automatic way. In this respect, the model should be mathematically tractable. That is, it should be possible to perform mathematical analysis on it.

6.2 Applications to real ecosystems

Recent research works have been focused on using P systems as a modeling tool for biological phenomena, within the framework of Computational Systems Biology [25, 26, 47, 121, 128, 152] and Population Dynamics [51, 33, 50], being complementary and an alternative to more classical approaches (i.e. ODEs, Petri Nets, etc.). They are used as a formalism for describing, and simulating, the behavior of biological systems, with the advantage of providing a discrete and modular formal model [51].

In [33], a P systems based general framework for modeling ecosystems dynamics is presented. This computational modeling has been used for real ecosystems, such as the scavenger birds in the Catalan Pyrenees [34] and the zebra mussel in Ribarroja reservoir [33]. These P system based models are able to analyze the simultaneous evolution of a high number of species, and they can also handle a large number of auxiliary objects representing e.g. grass, biomass, etc.

The aim of this P system based modeling tool is to help the ecologists to adopt *a priori* management strategies for the real system by executing virtual experiments. However, since no *in vivo* nor *in vitro* implementations of P systems are yet available, the computation and analysis of these models is currently performed by simulators. Thus, the design of simulators and other related software tools becomes a critical point in the process of model validation, as well as for virtual experimentation. Indeed, after running virtual experiments it is often required to add or remove some ingredients or characteristics from the initial blueprint of the model. These modifications are quite simple to implement thanks to the modularity of these models.

The modeling framework mentioned above had an associated software tool, called MeCoSim, providing, among others, the following features: a graphical user interface (users be ecologists or model designers), definition of model and ecosystem's parameters, execution of simulations, creation of statistical data in form of tables, graphs, etc. [125]. The core of this application is *pLinguaCore* [70], a software library for Membrane Computing. The models are defined by plain-text files using P-Lingua specification language. The application loads that file, configures the corresponding parameters, *pLinguaCore* to execute, and collects the results of the simulation.

Inside the *pLinguaCore* library, many simulation algorithms are defined for the different supported P system variants. Specifically, for PDP systems, the implemented simulation algorithms is based on binomial distribution and blocks of rules. This simulation algorithm is efficient enough only for small and medium instance sizes, but it lacks maximal consistency application of rules and calculation of probability functions. This simulation algorithm is called *BBB*, and is described in Section 6.4, together with the corresponding concepts and notions. Two new algorithms were defined for this work, and they are DNDP and DCBA, described in Section 6.5 and Section 6.6, respectively.

6.3 Simulation algorithms for PDP systems

In order to simulate a P system, we have seen in the previous chapters that it is necessary to define a simulation algorithm. It has to be able to provide a representation of the syntactical elements and to faithful reproduce the semantics of the model. We can assume that the syntactical elements refer to all the data elements of P systems: graph associated membrane structure, multisets of objects, rules, etc. The semantics of the model refers to how the system evolve, that is, how rules are applied.

Therefore, the aim of the developed simulation algorithms for PDP systems is to serve as inference engines to reproduce the semantics of the model, in a reliable and accurate way. Before introducing the algorithms, we should note some semantical properties that are desirable on the simulation of PDP systems. They are defined to adjust the behavior of the models according to how the experts and designers think about the modeled phenomena. Others come from the fact that we are actually using P systems and Markov Chains based probabilistic systems.

- *Probabilistic behavior*: PDP systems aims to reproduce the stochasticity of nature processes according to given probabilities. This random behavior is directly associated to rules by the number of times they are going to be applied. To do this, a simulation algorithm should calculate random numbers, in such a way that each time the system is simulated the reproduced computation should differentiate (according to the probabilities). Thus, statistical studies over several parallel and independent simulations should fit the expected mean and variance.

- *Resource competition*: since their cooperation degree is greater than 1, rules are utilized to dictate how groups of elements and individuals evolve in the model. According to the semantics of PDP systems (and the way experts think on population dynamics), what happens to the same group of individuals must be predefined, and the probability of each option has to sum 1. However, the same elements can participate in different groups, and so, they can participate in different evolutions. This issue is also known as competition for resources (by the evolution of groups viewpoint). The behavior for this is not such explicitly specified, and can be done by several approaches. It is actually the tricky part of simulation algorithms. Some algorithms implement a random way to distribute the resources. Others assume that nature has a tendency towards proportionally distribute resources to such groups were less elements are required (the more required, the more energy is needed).

- *Maximality of the model*: rules are applied in a maximally parallel way, as traditionally in P systems. This property can, in fact, help to control the evolution of every element, even if they remain unchanged or disappear. We will see that, sometimes, an extra phase assuring maximality will be required.

- *Consistency of rules*: the property of maximality is, however, restricted to those rules that produce consistent states of the system; that is, two rules producing a different charge to the same membrane cannot be simultaneously applied.

We have only mention the most important semantical properties that have to be captured by the simulation algorithms, but we also desire an extra condition: efficiency. A simulation algorithm that require an exponential growth of time or space will not be practical. Indeed, we look to procedures that can help to ecology experts to run their models. However, we will see that the simulation algorithms can be also implemented in parallel platforms to look for good performance.

6.4 Binomial Block Based algorithm (BBB)

In this section we describe the first simulation algorithm developed for PDP systems, presented in [33]. It is also available in the current release of *pLinguaCore* library [70], and it is implemented following a strategy based on the binomial distribution and blocks of rules.

Let us consider a PDP system of degree (q, m) with $q \geq 1,\ m \geq 1$, taking T time units, $T \geq 1$, $\Pi = (G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{i,j} : 1 \leq i \leq q, 1 \leq j \leq m\})$, as defined in section 6.1. The computation of the system is a sequence of configurations C_t, $0 \leq t \leq T$ constructed by the application of rules from $R = \bigcup_{j=1}^{m} \mathcal{R}_{\Pi_j} \cup \mathcal{R}_E$.

The algorithm introduced in this section is based on the concept of rule blocks. Next, we will define the related notions with left and right hand sides (Definitions 6.2 and 6.3), and blocks of rules (Definition 6.4).

Definition 6.2. *The left and right-hand sides of the rules are defined as follows:*

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

- *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 $charge(RHS(r)) = \alpha'$.*

The charge of $LHS(r)$ is the second component of the tuple (idem for $RHS(r)$).

Definition 6.3. *Given $x \in \Gamma$, $l \in H$, and $r \in \mathcal{R}$ such that $LHS(r) = (i, \alpha, u, v)$, we say that (x, l)* appears *in $LHS(r)$ with multiplicity k in any of the following cases:*

- *$l = i$, and x appears in the multiset v with multiplicity k.*
- *l is the label of the parent membrane of membrane i, and x appears in the multiset u with multiplicity k.*

Definition 6.4. *Rules from $\mathcal{R}$ and $\mathcal{R}_E$ can be classified in blocks as follows: (a) the block associated to (i, α, u, v) is $B_{i,\alpha,u,v} = \{r \in \mathcal{R} : LHS(r) = (i, \alpha, u, v)\}$; and (b) the block associated with (e_j, x) is $B_{e_j,x} = \{r \in \mathcal{R}_E : LHS(r) = (e_j, x)\}$.*

Additionally, we also provide the notion of consistent rules and set of rules, in Definition 6.5 and Definition 6.6, respectively.

Definition 6.5. *Two rules, $r_1 \equiv u_1[v_1]_{i_1}^{\alpha_1} \rightarrow u_1'[v_1']_{i_1}^{\alpha_1'}$ and $r_2 \equiv u_2[v_2]_{i_2}^{\alpha_2} \rightarrow u_2'[v_2']_{i_2}^{\alpha_2'}$, are consistent if and only if $(i_1 = i_2 \wedge \alpha_1 = \alpha_2 \rightarrow \alpha_1' = \alpha_2')$*

Definition 6.6. *A set of rules is consistent if every pair of rules of the set is consistent.*

That is, the concept of consistent set of rules aims to capture the next idea: all rules of the set can be simultaneously applied if given the necessary conditions in the corresponding configuration. Thereby, the semantics of PDP systems is attributed to require a maximally consistent multiset of selected rules to move from one configuration to another one in a computation step. In fact, this multiset determines a consistent set of rules (in this case we will say that a maximally consistent multiset of applicable rules have been selected).

After the introduction of required notions, the pseudocode of the *Binomial Block Based* (BBB) simulation algorithm is shown below (Algorithm 6.4.1). Roughly speaking, it is divided into two main phases: *selection* and *execution.*

In the first one, the rules are selected, determining the number of times that they will be applied in the simulated step, according to their left-hand sides and the available objects in the configuration. In the second phase, the chosen rules are applied the elected number of times by adding the multisets of the right-hand side to the configuration, and possibly changing the polarization of membranes.

Algorithm 6.4.1 BBB MAIN PROCEDURE

```
Input: A PDP system of degree $(q, m)$ with $q, m \geq 1$, and $T \geq 1$.
1: for $t \leftarrow 0$ to $T - 1$ do
2:     $R_{sel} \leftarrow$ SELECTION PHASE $(C_t, R)$              ▷ (Algorithm 6.4.2)
3:     $C_{t+1} \leftarrow$ EXECUTION PHASE $(C_t, R_{sel})$       ▷ (Algorithm 6.4.3)
4: end for
```

The input data for selection phase consists of the configuration in time t, C_t, and the set of defined rules $\mathcal{R}$. The output data of this stage is a multiset of the form $R_{sel} = \{\langle r, n_r \rangle\}$, where $r \in \mathcal{R}$ and $n_r \in \mathbb{N}$ is the number of times to be executed. Note that adding new objects before finishing selection phase could mislead the algorithm yielding inconsistent states, since the algorithm could use such new objects for triggering rules of the PDP system that were not supposed to be applied until the next transition step. The pseudocode of the selection phase is as shown in Algorithm 6.4.2.

The selection mechanism starts from the assumption that rules in $\mathcal{R}$ can be classified into blocks of rules having the same left-hand side, following the definitions 6.2 and below. Recall that, according to the semantics of the model, the sum of probabilities of all the rules from a block is always equal to 1 – in particular, rules with probability equal to 1 form individual blocks. Note that rules with overlapping (but different) left-hand sides are classified in different blocks.

The rules selection mechanism iterates through the randomly ordered set of blocks of rules, and within each block, rules are selected in a maximal way (i.e. they will consume as many objects from the configuration as possible). More precisely, given a block, the number of times that a rule r is applied is determined according to a binomial distribution $B(N, p_r)$ (see Section 7.5.2 for further information), where N is the number of copies of the multisets in $LHS(r)$ contained in the configuration, and p_r is the probability associated with r.

In order to guarantee that we do not consume more resources than what it is available, after determining the number of applications for the first rule

Algorithm 6.4.2 BBB SELECTION PHASE

```
1: Rules from $R = \mathcal{R}_E \cup \mathcal{R}_{\Pi_j}, 1 \leq j \leq m$ are clustered into blocks $B_l$ according to Definition 6.4.
2: Let $F_b(N,p)$ be a function that returns a random natural number using the binomial distribution $B(N,p)$.
3: A random order on the family of all blocks of rules is considered.
4: for all blocks of rules $B_l = \{r_1, \ldots, r_s\}$, according to the selected random order do
5:     A random order on the rules $\{r_1, \ldots, r_s\}$ is chosen.
6:     $(h, \alpha, u, v) \leftarrow l$ and $c_{r_1}, \ldots, c_{r_s}$ are the probabilistic constants of $\{r_1, \ldots, r_s\}$.
7:     $N \leftarrow max\{n : r_1^n$ is applicable to configuration $C_t\}$.
8:     if $N > 0$ then
9:         $d \leftarrow 1$
10:        for $k \leftarrow 1$ to $s - 1$, according to the selected order do
11:            $c_{r_k} \leftarrow \frac{c_{r_k}}{d}$
12:            $n_{r_k} \leftarrow F(N, c_{r_k})$
13:            $N \leftarrow N - n_{r_k}$
14:            $q \leftarrow 1 - c_{r_k}$
15:            $d \leftarrow d * q$
16:        end for
17:        $n_{r_s} \leftarrow N$
18:        for $k \leftarrow 1$ to $s$ do
19:            $C_t \leftarrow C_t - n_{r_k} * LHS(r_k)$
20:            $R_{sel} \leftarrow R_{sel} \cup \langle r_k, n_{r_k} \rangle$
21:        end for
22:    end if
23: end for
24: return $R_{sel}$
```

Algorithm 6.4.3 BBB EXECUTION PHASE

```
1: for all $\langle r, n \rangle \in R_{sel}$ do
2:     if $n > 0$ then
3:         $C_t \leftarrow C_t + n * RHS(r)$
4:         Update charges of membranes from $C_t$ using $RHS(r)$
5:     end if
6: end for
7: return $C_t$
```

of the block ($n_r = B(N, p_r)$), the first parameter of the binomial distribution is reduced so that for the next rule the maximum number of applications matches the available objects ($N \leftarrow N - n_r$). This is done for all rules in the block (randomly ordered) except for the last one, which skips the binomial distribution and takes directly all the remaining applications. It is worth noting that this process is equivalent to calculating a multinomial distribution.

When the selection phase finishes, it returns the multiset R_{sel} containing the applicable rules and the number of times they will be executed in the simulated step. Execution phase (Algorithm 6.4.3) iterates through the selected rules, and adds the corresponding right-hand sides to the configuration (taking into account the number of applications).

This simulation algorithm is useful for the majority of models, such as [34, 33]. However, it has several disadvantages that restrict the P systems models to be correctly simulated:

- It does not handle rules with intersections on their left-hand side (object competition). Rules with partial (no total) overlapping on their left-hand sides are classified into different blocks, so the common objects will not be distributed since these blocks are maximally executed.

- It does not check the consistency of charges in the selection of rules. As seen in Section 6.1, rules are executed in a maximally consistent way. If there are rules changing the charge of a membrane, and others changing to a different one, they cannot be executed in the same step. Fixing one value to the charge, all the rules consistent to it must be, if possible, executed.

- It does not evaluate probabilistic functions related to rules. Only constant probabilities are considered, what is not the case of the new P systems based models.

These constraints lead to develop new simulation algorithms, looking for flexibility on the semantics to simulate and providing supplementary features for the new requirements of models (maximal consistency and probabilistic functions).

6.5 Direct Non-Deterministic distribution with Probabilities algorithm (DNDP)

In this section we describe the pseudocode of one of our proposed simulation algorithms for PDP systems. It is called DNDP, which comes from the inspiration on the DND algorithm [116], and the extension for probabilities. The input is a PDP system of degree (q, m), taking T time units. The algorithm simulates only one computation of the PDP system (actually, only T transition steps), by executing rules in a non-deterministic maximal consistent parallel way.

6.5.1 Inspiration: Direct Non-deterministic Distribution algorithm (DND)

The algorithm to develop has to solve the restrictions mentioned above: object competition, maximal consistency and calculation of probability functions, and possibly improve the performance. In order to solve the first two points, a solution is to assure the object distribution to the rules (according to the probabilities) and maximal execution.

In [116], V. Nguyen et al. introduced an algorithm performing non-deterministic, maximally parallel object distribution for transition P systems, and its hardware implementation. Assuming that it is possible to have more than one solution to the object distribution problem, several approaches are also analyzed:

- Indirect approaches: These approaches consider both solutions and non-solutions to the problem in the searching process. For example, the *Incremental Approach* constructs a solution starting from a non-solution.
- Direct approaches: These approaches only consider solutions in the exploration process. For example, the *Direct Non-deterministic Distribution algorithm (DND)* constructs an object distribution in one step, with a possible second step to fix the maximal parallelism. In order to do that, the algorithm executes two phases:
  - Forward phase: it is a loop iterating the rules of a region in a random order, choosing a random value for each and storing it for the next rules that can have intersections.

– Backward phase: it is a loop iterating the rules of such region in the previous random order, checking if there exists more applications to the rules, and assigning that number to each.

It can be considered that the forward phase implements a non-deterministic object distribution, and the backward phase makes the maximal parallelism, both selecting the rules individually (without using blocks). Therefore, the idea of the DND algorithm is suitable for solving object competition in PDP systems.

6.5.2 DNDP pseudocode

Next we show the pseudocode of the DNDP algorithm (Algorithm 6.5.1).

Algorithm 6.5.1 DNDP MAIN PROCEDURE

Input: A PDP system of degree (q, m) with $q \geq 1,\ m \geq 1$, taking T time units, $T \geq 1$.

```
1: $C_0 \leftarrow$ initial configuration of the system
2: for $t \leftarrow 0$ to $T-1$ do
3:     $C'_t \leftarrow C_t$
4:     INITIALIZATION                                  ▷ (Algorithm 6.5.2)
5:     FIRST SELECTION PHASE: consistency              ▷ (Algorithm 6.5.3)
6:     SECOND SELECTION PHASE: maximality              ▷ (Algorithm 6.5.4)
7:     EXECUTION                                       ▷ (Algorithm 6.5.5)
8:     $C_{t+1} \leftarrow C'_t$
9: end for
```

Similarly to the previous algorithm (section 6.4), the transitions of the P system are simulated in two phases, selection and execution, in order to synchronize the consumption and production of objects. However, selection is divided in two micro-phases, following the design of the DND algorithm explained in section 6.5.1. The first one calculates a multiset of *consistent* applicable rules. The second eventually increases the multiplicity of some of the rules in the previous multiset to assure maximal application, obtaining a multiset of *maximally consistent* applicable rules.

Let us now describe the pseudocode of the four main modules of the simulation algorithm.

First of all, in order to simplify the selection and execution phases, the *initialization* process (Algorithm 6.5.2) constructs two ordered set of rules, A_j and B_j, gathering only applicable rules from $\mathcal{R}_E$ and $\mathcal{R}_{\Pi_j}$ in environment e_j, and having a probability greater than 0. Finally, a new feature is provided in this initialization procedure, which is that the probabilities of rules are recalculated for each moment t.

Algorithm 6.5.2 DNDP INITIALIZATION

```
1: for $j \leftarrow 1$ to $m$ do
2:     $R_{E,j} \leftarrow$ ordered set of rules from $\mathcal{R}_E$ related with the environment $j$
3:     $A_j \leftarrow$ ordered set of rules from $R_{E,j}$ whose probability is $> 0$ at step $t$
4:     $LC_j \leftarrow$ ordered set of pairs $\langle label, charge \rangle$ for all the membranes from $C_t$
           contained in the environment $j$
5:     $B_j \leftarrow \emptyset$
6:     for all $\langle h, \alpha \rangle \in LC_j$ (following the considered order) do
7:         $B_j \leftarrow B_j \cup$ ordered set of rules from $R_{\Pi_j}$ whose probability is $> 0$ at
           step $t$ for the environment $j$
8:     end for
9: end for
```

In the *first selection* phase (Algorithm 6.5.3), a multiset of consistent applicable rules, denoted by R_j for each environment j, is calculated. First, a random order is applied to $A_j \cup B_j$, and stored in an ordered set D_j. Moreover, a copy of the configuration C_t, called C'_t, is created and updated each time that a rule is selected (removing the LHS from C'_t).

Algorithm 6.5.3 DNDP FIRST SELECTION PHASE: CONSISTENCY

```
1: for $j \leftarrow 1$ to $m$ do
2:     $R_j \leftarrow \emptyset$
3:     $D_j \leftarrow A_j \cup B_j$ with a *random order*
4:     for all $r \in D_j$ (following the considered order) do
5:         $M \leftarrow$ maximum number of times that $r$ is applicable to $C'_t$
6:         if $r$ is *consistent* with the rules in $R_j^1 \wedge M > 0$ then
7:             $N \leftarrow$ maximum number of times that $r$ is applicable to $C_t$
8:             $n \leftarrow \min\{M, F_b(N, p_{r,j}(t))\}$
9:             $C'_t \leftarrow C'_t - n \cdot LHS(r)$
10:            $R_j \leftarrow R_j \cup \{< r, n >\}$
11:        end if
12:    end for
13: end for
```

Then, a rule r is considered applicable if the following holds: it is consistent with the previously[1] selected rules in R_j, and the number of possible applica-

[1]according to the order in D_j

tions M in C'_t is greater than 0. If a rule r is applicable, a random number of applications n is calculated according to the probability function.This number is generated by using a binomial distribution.

On the one hand, since C'_t has been updated by the previously selected rules, the number n cannot exceed M to guarantee a correct object distribution. On the other hand, if the generated number n is 0, the corresponding rule is also added to the multiset R_j, giving a new chance to be selected in the next phase (maximality). Therefore, we will handle the "multiset" of rules R_j as a set of pairs $\langle x, y\rangle$ where $x \in D_j$ and y is the number of times that x is going to be applied (eventually $y = 0$). We will denote $R_j^0 = \{r \in D_j : \langle x, 0\rangle \in R_j\}$ and $R_j^1 = \{r \in D_j : \langle x, n\rangle \in R_j, n > 0\}$. Only rules from R_j^1 are considered for the consistency condition, since rules from R_j^0 has not be applied in the first selection phase.

In the *second selection* phase (Algorithm 6.5.4), the consistent applicable rules are checked again in order to achieve maximality. Only consistent rules are considered, and taken from R_j. If one rule $r \in R_j$ has a number of applications M greater than 0 in C'_t, then M will be added to the multiplicity of the rule. In order to fairly distribute the objects among the rules, they are iterated in order with respect to the probabilities. Moreover, one rule from the multiset R_j^0 can be checked, so it is possible that another rule from R_j^1, inconsistent to this one, have been previously selected. In this case, the consistent condition has to be tested again.

Algorithm 6.5.4 DNDP SECOND SELECTION PHASE: MAXIMALITY

```
1: for $j \leftarrow 1$ to $m$ do
2:     $R_j \leftarrow R_j$ with an order by the rule probabilities, from highest to lowest
3:     for all $< r, n > \in R_j$ (following the selected order) do
4:         if $n > 0 \vee$ ($r$ is consistent with the rules in $R_j^1$) then
5:             $M \leftarrow$ maximum number of times that $r$ is applicable to $C'_t$
6:             if $M > 0$ then
7:                 $R_j \leftarrow R_j \cup \{< r, M >\}$
8:                 $C'_t \leftarrow C'_t - M \cdot LHS(r)$
9:             end if
10:        end if
11:    end for
12: end for
```

Finally, execution phase (Algorithm 6.5.5) is similar to the binomial block based algorithm. It will iterate all the rules in every R_j^1 (maximal applicable consistent rules from environment j), and it will add the right-hand sides of them to the configuration C'_t. At the end of the process, C'_t is actually the next configuration: the left-hand sides of rules have been removed in the first and second selection phases, and the right-hand sides are added in the execution stage.

Algorithm 6.5.5 DNDP EXECUTION

```
1: for all $< r, n >\in R_j$, $n > 0$ do
2:     $C'_t \leftarrow C'_t + n \cdot RHS(r)$
3:     Update the electrical charges of $C'_t$ according to $RHS(r)$
4: end for
```

The DNDP simulation algorithm aims to capture the semantic of a PDP system. It would therefore be interesting to justify that the execution of this algorithm to a system configuration simulates a transition step, in the sense that a maximal consistent multiset of rules is applied to that configuration. In this context, the application of the probability functions are not discussed here, since it cannot be the subject of a formal treatment.

Let us recall that the algorithm consists of a main loop, such that four modules are executed in each loop step: (a) *initialization*; (b) *first selection phase*; (c) *second selection phase*; and (d) *execution of rules.*

In the *initialization* process two ordered set of rules, A_j and B_j, gathering only rules from $\mathcal{R}_E$ and $\mathcal{R}_{\Pi_j}$ applicable in environment e_j, and having a probability greater than 0.

Let us start by analyzing the first selection phase. Its goal is to select a multiset of applicable rules for C_t, trying to capture the stochasticity of the system. The module receives as input:

- A number t ($0 \leq t \leq T - 1$), that indicates the step of the computation that is being simulated (actually, step t of the main loop of DNDP algorithm refers to the $(t + 1)$-th step of the P system computation).
- A number j ($1 \leq j \leq m$), representing which environment is being considered.
- C_t, the configuration at time t of the simulated PDP system of degree (q, m).
- The set A_j of all rules from $\mathcal{R}_E$ applicable on environment e_j and having a probability at time t strictly greater than 0.
- The set B_j of all rules $r \in \mathcal{R}_{\Pi_j}$ applicable in environment e_j such that their probabilities are strictly greater than 0.

The output generated by this module is a multiset of rules R_j and a configuration C'_t.

In [101] a proof of the following theorems formalizing the concept of correctness is given.

Theorem 6.1. *The multiset of rules R_j is applicable to C_t, and the result of removing the objects consumed by those rules is C'_t.*

Theorem 6.2. *There exists a maximally consistent multiset of applicable rules for C_t that can be obtained from R_j. That is, any rule that does not appear in R_j, cannot be consistently applied to C'_t.*

The module of the second phase of rules selection receives as input two numbers t and j ($0 \leq t \leq T-1$ and $1 \leq j \leq m$), like in the previous module, and it also receives the following:

- A multiset of rules R_j, obtained as output of the previous algorithmic module, with an order given by the probabilities of the rules at time t, from highest to lowest.
- An intermediate configuration C'_t, obtained from configuration C_t by removing all objects consumed by the application of the multiset R_j.

The output generated by this module is a multiset of rules R_j and a configuration C'_t.

The following theorems formalizes the concept of correctness, and the proof is also given in [101].

Theorem 6.3. *The multiset of rules R_j, obtained as output of the module, is a maximally consistent multiset of applicable rules to the configuration C_t.*

Theorem 6.4. *The configuration C'_t, obtained as output of this module, is the result of removing from C_t the objects consumed by the application of the multiset of rules R_j.*

The module of execution of selected rules receives as input, for each j ($1 \leq j \leq m$), the configuration C'_t and the selected multiset of rules R_j. The output is C'_t which is the next configuration C_{t+1}.

6.5.3 A test example: DNDP vs BBB

In this section we will compare the two algorithms, BBB and DNDP, towards a simple test example. The aim is to show the difference of them when working with systems having a specific behavior. The results here can be generated by hand, but they have been actually generated by using the software developed under *pLinguaCore* (see Chapter 7).

6.5.3.1 Test P systems

We have created a family of simple P systems with no biological meaning. An example of using the DNDP algorithm for simulating a more complex model of a theoretical ecosystem can be seen in [52].

These test P systems are PDP systems of degree $(2, m)$, of the following form:

$$\Pi = (G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_r : \ r \in \mathcal{R}\}, \mathcal{M}_1, \mathcal{M}_2)$$

where:

- G is an empty graph
- $\Gamma = \{a, b, x, y, z, d\} \cup \{f_i, g_i : 1 \leq i \leq N\}$
- $\Sigma = \emptyset$
- $\mathcal{R}_E = \emptyset$
- $\mu = [[\ ]_2]_1$ is the membrane structure.
  - $\mathcal{M}_1 = a^{N_a} \cup \{f_i^{N_f}, g_i^{N_g} : 1 \leq i \leq N\}$
  - $\mathcal{M}_2 = b^{N_b}$
- The rules $\mathcal{R}$ to apply are:

  $r_1 \equiv a\ f_i[b]_2^0 \xrightarrow{0.8} a\ f_i[b, x]_2^0, for 1 \leq i \leq N$

  $r_2 \equiv a\ f_i[b]_2^0 \xrightarrow{0.2} a\ f_i[b, y]_2^0, for 1 \leq i \leq N$

  $r_3 \equiv a\ g_i[b]_2^0 \xrightarrow{0.9} a\ g_i[b, z]_2^0, for 1 \leq i \leq N$

  $r_4 \equiv a\ g_i[b]_2^0 \xrightarrow{0.1} a\ g_i[b, d]_2^0, for 1 \leq i \leq N$

Let us remark that the skeleton of Π_j is composed by two membranes, but the number of environments is parametrized by m. On one hand, the parameter N configures the number of different objects f_i and g_i, what increases the number of rules in the system for performance testing (see Section 7.1.1). On the other hand, the parameters N_a, N_b, N_f, N_g configure the multiplicities of the corresponding objects in order to test the behavior of the algorithm.

6.5.3.2 Behavior evaluation

In order to experimentally validate the probabilistic behavior of the algorithm, we have configured three test P systems with the following parameters values:

- Test 1: $m = 1, N = 1, N_a = 1000, N_b = 1000, N_f = 100, N_g = 500$. The maximum numbers of applications for the rules r_1 and r_2 are 100 (limited by object f_1), and for r_3, r_4 are 500 (limited by object g_1).
- Test 2: $m = 1, N = 1, N_a = 100, N_b = 500, N_f = 1000, N_g = 1000$. The maximum numbers of applications for the rules r_1, r_2, r_3 and r_4 are 100 (limited by object a).
- Test 3: $m = 1, N = 1, N_a = 100, N_b = 500, N_f = 1000, N_g = 10$. The maximum numbers of applications for the rules r_1 and r_2 are 100 (limited by object a), and for r_3, r_4 are 10 (limited by object g_1).

These P systems have been simulated using the implemented DNDP algorithm, using $T = 1$ (only one transition step) and 30 simulations to calculate rounded average data. A summary of results can be viewed in table 6.1.

In test 1, the average numbers of applications for rules r_1 and r_2 are 84 and 18 respectively (that is, they follow an approximate ratio of 80% and 20% of applications). For rules r_3 and r_4, the average approximates the ratio of 90% and 10%. Thus, in this test, the average applications of rules approximate the probabilities defined for each rule.

However, in test 2, the four rules compete for object a, which limits the number of applications to 100. Therefore, the DNDP algorithm distributes this object following the probabilities of the rules. For example, in simulation 1, r_1 is firstly selected (77 times), r_2 is secondly selected (18 times), and r_4 is thirdly selected (only 5 times, because there are only 5 objects a left). In simulation 2, the order of selection of rules is r_3, r_2 (until finishing objects a), and in simulation 3, the order is r_3, r_1. So, we can see that the order is randomly chosen, but the rules cannot be always applied a number of times according to

Test	Rule	Simulation 1	Simulation 2	Simulation 3	Average
1	r_1	65	83	91	84
	r_2	35	17	9	18
	r_3	422	467	443	447
	r_4	78	33	57	55
2	r_1	77	0	8	38
	r_2	18	23	0	12
	r_3	0	77	92	46
	r_4	5	0	0	5
3	r_1	72	74	75	76
	r_2	18	18	15	17
	r_3	8	8	1	6
	r_4	2	0	1	1

Table 6.1: Number of applications for each rule in three different simulations, and the rounded average for 30 simulations.

the probabilities. Nevertheless, if we look to the average number of applications, we can see that the approximated ratio for each rule ($40\%, 10\%, 45\%, 5\%$ for rules r_1, r_2, r_3, r_4) is the half of the defined probabilities, since the common object a is distributed among them.

Finally, in test 3, the competition for object a by rules r_3 and r_4 with r_1 and r_2 influences in decreasing the average number of applications for each one, since object a is distributed and g_1 limits this competition.

6.6 Direct distribution based on Consistent Blocks Algorithm (DCBA)

The algorithms mentioned above share a common drawback. This drawback involves the distortion of the way in which blocks and rules are selected. That is, instead of blocks and rules being selected according to its probabilities in a uniform manner, this selection process is biased towards those with the highest probabilities. This section introduces our most recent algorithm, known as Direct distribution based on Consistent Blocks Algorithm (*DCBA*). This algorithm is introduced to solve the aforementioned distortion, thus not biasing the selection process towards the most likely blocks and rules.

First we introduce new concepts required to define the algorithm. Then we provide the pseudocode of the algorithm, and finally we compare the DCBA and the DNDP algorithms by a very simple test example.

6.6.1 Definitions for blocks and consistency

The DCBA selection mechanism starts from the assumption that rules in $\mathcal{R}$ and $\mathcal{R}_E$ can be classified into blocks of rules having the same left-hand side, following the Definitions 6.2 and 6.4 given in Section 6.1.

Recall that, according to the semantics of our model, the sum of probabilities of all the rules belonging to the same block is always equal to 1; in particular, rules with probability equal to 1 form individual blocks. Note that rules that have exactly the same left-hand side (LHS) belongs to the same block, but rules with overlapping (but different) left-hand sides are classified into different blocks. The latter leads to object *competition*, what is a critical aspect to manage with the simulation algorithms.

Definition 6.7. *Given* (i, α, u, v) *where* $1 \leq i \leq q$, $\alpha \in EC$, $u, v \in \Gamma^*$, *the block* $B_{i,\alpha,u,v}$ *is consistent if and only if there exists* α' *such that, for each* $r \in B_{i,\alpha,u,v}$, $charge(RHS(r)) = \alpha'$.

In other words, a block of rules is consistent if and only if it determines a consistent set of rules. We will consider rule blocks holding this property. In fact, the design of the new algorithm will consider a slightly different concept of block introduced in Section 6.4. The main reason for that is the convenience of incorporating the consistent property inside the blocks. This will be achieved by including in the same definition the new electrical charge represented in the right-hand side of the rules.

Definition 6.8. *Given* i, α, α', u, v *where* $1 \leq i \leq q$, $\alpha, \alpha' \in EC$, $u, v \in \Gamma^*$, *the block associated with* $(i, \alpha, \alpha', u, v)$ *is the set:*

$$B_{i,\alpha,\alpha',u,v} = \{r \in \mathcal{R} : LHS(r) = (i, \alpha, u, v) \wedge charge(RHS(r)) = \alpha'\}$$

Note that the block $B_{i,\alpha,\alpha',u,v}$ determines a consistent set of rules. Then, the left-hand side of a block B, denoted by $LHS(B)$, is defined as the left-hand side of any rule in the block.

Definition 6.9. *We say that two blocks* $B_{i_1,\alpha_1,\alpha'_1,u_1,v_1}$ *and* $B_{i_2,\alpha_2,\alpha'_2,u_2,v_2}$ *are mutually consistent with each other, if and only if* $(i_1 = i_2 \wedge \alpha_1 = \alpha_2) \Rightarrow (\alpha'_1 = \alpha'_2)$.

That is, two rule blocks are mutually consistent if the union of them represents a consistent set of rules.

Definition 6.10. *A set of blocks $\mathcal{B} = \{B^1, B^2, \dots, B^s\}$ is self consistent (or mutually consistent) if and only if they are pairwaise mutually consistent, that is $\forall i, j$ (B^i and B^j are mutually consistent).*

In such a context, a set of blocks has an associated set of tuples (i, α, α'), that is, a relation between labels and electrical charges ($H \times EC$) in EC. Then, a set of blocks is mutually consistent if and only if the associated relationship $H \times EC$ in EC is functional.

6.6.2 DCBA pseudocode

The goal of the DCBA (Direct distribution based on Consistent Blocks Algorithm) [99, 98] is to perform a proportional distribution of objects among competing blocks (with overlapping LHS), determining in this way the number of times that each rule in $\bigcup_{j=1}^{m} \mathcal{R}_{\Pi_j} \cup \mathcal{R}_E$ is applied. *I.e.* the algorithm simulates the computational steps of a PDP systems. Algorithm 6.6.1 describes the main loop of the DCBA. It follows the same general scheme as its predecessors, *DNDP* and *BBB* [100] where the simulation of a computing step is structured in two stages: The first stage (selection), selects which rules are to be applied (and how many times) on each environment. The second stage (execution), implements the effects of applying the previously selected rules, yielding the next configuration of the PDP system. Note that, although every Π_j has the same set of rules $\mathcal{R}$, their probability functions may be different for each environment. See [99] for a more detailed explanation and examples of how to apply this algorithm.

As shown in Algorithm 6.6.1, the selection stage consists of three phases: Phase 1 distributes objects to the blocks in a certain proportional way, Phase 2 assures the *maximality* by checking the maximal number of applications of each block, and Phase 3 translates block applications to rule applications by calculating random numbers using the multinomial distribution.

The INITIALIZATION procedure (Algorithm 6.6.2) constructs a static distribution table $\mathcal{T}_j$ for each environment. Two variables, B^j_{sel} and R^j_{sel}, are also initialized, in order to store the selected multisets of blocks and rules, respectively.

Observation 6.5. *Each column label of the tables $\mathcal{T}_j$ contains the information of the corresponding block left-hand side.*

Observation 6.6. *Each row of the tables $\mathcal{T}_j$ contains the information related to the object competitions: for a given object, its row indicates which blocks are competing for it (those columns having non-null values).*

Algorithm 6.6.1 DCBA MAIN PROCEDURE

```
Require: A Population Dynamics P system of degree $(q, m)$, $T \geq 1$ (time units),
    and $A \geq 1$ (Accuracy). The initial configuration is called $C_0$.
1: INITIALIZATION                                        ▷ (Algorithm 6.6.2)
2: for $t \leftarrow 1$ to $T$ do
3:     Calculate probability functions $f_{r,j}(t)$ and $p(t)$.
4:     $C'_t \leftarrow C_{t-1}$
5:     SELECTION of rules:
           – PHASE 1: distribution                       ▷ (Algorithm 6.6.3)
           – PHASE 2: maximality                         ▷ (Algorithm 6.6.4)
           – PHASE 3: probabilities                      ▷ (Algorithm 6.6.5)
6:     EXECUTION of rules.                               ▷ (Algorithm 6.6.6)
7:     $C_t \leftarrow C'_t$
8: end for
```

Algorithm 6.6.2 INITIALIZATION

```
1: Construction of the static distribution table $\mathcal{T}$:
       • Column labels: consistent blocks $B_{i,\alpha,\alpha',u,v}$ of rules from $\mathcal{R}$.
       • Row labels: pairs $(x, i)$, for all objects $x \in \Gamma$, and $0 \leq i \leq q$.
       • For each row, for each cell of the row: place $\frac{1}{k}$ if the object in the row label
         appears in its associated compartment with multiplicity $k$ in the LHS of
         the block of the column label.
2: for $j = 1$ to $m$ do                 ▷ (Construct the expanded static tables $\mathcal{T}_j$)
3:     $\mathcal{T}_j \leftarrow \mathcal{T}$.                   ▷ (Initialize the table with the original $\mathcal{T}$)
4:     For each rule block $B_{e_j,x}$ from $\mathcal{R}_E$, add a column labeled by $B_{e_j,x}$ to $\mathcal{T}_j$;
           place the value 1 at row $(x, 0)$ for that column.
5:     Initialize the multisets $B^j_{sel} \leftarrow \emptyset$ and $R^j_{sel} \leftarrow \emptyset$
6: end for
```

Algorithm 6.6.3 SELECTION PHASE 1: DISTRIBUTION

```
1: for $j = 1$ to $m$ do                                      ▷ (For each environment $e_j$)
2:     Apply filters to table $\mathcal{T}_j$, using $C'_t$ and obtaining $\mathcal{T}_j^t$, as follows:
         a. $\mathcal{T}_j^t \leftarrow \mathcal{T}_j$
         b. FILTER 1 $(\mathcal{T}_j^t, C'_t)$.
         c. FILTER 2 $(\mathcal{T}_j^t, C'_t)$.
         d. Check mutual consistency for the blocks remaining in $\mathcal{T}_j^t$. If there is at
            least one inconsistency then report the information about the error,
            and optionally halt the execution (in case of not activating step 3).
         e. FILTER 3 $(\mathcal{T}_j^t, C'_t)$.
3:     (OPTIONAL) Generate a set $S_j^t$ of sub-tables from $\mathcal{T}_j^t$, formed by sets of
           mutually consistent blocks, in a maximal way in $\mathcal{T}_j^t$ (by the inclusion
           relationship). Replace $\mathcal{T}_j^t$ with a randomly selected table from $S_j^t$.
4:     $a \leftarrow 1$
5:     repeat
6:         for all rows $X$ in $\mathcal{T}_j^t$ do
7:             $RowSum_{X,t,j} \leftarrow$ total sum of the non-null values in the row $X$.
8:         end for
9:         $\mathcal{TV}_j^t \leftarrow \mathcal{T}_j^t$                ▷ (A temporal copy of the dynamic table)
10:        for all non-null positions $(X, Y)$ in $\mathcal{T}_j^t$ do
11:            $mult_{X,t,j} \leftarrow$ multiplicity in $C'_t$ at $e_j$ of the object at row $X$.
12:            $\mathcal{TV}_j^t(X, Y) \leftarrow \lfloor mult_{X,t,j} \cdot \frac{(\mathcal{T}_j^t(X,Y))^2}{RowSum_{X,t,j}} \rfloor$
13:        end for
14:        for all not filtered column, labeled by block $B$, in $\mathcal{T}_j^t$ do
15:            $N_B^a \leftarrow \min_{X \in rows(\mathcal{T}_j^t)}(\mathcal{TV}_j^t(X, B))$   ▷ (The minimum of the column)
16:            $B_{sel}^j \leftarrow B_{sel}^j + \{B^{N_B^a}\}$         ▷ (Accumulate the value to the total)
17:            $C'_t \leftarrow C'_t - LHS(B) \cdot N_B^a$            ▷ (Delete the LHS of the block.)
18:        end for
19:        FILTER 2 $(\mathcal{T}_j^t, C'_t)$
20:        FILTER 3 $(\mathcal{T}_j^t, C'_t)$
21:        $a \leftarrow a + 1$
22:    until $(a > A) \vee$ (all the selected minimums at step 15 are $0$)
23: end for
```

The distribution of objects among the blocks with overlapping LHS (competing blocks) is performed in selection Phase 1 (Algorithm 6.6.3). The expanded static tables $\mathcal{T}_j$ are used for this purpose in each environment, together with three different filter procedures. FILTER 1 discards the columns of the table corresponding to non-applicable blocks due to mismatch charges in the

LHS and in the configuration C'_t. Then, FILTER 2 discards the columns with objects in the LHS not appearing in C'_t. Finally, in order to save space in the table, FILTER 3 discards empty rows. These three filters are applied at the beginning of Phase 1, and the result is a *dynamic table* $\mathcal{T}_j^t$ (for the environment j and time step t).

The semantics of the modeling framework requires a set of mutually consistent blocks before distributing objects to the blocks. For this reason, after applying FILTERS 1 and 2, the mutually consistency is checked. Note that this checking can be easily implemented by a loop over the blocks. If it fails, meaning that an inconsistency was encountered, the simulation process is halted, providing a warning message to the user. Nevertheless, it could be interesting to find a way to continue the execution by non-deterministically constructing a subset of mutually consistent blocks. Since this method can be exponentially expensive in time, it is optional for the user whether to activate it or not.

Once the columns of the *dynamic table* $\mathcal{T}_j^t$ represent a set of mutually consistent blocks, the distribution process starts. This is carried out by creating a temporal copy of $\mathcal{T}_j^t$, called $\mathcal{TV}_j^t$, which stores the following products:

- The normalized value with respect to the row: this is a way to *proportionally* distribute the corresponding object along the blocks. Since it depends on the multiplicities in the LHS of the blocks, in fact, the blocks requiring more copies of the same object are penalized in the distribution. This is inspired in the amount of energy required to gather individuals from the same species.

- The value in the dynamic table (i.e. $\frac{1}{k}$): this indicates the number of possible applications of the block with the corresponding object.

- The multiplicity of the object in the configuration C'_t: this performs the distribution of the number of copies of the object along the blocks.

After the object distribution process, the number of applications for each block is calculated by selecting the minimum value in each column. This number is then used for consuming the LHS from the configuration. However, this application could be not maximal. The distribution process can eventually deliver objects to blocks that are restricted by other objects. As this situation may occur frequently, the distribution and the configuration update process is performed A times, where A is an input parameter referring to *accuracy*. The more the process is repeated, the more accurate the distribution becomes, while the performance of the simulation decreases. We have experimentally checked

Algorithm 6.6.4 SELECTION PHASE 2: MAXIMALITY

```
1: for $j = 1$ to $m$ do                                  ▷ (For each environment $e_j$)
2:     Set a random order to the blocks remaining in the last updated table $\mathcal{T}_j^t$.
3:     for all block $B$, following the previous random order do
4:         $N_B \leftarrow$ number of possible applications of $B$ in $C'_t$.
5:         $B_{sel}^j \leftarrow B_{sel}^j + \{B^{N_B}\}$            ▷ (Accumulate the value to the total)
6:         $C'_t \leftarrow C'_t - LHS(B) \cdot N_B$   ▷ (Delete the LHS of block $B$, $N_B$ times.)
7:     end for
8: end for
```

that $A = 2$ gives the best accuracy/performance ratio. In order to efficiently repeat the loop for A, and also before going to the next phase (maximality), it is interesting to apply Filters 2 and 3 again.

After phase 1, it may be the case that some blocks are still applicable to the remaining objects. This may be caused by a low A value or by rounding artifacts in the distribution process. Due to the requirements of P systems semantics, a maximality phase is now applied (Algorithm 6.6.4). Following a random order, a maximal number of applications is calculated for each block still applicable.

Algorithm 6.6.5 SELECTION PHASE 3: PROBABILITY

```
1: for $j = 1$ to $m$ do                                  ▷ (For each environment $e_j$)
2:     for all block $B^{N_B} \in B_{sel}^j$ do
3:         Calculate $\{n_1, \dots, n_l\}$, a random multinomial $M(N_B, g_1, \dots, g_l)$ with
             respect to the probabilities of the rules $r_1, \dots, r_l$ within the block.
4:         for $k = 1$ to $l$ do
5:             $R_{sel}^j \leftarrow R_{sel}^j + \{r_k^{n_k}\}$.
6:         end for
7:     end for
8:     Delete the multiset of selected blocks $B_{sel}^j \leftarrow \emptyset$.      ▷ (Useful in next step)
9: end for
```

After the application of phases 1 and 2, a maximal multiset of selected (mutually consistent) blocks has been computed. The output of the selection stage has to be, however, a maximal multiset of selected rules. Hence, Phase 3 (Algorithm 6.6.5) passes from blocks to rules, by applying the corresponding probabilities (at the local level of blocks). The rules belonging to a block are selected according to a multinomial distribution $M(N, g_1, \dots, g_l)$, where N is the number of applications of the block, and $g_1, \dots, g_l$ are the probabilities associated with the rules $r_1, \dots, r_l$ within the block, respectively.

Algorithm 6.6.6 EXECUTION

```
1: for $j = 1$ to $m$ do                                        ▷ (For each environment $e_j$)
2:     for all rule $r^n \in R^j_{sel}$ do                      ▷ (Apply the RHS of selected rules)
3:         $C'_t \leftarrow C'_t + n \cdot RHS(r)$
4:         Update the electrical charges of $C'_t$ from $RHS(r)$.
5:     end for
6:     Delete the multiset of selected rules $R^j_{sel} \leftarrow \emptyset$.   ▷ (Useful for the next step)
7: end for
```

Finally, the execution stage (Algorithm 6.6.6) is applied. This stage consists on adding the RHS of the previously selected multiset of rules, as the objects present on the LHS of these rules have already been consumed. Moreover, the indicated membrane charge is set.

6.6.3 A test example: DCBA vs DNDP

Let us consider a test example, with no biological meaning, in order to show the different behavior of the algorithms. This test PDP system is of degree $(2,1)$, and of the following form:

$$\Pi_{test} = (G, \Gamma, \mu, \Sigma, \mathcal{R}, T, \{f_r : r \in \mathcal{R}\}, \mathcal{M}_e, \mathcal{M}_1, \mathcal{M}_2)$$

where:

- G is an empty graph because $\mathcal{R}_E = \emptyset$.
- $\Gamma = \{a, b, c, d, e, f, g, h\}$
- $\mu = [\ [\ ]_2\ ]_1$ is the membrane structure, and the corresponding initial multisets are:
  - $\mathcal{M}_e = b$ (in the environment)
  - $\mathcal{M}_1 = a^{60}$ (in membrane 1)
  - $\mathcal{M}_2 = a^{90}\ b^{72}\ c^{66}\ d^{30}$ (in membrane 2)
- $T = 1$, only one time step.
- The rules $\mathcal{R}$ to apply are:

  $r_{1.1} \equiv [\ a^4\ b^4\ c^2\ ]_2 \xrightarrow{0.7} e^2\ [\ ]_2$

  $r_{1.2} \equiv [\ a^4\ b^4\ c^2\ ]_2 \xrightarrow{0.2} [\ e^2\ ]_2$

$$r_{1.3} \equiv [\ a^4\ b^4\ c^2\ ]_2 \xrightarrow{0.1} [\ e\ f\ ]_2$$

$$r_2 \equiv [\ a^4\ d\ ]_2 \xrightarrow{1} f^2[\ ]_2$$

$$r_3 \equiv [\ b^5\ d^2\ ]_2 \xrightarrow{1} g^2[\ ]_2$$

$$r_4 \equiv b\ [\ a^7\ ]_1^- \xrightarrow{1} [\ h^{100}\ ]_1^-$$

$$r_5 \equiv a^3\ [\ ]_2 \xrightarrow{1} [\ e^3\ ]_2$$

$$r_6 \equiv a\ b\ [\ ]_2 \xrightarrow{1} [\ g^3\ ]_2^-$$

We can construct a set of six consistent rule blocks $B_{\Pi_{test}}$ (of the form $b_{h,\alpha,\alpha',u,v}$) from the set $\mathcal{R}$ of Π_{test} as follows:

- $b_1 \equiv b_{2,0,0,\emptyset,a^4b^4c^2} = \{r_{1.1}, r_{1.2}, r_{1.3}\}$
- $b_2 \equiv b_{2,0,0,\emptyset,a^4d} = \{r_2\}$
- $b_3 \equiv b_{2,0,0,\emptyset,b^5d^2} = \{r_3\}$
- $b_4 \equiv b_{1,-,-,b,a^7} = \{r_4\}$
- $b_5 \equiv b_{2,0,0,a^3,\emptyset} = \{r_5\}$
- $b_6 \equiv b_{2,0,-,ab,\emptyset} = \{r_6\}$

It is noteworthy that the set $B_{\Pi_{test}}$ is not mutually consistent. However, only the blocks b_1, b_2, b_3 and b_5 are applicable in the initial configuration, and they, in fact, conform a mutually consistent set of blocks. Block b_4 is not applicable since the charge of membrane 1 is neutral, and block b_6 cannot be applied because there are no b's in membrane 1.

Rules	Simulation 1	Simulation 2	Simulation 3	Simulation 4	Simulation 5
$r_{1.1}$	11	0	0	0	0
$r_{1.2}$	4	4	3	0	0
$r_{1.3}$	1	0	0	0	0
r_2	6	18	6	22	2
r_3	1	6	12	4	14
r_4	-	-	-	-	-
r_5	20	20	20	20	20
r_6	-	-	-	-	-

Table 6.2: Simulating Π_{test} using the DNDP algorithm

Table 6.2 shows five different runs for one time step of Π_{test} using the DNDP algorithm. The values refers to the number of applications for each rule, which is actually the output of the selection stage (and the input of the execution stage). Note that for simulation 1, the applications for $r_{1.1}$, $r_{1.2}$ and $r_{1.3}$ follows the multinomial distribution. The applications of these rules are reduced because they are competing with rules r_2 and r_3. However, this competition leads to situations where the applications of the block b_1 does not follow a multinomial distribution. It comes from the fact of using a random order over the rules, but not over the blocks. Rules having a probability equals to 1 are more restrictive on the competitions because they are applied in a maximal way in their turn. This is the reason because on simulations 4 and 5, none of the rules $r_{1.i}, 1 \leq i \leq 3$ are applied.

This behavior could create a distortion of the reality described in the simulated model. But it is usually appeased running several simulations and making a statistical study. Finally, rules not competing for objects are applied as is, in a maximal way. For example, rule r_5 is always applied 20 times because its probability is equal to 1.

Next, the test example is executed using the DCBA. The main results of the different phases of the process is also detailed.

In the initialization phase, the static table is created, containing all the consistent blocks. The static table of Π_{test} is showed in Table 6.3. As shown, the values inside the cells of the table represents the inverse $(1/k)$ of the multiplicity of the object (in the membrane, as specified in the row) inside the block indicated in the header of the column.

Objects	**Consistent Blocks**					
	$b_{2,0,0,\emptyset,a^4b^4c^2}$	$b_{2,0,0,\emptyset,a^4d}$	$b_{2,0,0,\emptyset,b^5d^2}$	$b_{1,-,-,b,a^7}$	$b_{2,0,0,a^3,\emptyset}$	$b_{2,0,-,ab,\emptyset}$
$< a$,2>	1/4	1/4	-	-	-	-
$< b$,2>	1/4	-	1/5	-	-	-
$< c$,2>	1/2	-	-	-	-	-
$< d$,2>	-	1/1	1/2	-	-	-
$< a$,1>	-	-	-	1/7	1/3	1/1
$< b$,1>	-	-	-		-	1/1
$< b$,e>	-	-	-	1/1	-	-

Table 6.3: Static table

Once the static table has been initialized, the simulation main loop runs for the stated steps. Then, for each step of computation, the selection and execution of rules runs, as illustrated in the following paragraphs.

The selection starts with the distribution phase. The needed filters are performed, causing some objects and blocks to be discarded, as they need not present charges and/or objects. Then the corresponding calculus take place, getting the minimum number of applications of each way. The result of the selection phase 1 of the step 1 is showed in Table 6.4. The sum of the previously obtained values is showed in the last column. Then, the possible number of applications of a block is calculated for each object, considering its multiplicity in the current configuration and the block, and the relation with the sum of the row. This relation somehow captures the proportion of objects to be initially assigned to each block. Then, the minimum number of each block (given by the column) is calculated.

Objects	**Consistent Blocks**				**Sum**
	$b_{2,0,0,\emptyset,a^4b^4c^2}$	$b_{2,0,0,\emptyset,a^4d}$	$b_{2,0,0,\emptyset,b^5d^2}$	$b_{2,0,0,a^3,\emptyset}$	
$< a$,2> * 90	0.25 | 11	0.25 | 11	-	-	0.5
$< b$,2> * 72	0.25 | 10	-	0.2 | 6	-	0.45
$< c$,2> * 66	0.5 | 33	-	-	-	0.5
$< d$,2> * 30	-	1.0 | 20	0.5 | 5	-	1.5
$< a$,1> * 60	-	-	-	0.33 | 20	0.33
Applications	10	11	5	20	

Table 6.4: Selection Phase 1 - Distribution

The next phase, maximality, starts from the remaining objects, selecting new applications of the blocks in a maximal way. The result of this phase is showed in Table 6.5. This table presents the remaining objects (the ones not assigned in phase 1) and the possible blocks to be selected. The blocks are chosen in a random way, as shown in Algorithm 6.6.4, and the possible applications of the block are calculated. This process guarantees a maximal set of blocks to be selected, with a maximal number of applications of each block. The last row, applications, shows that the block $b_{2,0,0,\emptyset,\{a^4,b^4,c^2\}}$ is applying 1 time, additional to the number of applications calculated in the distribution phase.

Then the phase 3, probability, take place. For each block selected in the previous phases, its number of applications is divided among the rules being part of the block, according to their probabilities. As a result, the number of applications of each rule is obtained, as showed in Table 6.6.

Objects	**Consistent Blocks**		
	$b_{2,0,0,\emptyset,a^4b^4c^2}$	$b_{2,0,0,\emptyset,a^4d}$	$b_{2,0,0,\emptyset,b^5d^2}$
$< a$,2> * 6	-	-	-
$< b$,2> * 7	-	-	-
$< c$,2> * 46	-	-	-
$< d$,2> * 9	-	-	-
Applications	1	-	-

Table 6.5: Selection Phase 2 - Maximality

Rules	Simulation 1	Simulation 2	Simulation 3	Simulation 4	Simulation 5
$r_{1.1}$	7	10	7	6	7
$r_{1.2}$	3	0	4	1	2
$r_{1.3}$	1	1	5	3	1
r_2	11	11	11	12	12
r_3	5	5	5	6	6
r_4	-	-	-	-	-
r_5	20	20	20	20	20
r_6	-	-	-	-	-

Table 6.6: Simulating Π_{test} using the DCBA algorithm

It is noteworthy that the selection of rules belonging to block 1 $\{r_{1.i}, 1 \leq i \leq 3\}$, in Table 6.6, always follows a multinomial distribution with respect to the 3 probabilities. This solves the drawback we showed on Table 6.2. Moreover, it can be seen that the maximality sometimes can give one more application to blocks 2 and 3, in spite of keeping the original 10 applications for block 1 from phase 1. In any case, the number of applications is proportionally distributed, avoiding the distortion of using a random order over the blocks (or rules), as made in the DNDP algorithm.

6.7 Validation

6.7.1 Improved model for the scavenger bird ecosystem

In this section, a novel PDP systems based model for an ecosystem related to the Bearded Vulture in the Pyrenees (NE Spain) is presented. This model is an improved model from the one provided in [35]. The Bearded Vulture (*Gypaetus barbatus*) is an endangered species in Europe that feeds almost exclusively on bone remains of wild and domestic ungulates. In this model, the evolution of six species is studied: the Bearded Vulture and five subfamilies of domestic and wild ungulates upon which the vulture feeds.

The model consists of a PDP system of degree $(2,1)$,

$$\Pi = (G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_{r,1} :\ r \in \mathcal{R}\}, \mathcal{M}_1, \mathcal{M}_2)$$

where:

- $G = (V, S)$ with $V = \{e_1\}$ and $S = \emptyset$.
- In the alphabet Γ, we represent the seven species of the ecosystem (index i is associated with the species and index j is associated with their age, and the symbols X, Y and Z represent the same animal but in different "states"); it also contains the auxiliary symbol B, which represents 0.5 kg of bones, and C, which allows a change in the polarization of the membrane labeled by 2 at a specific stage.

  $$\Gamma = \{X_{i,j}, Y_{i,j}, Z_{i,j} : 1 \le i \le 7, 0 \le j \le k_{i,4}\} \cup \{B, C\}$$

  The species are the following:

  - Bearded Vulture ($i = 1$)
  - Pyrenean Chamois ($i = 2$)
  - Female Red Deer ($i = 3$)
  - Male Red Deer ($i = 4$)
  - Fallow Deer ($i = 5$)
  - Roe Deer ($i = 6$)
  - Sheep ($i = 7$)

  Note that although the male red deer and female red deer are the same species, we consider them as different species. This is because mortality of male deer is different from the female deer by reason of hunting.

- $\Sigma = \emptyset$.
- Each year in the real ecosystem is simulated by 3 computational steps, so $T = 3 \cdot Years$, where $Years$ is the number of years to simulate.
- $\mathcal{R}_E = \emptyset$.

- $\mu = [\ [\ ]_2\ ]_1$ is the membrane structure and the corresponding initial multisets are:

  - $\mathcal{M}_1 = \{\ X_{i,j}^{q_{i,j}} : 1 \leq i \leq 7, 0 \leq j \leq k_{i,4}\}$

  - $\mathcal{M}_2 = \{\ C, B^{\alpha}\}$ where $\alpha = \lceil \sum_{j=1}^{k_{1,4}} q_{1,j} \cdot 1.10 \cdot 682 \rceil$

    Value α represents an external contribution of food which is added during the first year of study so that the Bearded Vulture survives. In the formula, $q_{1,j}$ represents the number of bearded vultures that are j years old, the goal of the constant factor 1.10 is to guarantee enough food for 10% population growth. At present, the population growth is estimated an average 4%, but this value can reach higher values. Thus, to avoid problems related with the underestimation of this value the first year we have overestimated the population growth at 10%. The constant value 682 represents the amount of food needed per year for a Bearded Vulture pair to survive.

- The set $\mathcal{R}$ is defined as follows:

  - Reproduction rules for ungulates

    Adult males

    $$r_{0,i,j} \equiv [X_{i,j}]_1 \xrightarrow{1-k_{i,13}} [Y_{i,j}]_1 : k_{i,2} \leq j \leq k_{i,4}, 2 \leq i \leq 7$$

    Adult females that reproduce

    $$r_{1,i,j} \equiv [X_{i,j}]_1 \xrightarrow{k_{i,5}k_{i,13}} [Y_{i,j}, Y_{i,0}]_1 : k_{i,2} \leq j < k_{i,3}, 2 \leq i \leq 7, i \neq 3$$

    Red Deer females produce 50% of female and 50% of male springs

    $$r_{2,j} \equiv [X_{3,j}]_1 \xrightarrow{k_{3,5}k_{3,13}0.5} [Y_{3,j}Y_{3,0}]_1 : k_{3,2} \leq j < k_{3,3}$$

    $$r_{3,j} \equiv [X_{3,j}]_1 \xrightarrow{k_{3,5}k_{3,13}0.5} [Y_{3,j}Y_{4,0}]_1 : k_{3,2} \leq j < k_{3,3}$$

    Fertile adult females that do not reproduce

    $$r_{4,i,j} \equiv [X_{i,j}]_1 \xrightarrow{(1-k_{i,5})k_{i,13}} [Y_{i,j}]_1 : k_{i,2} \leq j < k_{i,3}, 2 \leq i \leq 7$$

    Not fertile adult females

    $$r_{5,i,j} \equiv [X_{i,j}]_1 \xrightarrow{k_{i,13}} [Y_{i,j}]_1 : k_{i,3} \leq j \leq k_{i,4}, 2 \leq i \leq 7$$

    Young ungulates that do not reproduce

$$r_{6,i,j} \equiv [X_{i,j}]_1 \xrightarrow{1} [Y_{i,j}]_1 : 0 \leq j < k_{i,2}, 2 \leq i \leq 7$$

– Growth rules for the Bearded Vulture

$$r_{7,j} \equiv [X_{1,j}]_1 \xrightarrow{k_{1,6}+k_{1,10}} [Y_{1,k_{1,2}-1} Y_{1,j}]_1 : k_{1,2} \leq j < k_{1,4}$$

$$r_{8,j} \equiv [X_{1,j}]_1 \xrightarrow{1-k_{1,6}-k_{1,10}} [Y_{1,j}]_1 : k_{1,2} \leq j < k_{1,4}$$

$$r_9 \equiv [X_{1,k_{1,4}}]_1 \xrightarrow{k_{1,6}} [Y_{1,k_{1,2}-1} Y_{1,k_{1,4}}]_1$$

$$r_{10} \equiv [X_{1,k_{1,4}}]_1 \xrightarrow{1-k_{1,6}} [Y_{1,k_{1,4}}]_1$$

– Mortality rules for ungulates

Young ungulates which survive

$$r_{11,i,j} \equiv Y_{i,j}[\,]_2 \xrightarrow{1-k_{i,7}-k_{i,8}} [Z_{i,j}]_2 : 0 \leq j < k_{i,1}, 2 \leq i \leq 7$$

Young ungulates which die

$$r_{12,i,j} \equiv Y_{i,j}[\,]_2 \xrightarrow{k_{i,8}} [B^{k_{i,11}}]_2 : 0 \leq j < k_{i,1}, 2 \leq i \leq 7$$

Young ungulates which are retired from the ecosystem

$$r_{13,i,j} \equiv Y_{i,j}[\,]_2 \xrightarrow{k_{i,7}} [\,]_2 : 0 \leq j < k_{i,1}, 2 \leq i \leq 7$$

Adult ungulates that do not reach the average life expectancy

Those which survive

$$r_{14,i,j} \equiv Y_{i,j}[\,]_2 \xrightarrow{1-k_{i,10}} [Z_{i,j}]_2 : k_{i,1} \leq j < k_{i,4}, 2 \leq i \leq 7$$

Those which die

$$r_{15,i,j} \equiv Y_{i,j}[\,]_2 \xrightarrow{k_{i,10}} [B^{k_{i,12}}]_2 : k_{i,1} \leq j < k_{i,4}, 2 \leq i \leq 7$$

Ungulates that reach the average life expectancy

Those which die in the ecosystem

$$r_{16,i} \equiv Y_{i,k_{i,4}}[\,]_2 \xrightarrow{k_{i,9}+(1-k_{i,9})k_{i,10}} [B^{k_{i,12}}]_2 : 2 \leq i \leq 7$$

Those which die and are retired from the ecosystem

$$r_{17,i} \equiv Y_{i,k_{i,4}}[\,]_2 \xrightarrow{(1-k_{i,9})(1-k_{i,10})} [\,]_2 : 2 \leq i \leq 7$$

– Mortality rules for the Bearded Vulture

$$r_{18,j} \equiv Y_{1,j}[\,]_2 \xrightarrow{1-k_{1,10}} [Z_{1,j}]_2 : k_{1,2} \leq j < k_{1,4}$$

$$r_{19,j} \equiv Y_{1,j}[\,]_2 \xrightarrow{k_{1,10}} [\,]_2 : k_{1,2} \leq j < k_{1,4}$$

$$r_{20} \equiv Y_{1,k_{1,4}}[\,]_2 \xrightarrow{1} [Z_{1,k_{1,2}-1}]_2$$

$$r_{21} \equiv Y_{1,k_{1,2}-1}[\,]_2 \xrightarrow{1} [Z_{1,k_{1,2}-1}]_2$$

– Feeding rules

$$r_{22,i,j} \equiv [Z_{i,j} B^{k_{i,14}}]_2 \xrightarrow{1} X_{i,j+1}[\,]_2^+ : 0 \leq j \leq k_{i,4}, 1 \leq i \leq 7$$

– Balance rules

Elimination of remaining bones

$$r_{23} \equiv [B]_2^+ \xrightarrow{1} [\,]_2$$

Adult animals that die because they have not enough food

$$r_{24,i,j} \equiv [Z_{i,j}]_2^+ \xrightarrow{1} [B^{k_{i,12}}]_2 : k_{i,1} \leq j \leq k_{i,4}, 1 \leq i \leq 7$$

Young animals that die because the have not enough food

$$r_{25,i,j} \equiv [Z_{i,j}]_2^+ \xrightarrow{1} [B^{k_{i,11}}]_2 : 0 \leq j < k_{i,1}, 1 \leq i \leq 7$$

Change the polarization

$$r_{26} \equiv [C]_2^+ \xrightarrow{1} [C]_2$$

The constants associated with the rules have the following meaning:

- $k_{i,1}$: Age at which adult size is reached. This is the age at which the animal consumes food as an adult does, and at which, if the animal dies, the amount of biomass it leaves behind is similar to the total left by an adult. Moreover, at this age it will have surpassed the critical early phase during which the mortality rate is high.
- $k_{i,2}$: Age at which it begins to be fertile.
- $k_{i,3}$: Age at which it stops being fertile.
- $k_{i,4}$: Average life expectancy in the ecosystem.
- $k_{i,5}$: Fertility ratio (number of descendants by fertile females).
- $k_{i,6}$: Population growth (this quantity is expressed in terms of 1).
- $k_{i,7}$: Animals retired from the ecosystem in the first year, age $< k_{i,1}$ (this quantity is expressed in terms of 1).

- $k_{i,8}$: Natural mortality ratio in first years, age $< k_{i,1}$ (this quantity is expressed in terms of 1).
- $k_{i,9}$: 0 if the live animals are retired at age $k_{i,4}$, in other cases, the value is 1.
- $k_{i,10}$: Mortality ratio in adult animals, age $\geq k_{i,1}$ (this quantity is expressed in terms of 1).
- $k_{i,11}$: Amount of bones from young animals, age $< k_{i,1}$.
- $k_{i,12}$: Amount of bones from adult animals, age $\geq k_{i,1}$.
- $k_{i,13}$: Proportion of females in the population (this quantity is expressed in terms of 1).
- $k_{i,14}$: Amount of food necessary per year and breeding pair (1 unit is equal to 0.5 kg of bones).

In [35], some actual values for the constants associated with the rules can be found, as well as actual values for the initial populations $q_{i,j}$ for each species i with age j. There are two sets of initial populations values, one beginning on year 1994 and another one beginning on year 2008.

6.7.2 Simulation results

PLinguaCore is a software library for simulation that accepts an input written in P-Lingua [70] and provides simulators of the defined P systems. For each supported type of P system, there are one or more simulation algorithms implemented in pLinguaCore. It is a software framework, so it can be expanded with new simulation algorithms.

Thus, we have extended the pLinguaCore library to include the DCBA simulation algorithm for PDP systems. The current version of pLinguaCore is 3.0 and can be downloaded from [16].

In this section, we use the model of the Bearded Vulture described above to compare the simulation results produced by the pLinguaCore library using two different simulation algorithms: DNDP [100] and DCBA. We also compare the results of the implemented simulation algorithms with the results provided by the C++ *ad hoc* simulator and with the actual ecosystem data, both obtained from [35]. In [9] it can be found the P-Lingua file which defines the model and instructions to reproduce the comparisons.

We have set the initial population values with the actual ecosystem values for the year 1994. For each simulation algorithm we have made 100 simulations of 14 years, that is, 42 computational steps. The simulation workflow has been implemented on a Java program that runs over the pLinguaCore library (this Java program can be downloaded from [9]). For each simulated year (3 computational steps), the Java program counts the number of animals for each species i, that is, $X_i = \sum_{j=0}^{k_{i,4}} X_{i,j}$. After 100 simulations, the Java program calculates average values for each year and species and writes the output to a text file. Finally, we have used the GnuPlot software [12] to produce the population graphics.

The population graphics for each species and simulation algorithm are represented in Figures 6.1 to 6.7.

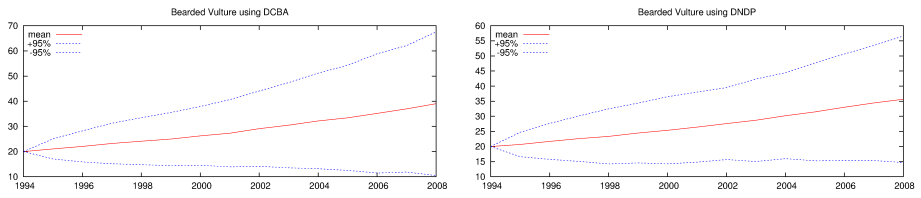


Figure 6.1: Evolution of the Bearded Vulture birds

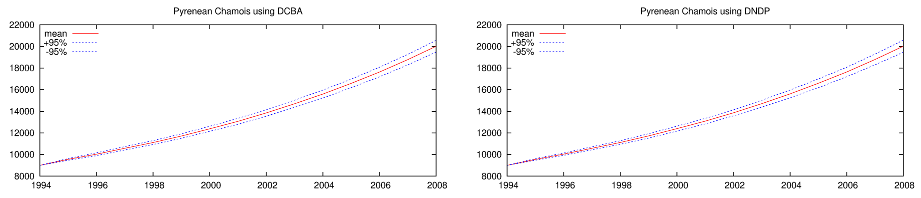


Figure 6.2: Evolution of the Pyrenean Chamois

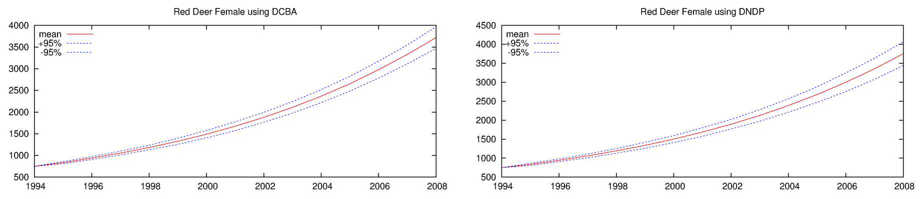


Figure 6.3: Evolution of the female Red Deer

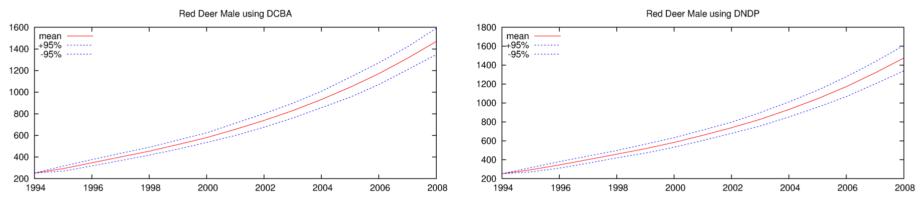


Figure 6.4: Evolution of the male Red Deer

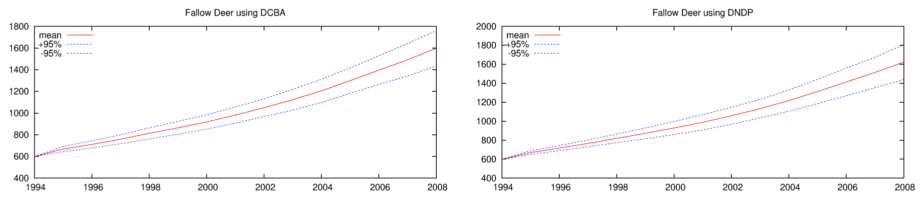


Figure 6.5: Evolution of the Fallow Deer

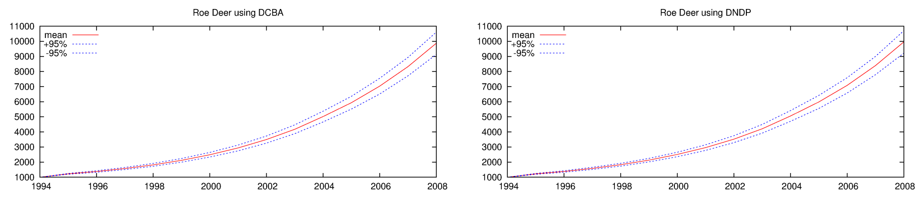


Figure 6.6: Evolution of the Roe Deer

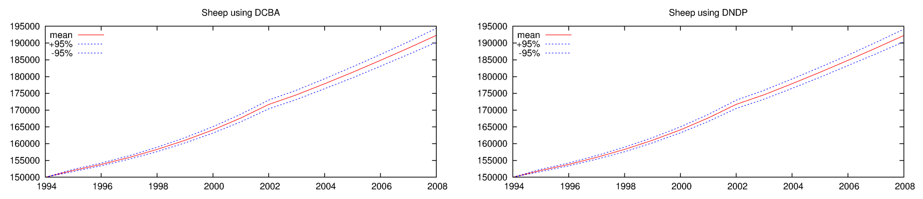


Figure 6.7: Evolution of the Sheep

The main difference between the DNDP and the DCBA algorithms is the way they distribute the objects between different rule blocks that compete for the same objects. In the model, the behavior of the ungulates are modeled by using rule blocks that do not compete for objects. So, the simulator provides similar results for both DCBA and DNDP algorithms. In the case of the

Bearded Vulture, there is a set of rules $r_{22,i,j}$ that compete for objects because $k_{1,14}$ is not 0 (the Bearded Vulture needs to feed on bones to survive). The $k_{i,14}$ constants are 0 for ungulates ($2 \leq i \leq 7$), because they do not need to feed on bones to survive. The initial amount of bones and the amount of bones generated during the simulation is enough to support the Bearded Vulture population regardless the way the simulation algorithm distributes the bones between vultures of different ages (rules $r_{22,1,j}$). Since there is a small initial population of bearded vultures (20 pairs), some differences can be noticed between the results from DCBA, DNDP, C++ simulator and the actual ecosystem data for the Bearded Vulture (39 bearded vultures with DCBA for year 2008, 36 with DNDP, 38 with the C++ simulator and 37 on the actual ecosystem).

In Figure 6.8 it is shown the comparison between the actual data for the year 2008 and the simulation results obtained by using the C++ *ad hoc* simulator, the DNDP algorithm and the DCBA algorithm implemented in pLinguaCore. In the case of the Pyrenean Chamois, there is a difference between the actual population data on the ecosystem (12000 animals) and the results provided by the other simulators (above 20000 animals), this is because the population of Pyrenean Chamois was regulated on year 2004 [35]. Taking this into account, one can notice that all the simulators behave in a similar way for the above model and they can reproduce the actual data after 14 simulated years. So, the DCBA algorithm is able to reproduce the semantics of PDP systems and it can be used to simulate the behavior of actual ecosystems by means of PDP systems.

6.8 Conclusions

In this chapter we have introduced the two simulation algorithms for PDP systems, called DNDP and DCBA. They are focused on the distortion caused by the previous algorithm, called BBB, when several rule blocks compete for same objects.

- *DNDP* (Direct Non-Deterministic distribution with Probabilities algorithm): it is the first adopted approach. Based on the idea of DND algorithm, it performs a non-deterministic distribution of objects along the rules, but considering the probabilities. The algorithm is split into two phases, selection and execution. This time, selection phase is divided into two micro-phases: phase selection 1 (consistency) and phase selection 2 (maximality). Together with an initialization phase, it has

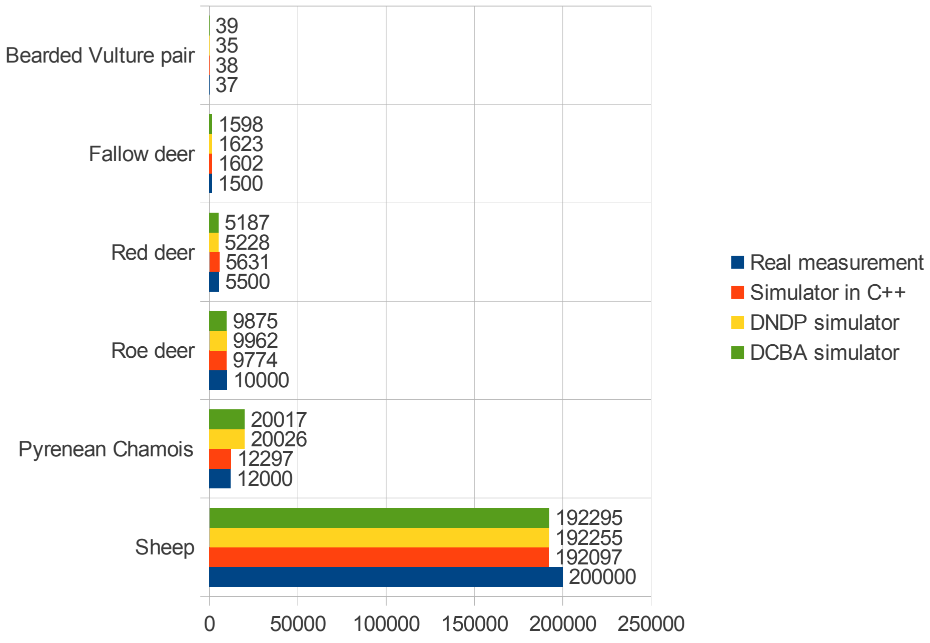


Figure 6.8: Data of the year 2008 from: real measurements of the ecosystem, original simulator in C++, simulator using DNDP and simulator using DCBA.

a total of four phases. The first selection phase calculates a multiset of consistent applicable rules. This is performed by looping the rules in a random order, and applying each one (if consistent with the already selected rules) using the binomial distribution according to the probabilities. The second selection phase eventually increases the multiplicity of some of the rules in the previous multiset to assure maximal application, obtaining a multiset of maximally consistent applicable rules. Again, there is a loop over the remaining rules, checking the maximality condition. Although the DNDP algorithm achieves better results than its predecessor (BBB), the behavior still produces some distortion in many situations (it is biased towards the rules with the highest probabilities).

- *DCBA* (Direct distribution based on Consistent Blocks Algorithm): this second approach is based on the idea of proportionally distributing the amount of objects along the rule blocks. A proportional calculus is made in such a way that rules requesting for more objects are penalized. However, this calculation can be adapted to the biological semantics to be captured by the model. Probabilities are applied locally to rule blocks. The simulation algorithm consists on two phases, selection and execution. But this time, selection is split into three micro-phases: phase 1 (distribution), phase 2 (maximality), and phase 3 (probabilities). Selection phase 1 uses a distribution table, where rows represent objects inside regions, and columns are rule blocks. A normalized distribution of the objects is performed over the rows. Phase 2 iterates the remaining rule blocks assuring maximality, and phase 3, once rule blocks have been selected, calculate multinomial distributions for each one (according to the selected number for it, and the probabilities of the corresponding rules). DCBA is able to reproduce the desired semantics for the model of PDP systems. However, its efficient implementation is a challenge (the distribution table can be very large).

These two algorithms have been validated towards a real ecosystem, showing that they can still support already existing PDP systems based models. Their new functionalities were tested against toy examples. New P systems models showing the commented distortions are going to be run with these algorithms when the new parallel simulators are available for pLinguaCore.

"We define the art of conjecture, or stochastic art, as the art of evaluating as exactly as possible the probabilities of things, so that in our judgments and actions we can always base ourselves on what has been found to be the best, the most appropriate, the most certain, the best advised; this is the only object of the wisdom of the philosopher and the prudence of the statesman."

Jacob Bernoulli

7 Parallel Simulation of PDP Systems

One of the main objectives of using PDP systems is to help the ecologists to adopt *a priori* management strategies for real ecosystems. This is accomplished by executing virtual experiments. Thus, the design of simulators and other related software tools becomes a critical point in the process of model validation, as well as for virtual experimentation. Indeed, after running virtual experiments, it is often required to add or remove some ingredients or characteristics from the initial model's blueprint. These modifications are quite simple to implement thanks to the modularity of these models.

The PDP systems modeling framework is supported by a software tool called MeCoSim [125]. It provides, among others, the following features: a graphical user interface (users be ecologists or model designers), definition of model and ecosystem's parameters, execution of simulations, creation of statistical data in form of tables, graphs, etc. The core of this application is *pLinguaCore* [70], a simulation software library for Membrane Computing. The models are defined by plain-text files using the P-Lingua specification language. The application loads that file, configures the corresponding parameters, calls *pLinguaCore* to execute, and collects the results of the simulation.

Many simulation algorithms are defined in *pLinguaCore* for different P system models. Specifically for PDP systems, the implemented simulation algorithms are BBB [100], DNDP [100, 101] and DCBA [99, 98]. For the DNDP simulation algorithm, there are two implementations: in sequential and in parallel. The parallel version uses Java threads, so the parallelism is actually a concurrent execution of threads managed by the JVM[1]. The DCBA has been implemented considering performance. The big effort is the compression of the static and dynamic tables by using a hash table.

In this chapter we will detail these implementations in *pLinguaCore* for both DNDP (Section 7.1) and DCBA (Section 7.2). Moreover, we will introduce a set of alternative implementations for the DCBA in C++ [88, 103] (Section 7.3), OpenMP [15, 103] (Section 7.4) and CUDA [5, 104] (Section 7.5). These implementations are placed in a stand-alone simulation tool, that will be connected with *pLinguaCore* in future.

All the simulators (C++, OpenMP and CUDA versions) are included in the framework named *ABCD-GPU*. It can be downloaded from the PMCGPU project website: `http://sourceforge.net/p/pmcgpu` [17]. More information about the simulator, how to install and use it, refer to Appendix A.

7.1 DNDP algorithm implementation in pLinguaCore

In this section we show an implementation to the DNDP algorithm following the imperative programming paradigm. We have included this implementation in the *pLinguaCore* library as a simulation algorithm for the model of PDP systems.

Let us suppose that the PDP system to be simulated is of degree (q, m), with $q, m \geq 1$, and using $T \geq 1$ time units of computation. Then, the algorithm must receive the following input parameters:

- The initial configuration of the system, C_0.
- The sets of rules $\mathcal{R}_E$ (communication rules) and $\mathcal{R}$ (evolution rules).
- The values q, m and T.
- The set of functions $\{f_{r,j} :\ r \in \mathcal{R}, 1 \leq j \leq m\}$.

The main routines of this DNDP implementation are shown in Algorithm 7.1.1. The key aspect of this implementation is the provided parallel solution. Considering that the left-hand sides of rules only affect to one environment, the selection of rules can be executed in parallel for each environment without

[1]Java Virtual Machine

compromising the concurrency when accessing to common objects. However, the selection and execution of rules has to be synchronized, so that the execution phase cannot start before finishing the selection in every environment.

Algorithm 7.1.1 Main DNDP procedure

```
Procedure DNDP ($C_0$, $\mathcal{R}_E$, $\mathcal{R}$, $q$, $m$, $T$, $K$, $\{f_{r,j} :\ r \in \mathcal{R}, 1 \leq j \leq m\}$)
 1: $\{L_{\mathcal{R}_E,j}, L_{\mathcal{R},h,\alpha} :\ 1 \leq j \leq m,\ 0 \leq h < q,\ \alpha \in \{-,0,+\}\}$ ← Initialization($\mathcal{R}_E$, $\mathcal{R}$, $q$, $m$)
 2: for $t \leftarrow 0$ to $T-1$ do
 3:     $C'_t \leftarrow C_t$
 4:     for $j \leftarrow 1$ to $m$ do
 5:         Throw new Thread Selection-execution($j$, $t$, $C'_t$, $C_t$, $L_{\mathcal{R}_E,j}$,
              $\{L_{\mathcal{R},h,\alpha}\ 0 \leq h < q, \alpha \in \{-,0,+\}\}$, $q$, $m$, $K$, $\{f_{r,j} :\ r \in \mathcal{R}\}$)
 6:     end for
 7:     Barrier-synchronization ▷ All threads wait until everyone reaches this point
 8:     $C_{t+1} \leftarrow C'_t$
 9:     print $C_{t+1}$
10: end for

Thread Selection-execution ($j$, $t$, $C'_t$, $C_t$, $L_{\mathcal{R}_E,j}$, $\{L_{\mathcal{R},h,\alpha} : 0 \leq h \leq q-1, \alpha \in \{-,0,+\}\}$, $q$, $m$, $K$, $\{f_{r,j} :\ r \in \mathcal{R}\}$)
 1: $D_j, max_{D_j}, B_j$ ← Initialize-selection-phase($j$, $t$, $C_t$, $L_{\mathcal{R}_E,j}$, $\{L_{\mathcal{R},h,\alpha}\ 0 \leq h < q, \alpha \in \{-,0,+\}\}$, $q$, $m$, $K$, $\{f_{r,j} :\ r \in \mathcal{R}\}$)
 2: $D_j, max_{D_j}, B_j$ ← Selection-first-phase($j$, $D_j$, $max_{D_j}$, $B_j$, $K$, $C'_t$, $C_t$)
 3: $D_j, max_{D_j}, B_j$ ← Selection-second-phase($j$, $D_j$, $max_{D_j}$, $B_j$, $C'_t$)
 4: Barrier-synchronization ▷ All threads wait until everyone reaches this point
 5: Execution($j$,$D_j$,$max_{D_j}$,$C'_t$)
```

Before executing the *Selection-first-phase* and *Selection-second-phase* functions, the algorithm needs to initialize some data structures. First, the initialization phase of the pseudocode is executed by the function *Initialization*, and the data structures for the rest of phases are initialized with rules from A_j and B_j (represented by $L_{\mathcal{R}_E,j}$ and $L_{\mathcal{R},h,\alpha}$) by the function *Initialize-selection-phase.*

```
Function Initialization ($\mathcal{R}_E$, $\mathcal{R}$, $q$, $m$)
 1: for $j \leftarrow 1$ to $m$ do
 2:     The set of rules of the form $(x)_{e_j} \xrightarrow{pr} (x_1)_{e_{j_1}}, \ldots, (x_s)_{e_{j_s}} \in \mathcal{R}_E$ is
            lexicographically sorted with respect of $x, x_1, \ldots, x_s$.
 3:     Let $L_{\mathcal{R}_E,j}$ be the list of rules with the previous order.
 4: end for
 5: for $h \leftarrow 0$ to $q-1$ do
 6:     for all $\alpha \in \{-,0,+\}$ do
 7:         The set of rules of the form $u[v]_h^{\alpha} \rightarrow u'[v']_h^{\alpha'} \in \mathcal{R}$ is lexicographically
              sorted with respect of $u, v, u', v'$.
 8:         Let $L_{\mathcal{R},h,\alpha}$ be the list of rules with the previous order.
 9:     end for
10: end for
11: return $\{L_{\mathcal{R}_E,j}, L_{\mathcal{R},h,\alpha}\ 0 \leq j \leq m-1, 0 \leq h \leq q-1, \alpha \in \{-,0,+\}\}$
```

Function Initialize-selection-phase (j, t, C_t, $L_{\mathcal{R}_E,j}$, $\{L_{\mathcal{R},h,\alpha}\ 0 \leq h \leq q-1, \alpha \in \{-,0,+\}\}$, q, m, K, $\{f_{r,j} :\ r \in \mathcal{R}\}$)

```
 1: $D_j \leftarrow$ New array of triples $<rule-id,\ float,\ integer>$, representing
    $<rule,\ probability,\ selection-number>$ respectively, of size:

       $$\sum_{h \leftarrow 0}^{q-1} Length(L_{\mathcal{R}_E,j}) + maximum\{Length(L_{\mathcal{R},h,\alpha}) : \alpha \in \{-,0,+\}\}$$

 2: $max_{D_j} \leftarrow 0$
 3: for $h \leftarrow 0$ to $q-1$ do
 4:     $\alpha \leftarrow$ charge of membrane $h$ in environment $j$, from $C_t$
 5:     for all $r \in L_{\mathcal{R},h,\alpha}$ do
 6:         $p \leftarrow f_{r,j}(t)$
 7:         if $p > 0$ then
 8:             $D_j[max_{D_j}] \leftarrow <r, p, 0>$
 9:             $max_{D_j} \leftarrow max_{D_j} + 1$
10:         end if
11:     end for
12:     $B_j[h] \leftarrow false$
13: end for
14: for all $r \in L_{\mathcal{R}_E,j}$ do
15:     $p \leftarrow p_r(t)$
16:     if $p > 0$ then
17:         $D_j[max_{D_j}] \leftarrow <r, p, 0>$
18:         $max_{D_j} \leftarrow max_{D_j} + 1$
19:     end if
20: end for
21: return $max_{D_j}$, $D_j$, $B_j$
```

First, the set of rules is processed. For each transition and for each environment, a list of applicable rules to the configuration is constructed, according only to the membrane charge in the left-hand side. When these lists are completed, they are used to fill an array (one per environment), called D_j, of triples $< rule, probability, number - applications >$. At this point, the probabilities of rules are calculated using the associated function, and stored in the triple with the corresponding rule. The construction of this array, assigning a random order, corresponds to the set of rules D_j defined in the pseudocode.

Furthermore, a boolean array B with one entry per membrane and environment, is used to check the consistency of rules in an efficient way. In the pseudocode, the consistency condition is assured by testing each rule with the rest of the previously selected. In this implementation, every time that a rule $u[v]_h^{\alpha} \rightarrow u'[v']_h^{\alpha'}$ is selected a number of times greater than 0, the value in B for the active membrane h is set to true ($B[h] \leftarrow true$). Moreover, the charge is changed in C'_t. Then, an inconsistent rule can be found by looking at array B, saying that another rule has already reserved the change of charge (stored in C'_t).

We show below the code for the function Selection-first-phase, and its associated auxiliary functions. Additionally, the following computable functions are considered for random number generation:

- $F_b(N,p)$ is a function returning a random natural number $0 \leq n \leq N$, according to the binomial distribution $B(N,p)$ (see Section 7.5.2 for further information), where $N \in \{0,\dots,2^{64}\}$ and $p \in \mathbb{R} \cap [0,1]$. Real numbers are encoded in floating point (float) with a precision of 32 bits.
- $F_u(N)$ is a function returning a random natural number $0 \leq n \leq N-1$, according to the uniform distribution $U(N)$ (see Section 7.5.2 for further information), where $N \in \{1,\dots,2^{64}\}$.

Function Selection-first-phase (j,D_j,max_{D_j},B_j,K,C'_t,C_t)

```
1: $max_k \leftarrow maxD_j$
2: while $max_k > 0$ do
3:     $i \leftarrow F_u(max_k)$
4:     $\langle r,p,n \rangle \leftarrow D_j[i]$
5:     if Is-consistent($r$,$j$,$B_j$,$C'_t$) then
6:         $N' \leftarrow$ Count-applications($r$,$j$,$C'_t$)
7:         if $N' > 0$ then
8:             if $p = 1$ then
9:                 $n_i \leftarrow N'$
10:            else
11:                $N \leftarrow$ Count-applications($r$,$j$,$C_t$)
12:                $n_i \leftarrow F_b(N,p)$
13:                if $n_i > N'$ then
```

```
14:                 $n_i \leftarrow N'$
15:             end if
16:         end if
17:         if $n_i > 0$ then
18:             Remove-left-hand-rule-objects($r$,$j$,$n_i$,$C'_t$)
19:             Update-charge($r$,$j$,$B$,$C'_t$)
20:             $n \leftarrow n + n_i$
21:         end if
22:         Swap($D_j$,$i$,$max_k - 1$)
23:         $max_k \leftarrow max_k - 1$
24:     else
25:         Swap($D_j$,$i$,$max_k - 1$)
26:         Swap($D_j$,$max_k - 1$,$max_{D_j} - 1$)
27:         $max_k \leftarrow max_k - 1$
28:         $max_{D_j} \leftarrow max_{D_j} - 1$
29:     end if
30: else
31:     Swap($D_j$,$i$,$max_k - 1$)
32:     Swap($D_j$,$max_k - 1$,$max_{D_j} - 1$)
33:     $max_k \leftarrow max_k - 1$
34:     $max_{D_j} \leftarrow max_{D_j} - 1$
35: end if
36: end while
37: return $D_j$, $max_{D_j}$, $B_j$
```

Function Is-consistent (r, j, B, C_t)

```
1: $check \leftarrow true$
2: if $r$ is of the form $u[v]_h^{\alpha} \rightarrow u'[v']_h^{\alpha'}$ then
3:     $\beta \leftarrow$ charge of membrane $h$ in $\Pi_j \in C_t$
4:     if $B[h] = true \wedge \alpha' \neq \beta$ then
5:         $check \leftarrow false$
6:     end if
7: end if
8: return $check$
```

Function Count-applications (r, j, C_t)

```
1: if $r$ is of the form $(x)_{e_j} \xrightarrow{p_r} (x_1)_{e_{j_1}}, \ldots, (x_s)_{e_{j_s}}$ then
2:     $n \leftarrow$ multiplicity of object $x$ in environment $e_j \in C_t$
3: else if $r$ is of the form $u[v]_h^{\alpha} \rightarrow u'[v']_h^{\alpha'}$ then
4:     $n \leftarrow$ minimum value such that $u^n$ appears in the multiset of the parent
       compartment of $h$ in $\Pi_j \in C_t$ and $v^n$ appears in the multiset of $h$ in $\Pi_j \in C_t$
5: end if
```

```
6: return n
```

Procedure Remove-left-hand-rule-objects (r, j, n, C_t)

```
1: if $r$ is of the form $(x)_{e_j} \xrightarrow{pr} (x_1)_{e_{j_1}}, \ldots, (x_s)_{e_{j_s}}$ then
2:     Remove the multiset $x^n$ from environment $e_j \in C_t$
3: else if $r$ is of the form $u[v]_h^{\alpha} \to u'[v']_h^{\alpha'}$ then
4:     Remove the multiset $u^n$ from the multiset of the parent membrane of $h$ in $\Pi_j \in C_t$
5:     Remove the multiset $v^n$ from the multiset of membrane $h$ in $\Pi_j \in C_t$
6: end if
```

Procedure Update-charge (r, j, C_t)

```
1: if $r$ is of the form $u[v]_h^{\alpha} \to u'[v']_h^{\alpha'}$ then
2:     Assign $\alpha'$ to the electrical charge of membrane $h$ in $\Pi_j \in C_t$
3: end if
```

Procedure Swap (D, i, j)

```
1: $aux \leftarrow D[i]$
2: $D[i] \leftarrow D[j]$
3: $D[j] \leftarrow aux$
```

The function Selection-first-phase works directly with the array D_j, so it is a loop using a shuffle iterator (an index that takes random values between 0 and max_k). Like in the pseudocode (Algorithm 7.1.1), after checking the consistency condition, the number of applications is calculated. If it is non-zero, the binomial distribution is applied, getting a number n. When the rule is processed, n is stored in the corresponding triple, which is changed by the last one (max_k) in order to avoid selecting it again. In the case that the rule is inconsistent or not applicable, it has to be discarded. That is, the rule is changed by the last one (max_k) as before, but also changed with the last selected rule (max_{D_j}), avoiding to be processed in the second phase.

At the end of the Selection-first-phase, D_j corresponds to the consistent applicable rules, both with zero and non-zero selected numbers (R_j). The function Selection-second-phase is similar to the one depicted in the pseudocode. It iterates the rules in D_j, according to the order given by probabilities, to increase the number of applications of each one by checking the consistency and maximality conditions.

Function Selection-second-phase (j,D_j,max_{D_j},B,C'_t)

```
1: Sort triples $\langle r, p, n \rangle$ from $D_j$, from 0 to $max_{D_j}$, in decrement order by $p$.
2: for $i \leftarrow 0$ to $max_{D_j}$ do
```

```
3:      $< r,p,n > \leftarrow D_j[i]$
4:      **if** $n > 0 \vee$ Is-consistent($r$,$j$,$B_j$,$C'_t$) **then**
5:          $N' \leftarrow$ Count-applications($r$,$j$,$C'_t$)
6:          **if** $N' > 0$ **then**
7:              **if** $n = 0$ **then**
8:                  Update-charge($r$,$j$,$B_j$,$C'_t$)
9:              **end if**
10:             $n \leftarrow n + N'$
11:             Remove-left-hand-rule-objects($r$,$j$,$N'$,$C'_t$)
12:         **end if**
13:     **end if**
14: **end for**
15: **return** $D_j$, $max_{D_j}$, $B_j$
```

The implementation of the simulation algorithm ends with the procedure Execution, corresponding with the execution phase, which adds the right-hand side of rules to C'_t.

Procedure Execution (j, D_j, max_{D_j}, C'_t)

```
1: **for** $i \leftarrow 0$ to $max_{D_j}$ **do**
2:     $< r,p,n > \leftarrow D_j[i]$
3:     **if** $n > 0$ **then**
4:         Add-left-hand-rule-objects($r$,$j$,$n$,$C'_t$)
5:     **end if**
6: **end for**
```

Procedure Add-left-hand-rule-objects (r, j, n, C'_t)

```
1: **if** $r$ is of the form $(x)_{e_j} \xrightarrow{p_r} (x_1)_{e_{j_1}}, \ldots, (x_s)_{e_{j_s}}$ **then**
2:     Add the multisets $x_1^n, \ldots, x_s^n$ to the environments $e_{j_1}, \ldots, e_{j_s} \in C_t$, resp.
3: **else if** $r$ is of the form $u[v]_h^\alpha \rightarrow u'[v']_h^{\alpha'}$ **then**
4:     Add the multiset $u'^n$ to the multiset of the parent membrane of $h$ in $\Pi_j \in C_t$
5:     Add the multiset $v'^n$ to the multiset of membrane $h$ in $\Pi_j \in C_t$
6: **end if**
```

7.1.1 Performance evaluation

In order to evaluate the performance of the algorithm, we have used the test P systems from Section 6.5.3. Four test PDP systems have been configured with the following values:

- $m = 2, 4, 8, 16$: we vary the number of environments in each test P system to increase the parallelism in the algorithm.

- $N = 1000$: the total number of rules in $\mathcal{R}$ is 4000.
- $N_a = 2^{32}, N_b = 2^{32}, N_f = 1, N_g = 1$: using this values, each rule can be executed at most once.

These four P systems have been simulated using *pLinguaCore* with the three following implementations written in Java: BBB (binomial in Figure 7.1), DNDP sequential and DNDP parallel algorithms. The simulations have been executed in a computer with an Intel core2 Quad Q9550 system running at 2.83GHz with 4GB of main memory, and using Ubuntu Linux Server 8.04 with the Java Virtual Machine 1.6.0.

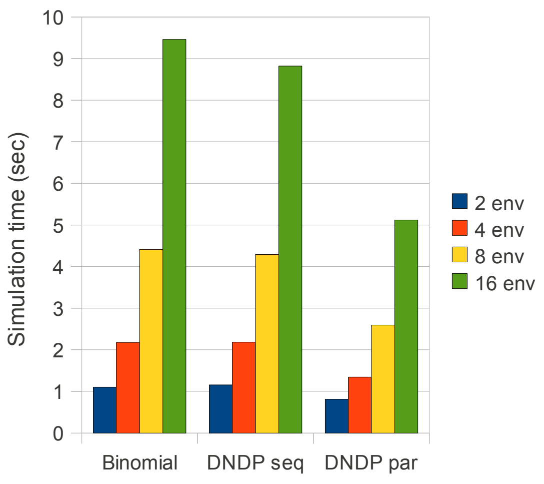


Figure 7.1: Simulation time of the three different implementations in a 4-core based computer

Figure 7.1 shows the simulation time for the three implementations simulating the example depicted above. It can be seen that the DNDP sequential code is a little bit faster than the BBB. As the complexity of the P system increments (in this example, the number of environments), the DNDP reaches better performance. In this experiment, we report up to 1,07x of speedup for the P system with 16 environments.

Furthermore, the parallel implementation of the DNDP algorithm has a better overall performance than the others. Using a 4-core based computer, the Java Virtual Machine distributes the threads (assigned to each environment of the system) among them. In this way, for the P system with 16 environments, we report a 1,72x of speedup with respect to the DNDP sequential, and 1,84x with the BBB.

We can conclude that using the Java Virtual Machine for executing the DNDP algorithm is not efficient enough. The speedup is not as much as expected when simulating 16 environments over a 4-core based processor in our experiments. Although being powerful enough for simulating most of the defined PDP system based models, the new models under development require a huge amount of resources, much higher than the resources in a personal computer using Java. For this reason, we will use our Java implementations in *pLinguaCore* for validation processes. But to better improve the performance, we will construct C++ based implementations by using libraries that let us manage real parallelism on parallel platforms.

7.2 DCBA implementation in pLinguaCore

pLinguaCore has been upgraded to provide an implementation of the DCBA, thus extending its existing probabilistic model simulation algorithms support. Along with the inclusion of other extensions, regarding to models such as Spiking Neural P Systems and Numerical P Systems, a new version of the library, named *pLinguaCore 3.0*, and featuring an implementation of the introduced DCBA can be downloaded from [16].

This Java implementation of the DCBA, within the *pLinguaCore* library, aims to serve as a validation framework for the models and the algorithms. Therefore, we are not going to consider the performance for this time, and move forward for the C++ and CUDA versions.

In what follows, details of the implementation of the DCBA in *pLinguaCore* are shown. Data structures, methods, code optimization and bug fixes are reviewed. Going Top-down, Java classes involved in the implementation:

- *DynamicMatrix.* It provides an implementation for the main operations of the DCBA.

  DynamicMatrix is built as a dynamic map indexed by MatrixKey class objects. MatrixKey objects are implemented as a pair of (MatrixRow, MatrixColumn) class objects. Associated to each MatrixKey object within the map, multiplicity k of the object specified by the *MatrixRow row* in the left hand side of the rule specified by the *MatrixColumn column* is stored. Note that k is stored instead of $1/k$ for accuracy reasons.

  As different filters are applied over the DynamicMatrix object, a couple of lists of MatrixRow and MatrixColumn objects respectively are associated to the matrix to keep track of its *valid cells*. Removal of elements from these lists is performed when filters are applied, while the DynamicMatrix object itself is reset in every step of the main loop of selection phase. Thus, DynamicMatrix object can be viewed as a *hash table* of multiplicities that allows a significant reduction of the required amount of memory for execution of the DCBA.

Also, attributes that store the sum of the multiplicities of the objects in the matrix by row as well as the minimum of the columns are included in the DynamicMatrix class. Inconsistent blocks are controlled by means of a list of pairs of MatrixColumn objects.

DynamicMatrix class directly extends from StaticMatrix class. Methods in DynamicMatrix implement the DCBA different phases themselves, remarkably:

- *initData()* initializes valid rows and columns lists in the DynamicMatrix object, clearing up them; also application of rules data structure is initialized.
- *filterColumns1()* computes valid columns and associates them to the DynamicMatrix object; applies Filter 1 to these columns;
- *filterColumns2()* applies Filter 2 to valid columns associated to the DynamicMatrix object, removing the required ones.
- *checkMutualConsistency()* checks mutual consistency over blocks of the DynamicMatrix object; if any inconsistency is found, an exception is thrown and execution of the simulator is stopped; a message listing the mutual inconsistent blocks found is shown to the user.
- *initFilterRows()* computes valid rows and associates them to the DynamicMatrix object; applies Filter 3 to these rows.
- *filterRows()* applies Filter 3 to valid rows, removing the required ones; this method is called inside the main loop of the selection phase, while the previous one is called outside, at the beginning of this phase.
- *normalizeRowsAndCalculateMinimums()* implements the main loop of selection phase.
- *maximality()* implements the maximality phase.

  – *executeRules()* implements execution phase; remarkably, multinomial distribution is computed by using binomial distributions, implemented through the specialized CERN Java library (`cern.jet.random.Binomial`).

- *StaticMatrix.* Provides an implementation for the static matrix used by the DCBA. Similarly to DynamicMatrix class, cells within the matrix are stored as a map indexed by MatrixKey class objects, each one of them associated to a multiplicity. A couple of immutable lists of MatrixRow and MatrixColumn class objects determines the structure of the matrix. Contents of the cells are fixed once initialized.

- *MatrixRow.* Provides an implementation for rows featured in DynamicMatrix, StaticMatrix and MatrixKey objects. Implemented by a pair of String objects representing *object and membrane label*, respectively, it also provides a method for computing the validity of the row, i.e. to determine if the row has to be kept within the DynamicMatrix object with respect to a given environment.

- *MatrixColumn.* Provides an implementation for columns featured in DynamicMatrix, StaticMatrix and MatrixKey objects. An abstract class is extended and implemented by a couple of classes representing the two kinds of rule blocks:

  – SkeletonRulesBlock, which implements blocks of skeleton rules.

  – EnvironmentRulesBlock, which implements blocks of environment rules.

  Both classes have the same structure: a *single* object to store the common LHS of the rule, plus a *collection* to store the several RHS objects that conforms the block. Also, each one provides an specific method for computing the validity of the corresponding column within the dynamic matrix.

To conclude, let us note that while conducting the DCBA implementation, several bugs have been fixed in *pLinguaCore*, notably some of them regarding to the way in which rules are parsed and stored, thus applying beyond the scope of the DCBA an affecting to implementation of probabilistic models simulators as a whole:

- Multisets of objects are now taken into account while checking rule blocks. In previous versions of *pLinguaCore*, when checking the consistency of probabilities of a rule block was conducted (i.e. checking that sum of probabilities of the rules must equal to one), multiplicities of objects in the left hand side of the rules were ignored.

- Issues with “intentional duplicate rules” solved assigning a unique identifier for every rule within the scope of probabilistic models. Issues found were:

  – Instantiation of parameters in syntactically different rule schemes for some models produced duplicated rules and caused the parser to throw an error and halt. As this duplicity proved intentional, the parser was modified subsequently to take it into account.

  – Probability was not taken into account when differencing rules. This made the parser to discard a rule syntactically identical, except for its probability, to a previous parsed one.

7.3 DCBA implementation in C++

As shown in Section 7.2, the DCBA was first implemented inside the *pLinguaCore* framework [99, 98, 16]. This version (hereafter *pdp-plcore-sim*) was validated towards a real ecosystem model (as shown in Chapter 6), reproducing the same data as the actual measurements. However, the performance was low since it as part of the *pLinguaCore* was written in Java.

Our first approach for making our implementation more efficient, was to develop a *stand alone* simulator written in C++ [103] (called *pdp-seq-sim*). We choose C++ for its similarity to the common GPGPU (General Purpose computations on Graphics Processing Units) languages of OpenCL and CUDA, and the support of many parallel libraries, such as OpenMP, PThreads and MPI.

We have designed an implementation which saves on memory by avoiding the creation of a static table for phase 1 of the DCBA. The implementation of this table can be inefficient in systems with a large number of rule blocks and/or objects. Therefore, the main challenge at phase 1 is the construction of the expanded static table $\mathcal{T}_j$. The size of this table is $O(|B| \cdot |\Gamma| \cdot (q+1))$, where $|B|$ is the number of rule blocks, $|\Gamma|$ is the size of the alphabet (amount of different objects), and $q+1$ corresponds to the number of membranes plus the space for the environment.

A full implementation of $\mathcal{T}_j$ can be expensive for large PDP systems. Moreover, it is a sparse matrix, having null values in the majority of the positions: competitions for one object appears for a relatively small number of blocks. This problem was overcome in the *pdp-plcore-sim* by using a hash table storing only non-null values. For *pdp-seq-sim*, the idea was to avoid the construction of $\mathcal{T}_j$, by translating the operations over the table to operations directly to the rule blocks information (using some of observations made in Chapter 6):

- Operations over columns: they can be transformed to operations for each rule block and the objects appearing in the multisets of the LHS.
- Operations over rows: they can be translated similarly to operations over columns, but the partial results are stored into a global array (one position per row).

Phase 1 can be implemented as described in Algorithm 7.3.1. Note that FILTER 3 is not needed any more. Although the full table is not created, some auxiliary data structures are used to virtually simulate it (we say it uses a *virtual table*):

- $activationVector$: the information of filtered blocks is stored here as boolean values. The full global size is $O(|B| * m * nsim)$, where m is the number of environments and $nsim$ the number of simulations carried out in parallel. This vector is actually implemented passing from boolean to bits.
- *addition*: the total calculated sums for rows are stored here, one number per each pair object and region. Its size is of order $O(|\Gamma| * (q + 1) * m * nsim)$.
- $MinN$: the minimum numbers calculated per column are stored here. This is needed in order to substract the corresponding number of applications to C'_t in each loop for the A value. The total global size is $O(|B| * m * nsim)$.
- *BlockSel*: the total number of applications for each rule block is stored here. The total global size is $O(|B| * m * nsim)$.
- *RuleSel*: the total number of applications for each rule is stored here. The total global size is $O(((|\mathcal{R}| * m) + |\mathcal{R}_E|) * nsim)$, where $|\mathcal{R}|$ is the number of evolution rules and $|\mathcal{R}_E|$ the number of communication rules.

Algorithm 7.3.1 Implementation of selection Phase 1 with virtual table

```
1: for $j = 1, \ldots, m$ do                                          ▷ For each environment
2:     for all block $B$ do
3:         $activationVector[B] \leftarrow true$
4:         if $charge(LHS(B))$ is different to the one presented $C'_t$ then
5:             $activationVector[B] \leftarrow false$                  ▷ (Apply FILTER 1)
6:         else if one of the objects in $LHS(B)$ does not exist in $C'_t$ then
7:             $activationVector[B] \leftarrow false$                  ▷ (Apply FILTER 2)
8:         end if
9:     end for
10:    Check the mutual consistency of blocks.
11:    repeat
12:        for all block $B$ having $activationVector[B] = true$ do    ▷ (Row sums)
13:            for each object $o^k$ appearing in $LHS(B)$, associated to region $i$ do
14:                $addition[o,i] \leftarrow addition[o,i] + \frac{1}{k}$
15:            end for
16:        end for
17:        for all block $B$ having $activationVector[B] = true$ do    ▷ (Col. min.)
18:            $MinN[B] \leftarrow Min_{[o^k]_i \in LHS(B)}(\frac{1}{k^2} * \frac{1}{addition[o,i]} * C'_t[o,i])$.
19:            $BlockSel[B] \leftarrow BlockSel[B] + MinN[B]$.
20:        end for
21:        for all block $B$ having $activationVector[B] = true$ do    ▷ (Updating)
22:            $C'_t \leftarrow C'_t \; - \; LHS(B) * MinN[B]$
23:        end for
24:        Apply FILTER 2 again (as described in step 6).
25:        $a \leftarrow a + 1$
26:    until $a = A$ or for each active block $B$, $MinN[B] = 0$
27: end for
```

7.4 DCBA parallel implementation for multicore platforms with OpenMP

As mentioned in the previous section, before we introduced parallelism, we first rewrote the simulator in C/C++ which is advantageous because OpenMP, PThreads and MPI are supported. In this section, we describe the implementation of the three forms of parallelism added to our simulator. A comparison of the three different techniques is also included, with a discussion of their strengths and weaknesses, and a evaluation of their performance on two generations of Intel processors with large memory sub-system differences. We will call this new simulator *pdp-omp-sim*.

From our analysis of these results we identify further ways to improve the running time of our simulator, by minimizing memory and cache bottlenecks using data compression and GPU computing. Ideas and references on compression and GPU computing can be found in [23].

7.4.1 DCBA parallel design

Our new implementation is parallelized in three ways: 1) simulations, 2) environments and 3) a hybrid approach. All of them are implemented using the parallel standard library for multicore platforms, OpenMP [15].

- *Simulations* are parallelized by using the `#pragma omp parallel for` OpenMP directive [15] on a simulation loop. This outermost loop for simulations can easily be added in the main procedure of the DCBA (Initialization, Selection and Execution stages must be executed in each iteration) [103]. The advantage of running simulations in parallel is there are no data dependencies between simulations, and, therefore, the problem is embarrassingly parallel. Also, the users of the simulator typically run 50 to 100 simulations of each set of input parameters, so there are enough simulations to consume all cores.

  However, there are disadvantages of running simulations in parallel. Each simulation needs its own memory space increasing the amount of memory used. If the number of simulations is not divisible by the number of processors then load balancing issues can occur with the final simulations running while some cores are idle. Also, running simulations in parallel can result in resource conflicts as cores compete for shared resources.

- *Environments* are parallelized by using the `#pragma omp parallel` [15] to generate a thread pool for the simulation. Then the **for** loops in Algorithm 7.3.1 that iterate over environments are parallelized with `#pragma omp for` [15], which has an implicit barrier that enforces the dependencies between the stages in each time step. Using this design, creating new thread blocks for each for loop is avoided.

The advantage of parallelizing environments over simulations is that memory usage does not increase. However, dependencies occur twice in each time step requiring synchronization steps. Also, since most models use 5 to 30 environments, there are cases where modern machines have more cores than environments and just parallelizing environments cannot take advantage of all computing resources. In addition, as with simulations, load balancing can be an issue if the number of environments is not divisible by the number of cores, or if the runtime of environments varies.

- *Hybrid* parallelization is accomplished by combining parallel environments with parallel simulations. We accomplish hybrid parallelization through command-line flags that allow the specification of how many environments or simulations to run in parallel. By combining both forms of parallelism, we can balance the amount of each resource used. This will become more important as the number of cores within a node increases. For example, the number of simulations can be increased until available memory is used and then environments within each system can be parallelized.

7.4.2 Experimental evaluation

In this section, we describe a series of tests performed on our implementation and the systems they were run on, along with the results from those experiments.

7.4.2.1 Test environment and methodology

The following experiments were run on the two machines shown in Table 7.1. The tests used random systems with similar amounts of data to real-life examples. Multiple configurations with environments and simulations varying from 10 to 50 were tested for the parallel environments, simulations and two hybrid combinations. Each of these tests were run on 1 to 8 cores for the Intel i5 machine, and 1 to 4 for the Intel i7. The measurements in this section, except when noted, correspond only to the parallelized part of the code.

Processor	**Speed**	**Bus speed**	**Cache**
i5 Nehalem (2x4)	2 Ghz	3x800 Mhz	2x4 MB
i7 Sandy Bridge (1x4)	3.4 Ghz	2x1333 Mhz	8 MB

Table 7.1: Specifications of the test machines.

Processor	**Setup**	**10 env & sim**	**50 env & sim**
Nehalem	0.8 s	48.0 s	251.0 s
Sandy Bridge	0.35 s	19.9 s	97.8 s

Table 7.2: Serial Runtimes

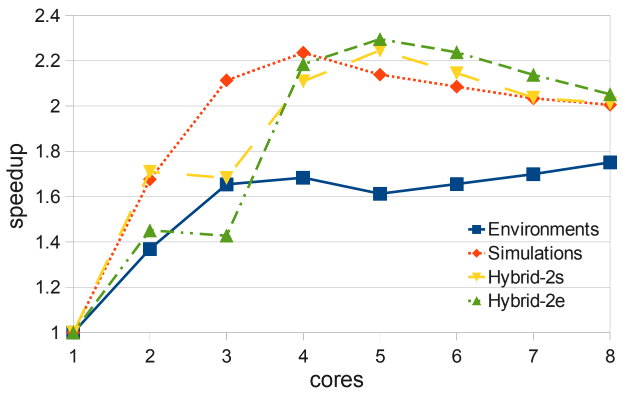
7.4.2.2 Results

The serial running time on both of our test machines is shown in Table 7.2. *Setup* is the cost of running the serial portion of the code. The other two columns represent the runtime extremes of our test cases when run in serial. From the table, we can see that in serial the setup portion is a small part of the overall runtime, and that the Sandy Bridge processor is about 2.5 times faster than the Nehalem.

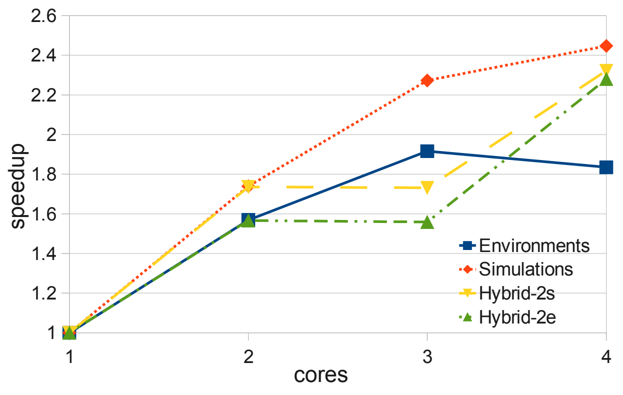
Figures 7.2 and 7.3 show the performance improvements of parallelizing our system in various ways. The two figures are representative of the other tests we performed with the best performance either being parallelizing by simulations or the hybrid method (2s), which uses two simulations and then parallelizes by environments. Another trend shown is that as the number of simulations increases, the advantage of parallelizing by simulations increases. The same effect is observed for environments.

On the Sandy Bridge system, the largest speedup of 2.5x occurs for 50 simulations and 50 environments. However, the maximum speedup on Sandy Bridge when going from 3 to 4 processors is only between 0.1 and 0.2, suggesting that the calculation is memory bound for larger core counts. On the Nehalem machine, the maximum parallel speedup was 2.3x for all tests, which is barely greater than the added available bandwidth from using the second socket. These results, led us to suspect we that the Nehalem system's performance was being limited our programming approach. In particular, we did not account for the Non Uniform Memory Access (NUMA), memory subsystem of the two sockets.

One final test was run to see if NUMA was hurting the performance of our code on the Nehalem machine. First one and then two instances of the code was run with 4 threads each (affinity locked to different sockets) on the machine. With two instances, a 2x speedup was achieved over the best parallel results

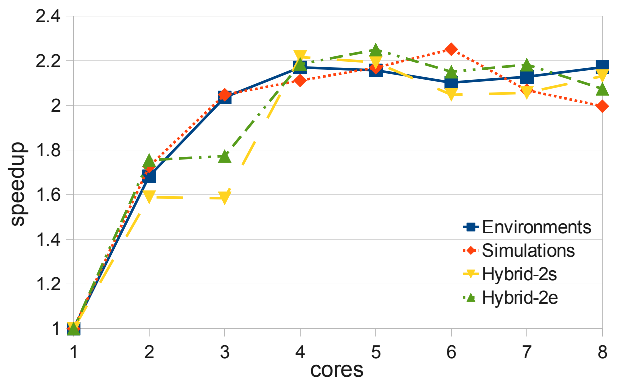


(a) Nehalem

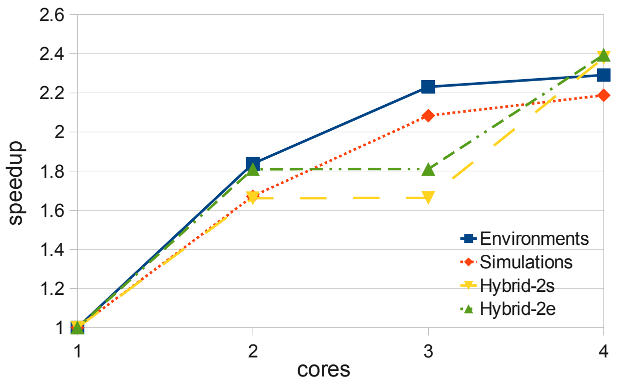


(b) Sandy Bridge

Figure 7.2: Speedups running 50 simulations with 10 environments in the system



(a) Nehalem



(b) Sandy Bridge

Figure 7.3: Speedups running 10 simulations with 50 environments in the system

from running one instance of the OpenMP version. Confirming this result is that locking all 4 threads to a single socket, performance results in a 50% performance increase when compared to locking 2 threads to each socket. For the current tests, however, affinity is controlled by the operating system and performance is similar to when two threads were locked to each socket. These preliminary tests also indicates that the code is memory bound since overall speedups on 8 cores were less than 5x.

In conclusion, experiments ran to test the simulator indicate the simulations are memory bound and the portion of the code we parallelized consumes over 98% of the runtime in serial. From this initial work we conclude that parallelizing by simulations or hybrid techniques yields the largest speedups. Also, using hardware, such as Intel's Sandy Bridge that has more memory bandwidth, is an easy way for scientists to improve the speed of our simulator. It can also be concluded that performance tuning to decrease data movement is important for P-system simulators, since P systems are memory bound computations.

7.5 DCBA parallel implementation on the GPU with CUDA

Our previous simulator, *pdp-omp-sim* [103], is the starting point of the new implementation using CUDA. In this new simulator (let call it *pdp-gpu-sim*) [104], the code and the data structures have been optimized, saving up to 27% of memory. We have also adapted *pdp-omp-sim* to these, achieving better speedups (1.25x for large systems).

Normally, the end user (i.e., ecological experts and model designers) runs many simulations on each set of parameters to extract statistical information of the probabilistic model. This can be automated by adding an outermost loop for simulations in the main procedure of the DCBA. This loop is easily parallelized. Indeed, our tests of *pdp-omp-sim* conclude that parallelizing by simulations or a hybrid technique (simulations plus environments) yields the largest speedups.

At first glance, these two levels of parallelism (simulations and environments) could fit the double parallelism of the CUDA architecture (thread blocks and threads). For example, we could assign each simulation to a block of threads, and each environment to a thread (since they require synchronization at each time step). However, the number of environments depends inherently on the model. Typically, 2 to 20 environments are considered, which is not enough for fulfilling the GPU resources. Number of simulations typically range from 50 to 100, which is sufficient for thread blocks, but still a poor number compared to the several hundred cores available on modern GPUs.

We therefore also parallelize the execution of rule blocks. Hence, our simulator can utilize a huge number of thread blocks by distributing simulations (parallel simulations, as memory can store them) and environments in each one, and process each rule block by each thread. Since there are normally more rule blocks (thousand of them) than threads per thread block (up to 512), we create 256 threads which iterate over the rule blocks in tiles. This design is graphically shown on Figure 7.4. Each phase of the algorithm has been designed following the general CUDA design explained above, and implemented separately as individual kernels. Thus, simulations and environments are synchronized by the successive calls to the kernels.

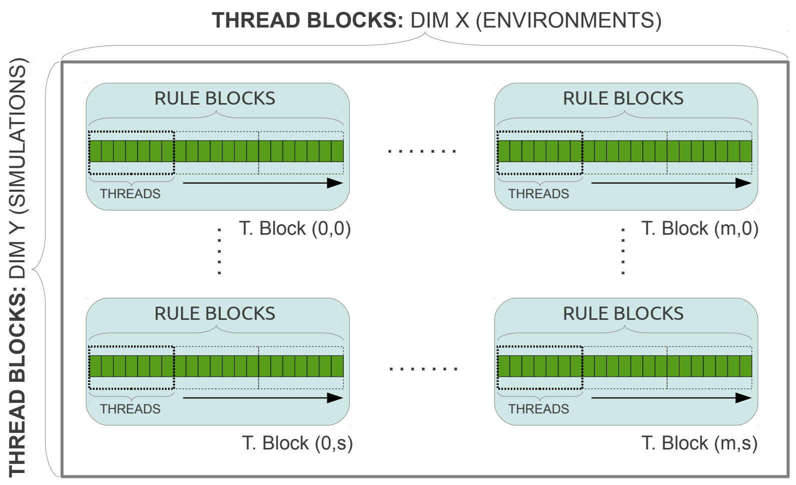


Figure 7.4: General design of our CUDA-based simulator: 2D grid, and 1D thread blocks. Threads loop the rule blocks in tiles.

7.5.1 GPU implementation of the DCBA phases

7.5.1.1 Implementation of Selection Phase 1

The main challenge at this phase is the construction of the expanded static table $\mathcal{T}_j$. The size of this table is $O(|B| \cdot |\Gamma| \cdot (q+1))$, where $|B|$ is the number of rule blocks, $|\Gamma|$ is the size of the alphabet (amount of different objects), and $q+1$ corresponds to the number of membranes plus the space for the environment.

A full implementation of $\mathcal{T}_j$ can be expensive for large PDP systems. Moreover, it is a sparse matrix, having null values in the majority of the positions: competitions for one object appears for a relatively small number of blocks. This problem was overcome in the *pdp-plcore-sim* by using a hash table storing only non-null values. For *pdp-omp-sim*, the idea was to avoid the construction of $\mathcal{T}_j$, by translating the operations over the table to operations directly to the rule blocks information (using the observations made in Section 6.6):

- Operations over columns: they can be transformed to operations for each rule block and the objects appearing in the multisets of the LHS.
- Operations over rows: they can be translated similarly to operations over rows, but the partial results into a global array (one position per row).

Phase 1 can be implemented as described in Algorithm 7.3.1. Note that Filter 3 is not needed any more. Although the full table is not created, some auxiliary data structures are used to virtually simulate it (we say it uses a *virtual table*):

- $activationVector$: the information of filtered blocks is stored here as boolean values. The total global size is $O(|B| * m * nsim)$, where m is the number of environments and $nsim$ the number of simulations carried out in parallel. This vector is actually implemented passing from boolean to bits.
- *addition*: the total calculated sums for rows are stored here, one number per each pair object and region. Its size is $O(|\Gamma| * (q + 1) * m * nsim)$.
- $MinN$: the minimum numbers calculated per column are stored here. This is needed in order to substract the corresponding number of applications to C'_t in each loop for the A value. The total global size is $O(|B| * m * nsim)$.
- $BlockSel$: the total number of applications for each rule block is stored here. The total global size is $O(|B| * m * nsim)$.
- $RuleSel$: the total number of applications for each rule is stored here. The total global size is $O(((|\mathcal{R}| * m) + |\mathcal{R}_E|) * nsim)$, where $|\mathcal{R}|$ is the number of rules and $|\mathcal{R}_E|$ the number of communication rules.

The implementation on the device has been constructed directly from Algorithm 7.3.1. Phase 1 has been implemented using several kernels, avoiding the overload of only one:

- Kernel for Filters (from line 2 to 10 in Algorithm 7.3.1): Filters 1 and 2 are implemented here by using our general CUDA design (Figure 7.4).
- Kernel for Normalization (from line 11 to 20): the two parts for row additions and minimum calculations (called normalization step) are implemented in a kernel. They are synchronized by *synchtreads* CUDA instruction. The work assigned to threads is divergent (scatter operation), that is, each thread works with one rule block, but writes information for each object appearing in the LHS. Therefore, the writes to *addition* are carried out by atomic operations.
- Kernel for Updating and Filter 2 (from line 21 to 26). As before, the work of each thread is divergent (scatter operation). Thus, the update of the configuration is also implemented with atomic operations.

Finally, we also had to deal with a GPU constraint concerning floating point operations. We found that on many NVIDIA GPUs (specially, in our GPUs with compute capability 1.3), addition and multiplication are IEEE 754-compliant operations, so they generate the same results than using a commodity CPU. However, many other operations, such as square root and division, result in the floating point value closest to the correct mathematical result [154, 5].

Executing Phase 1 many times with very large models, we find many discrepancies with the results from the CPU version. These differences, in terms of rule blocks selection number, are small (around only 1 or 2 units). However, this leads to miss information, and sometimes to incorrect results.

The problem takes place at line 14 in Algorithm 7.3.1. Initially, we have implemented those divisions and additions by using double values for the addition vector to avoid losing accuracy. However, when using GPUs with compute capability 1.2, they are demoted again to floats. Double operations are also inefficient. Therefore, our solution was to use two integer values representing the fractions in the addition vector, so that the real division is performed only at the end of the accumulative process (line 18 in Algorithm 7.3.1).

In order to avoid overflows, we have implemented a loop at the beginning of the algorithm. This loop initializes the addition vector to the total sums using the static table. Later on, the addition value is calculated by subtracting the values of filtered rule blocks, instead of summing those that are active.

7.5.1.2 Implementation of Selection Phase 2

Phase 2 is the most challenging part when parallelizing by blocks. The selection of blocks at this phase is performed in an inherently sequential way: we need to know how many objects a block can consume before selecting the next one. In our solution, Phase 2 is implemented by one kernel, using our general CUDA design.

The random order to the blocks is *simulated* by the CUDA thread scheduler: each thread calculates the position in the order of its rule block by using the *atomicInc* operation. Since it does not perform a real random order, random numbers are going to be used soon in next versions. Our first approach (let designate it *ph2-simorder-oneseq*) for Phase 2 was to launch 257 threads: 256 threads to calculate the "random" order, and an extra thread to iterate the blocks in that order, selecting and consuming the LHS. Since this approach is still sequentially executed in the GPU, an improved version was constructed.

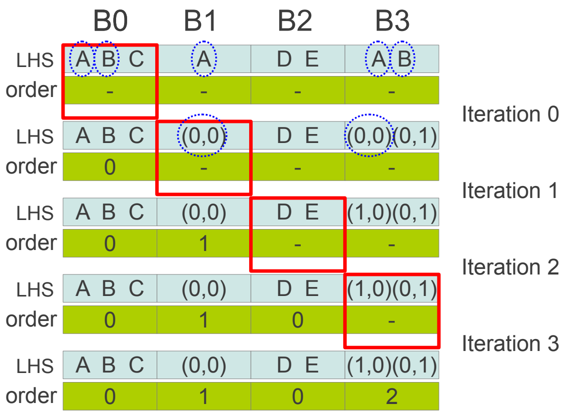


Figure 7.5: Sample of our *ph2-simorder-dyncomp* kernel execution.

Our new version (designated *ph2-simorder-dyncomp*) dynamically checks the blocks that are really competing for objects, and calculates which blocks can be selected in parallel, and which depend on the selection of the others. To do this, some previous computations are needed. Two arrays are used, one storing the information of the LHS, and another storing the selection order (rule blocks having the same selection order number will be selected in parallel). Both arrays are implemented using the GPUs shared memory to speedup this computation. Shared memory on the GPU is one of the on-chip memory spaces which is shared by all the SPs of a SM. Access times to the shared memory are comparable to those of a L1-cache on a traditional CPU.

Recall that GPUs also feature high-speed DRAM memory (device memory) with higher latency than on-chip memory (typically hundreds of times slower). The device memory is subdivided in read-write, non-cached (global and local) and read-only, cached (texture) areas.

Figure 7.5 shows a sample *ph2-simorder-dyncomp* kernel execution. We iterate for each rule block (using the pre-calculated random order). First, the rule blocks check if they have common objects with block $B0$. In the example, block $B1$ has object A, and block $B3$ has objects A and B. They annotate this competition with the pair $(block, object)$, using the indexes of the array. The current block also calculates the selection order by checking whether it has some depending objects. If so, the order is increased by one. For the first iteration, block $B0$ is assigned order 0, but in iteration 1, block $B1$ is assigned order 1 (competing with block $B0$). The rest of the iteration can be seen in Figure 7.5.

Our experiments shows that *ph2-simorder-dyncomp*, that includes extra computations but allows to execute independent blocks in parallel, achieves up to 20% of performance improvement from *ph2-simorder-oneseq*.

7.5.1.3 Implementation of Selection Phase 3

Phase 3 calculates the number of times a rule is applied using a binomial distribution, and the selected block number, both implemented in one kernel.

For random binomial number generation, we have made a *CUDA library* based on *cuRAND* [14], called *cuRNG_BINOMIAL* (see next subsection 7.5.2 for details). This module implements the BINV algorithm proposed by *Voratas Kachitvichyanukul* and *Bruce W. Schmeiser* [87]. Algorithm BINV executes with speed proportional to $n \cdot p$ and has been improved by exploiting properties listed in the paper [87]. Also, it has got the best results assuming a normal probability approximation when $n \cdot p > 10$.

The library implements an *inline device function* which executes binomial randomization (BINV) when $n \cdot p \leq 10$ and normal randomization (cuRAND), otherwise. Our implementation generates binomial random numbers while running the kernel; thus, they are not generated previously.

The implementation of the phase is directly translated from the pseudocode of the DCBA. Also, it has got the best parallelism exploiting until now, comparing to other phases of the algorithm.

7.5.1.4 Implementation of Execution (Phase 4)

Phase 4 is implemented as directly shown in the DCBA pseudocode using our general CUDA design. In this case, we go to another level of parallelism for threads, that now works with each rule. As before, threads iterate the rules by tiles, and adding the corresponding RHS (if it has a number of applications $N_r > 0$). Finally, since this operation is scatter or divergent (from rules to add objects), we use atomic operations again to update the configuration of the system.

7.5.2 Random binomial variate generator on the GPU

We have to take into account the generation of random numbers when working with probabilistic algorithms. The DCBA considers the multinomial distribution. In fact, it is calculated by using the binomial distribution for each of the random variables (i.e. number of selected applications for each rule). In the next subsections we will introduce the binomial distribution of probability, the random number generation problem, and a description of our new library for CUDA, called cuRNG_BINOMIAL.

7.5.2.1 The binomial distribution

The DCBA uses random multinomial variables, as stated in phase 3. This fact simplifies the distribution of objects along the possible evolutions that it can suffer from the system. It is based on the idea of deciding the evolution of a group of individuals. The decision is made by the system, which imposes the rules of the model (abstracted directly from nature). This idea has been successfully applied to develop comprehensive and modular models fitting the reality.

The experiment that typically explain the multinomial distribution is the following [91]: b balls are thrown into k bins, where the probability of a ball falling in the i-th bin is p_i. The probability mass function[2] of the normal distribution is

$$f(x_1, \dots, x_k; n, p_1, \dots, p_n) = n! \prod_{i=1}^{k} \frac{p_i^{x_i}}{x_i!}$$

where $n \geq 0$; $x_i \geq 0, p_i > 0, 1 \leq i \leq k$; $\sum_{i=1}^{k} x_i = n$; and $\sum_{i=1}^{k} p_i = 1$. We write the distribution as $M(n, p_1, \dots, p_k)$

[2]The probability mass function gives the probability that a discrete random variable is exactly equal to some value

The multinomial distribution is a generalization of the binomial distribution [91]. The latter describes the total number of successes in a sequence of n independent Bernoulli trials, which has two outcomes (success or failure). In the multinomial distribution, the categorical distribution is used, where each trial results in exactly one of some fixed finite number k of possible outcomes, with probabilities $p_1, ..., p_k$, as denoted before.

A set of random numbers following the normal distribution can be calculated through random binomial numbers. Actually, this is implicitly performed in the BBB algorithm (see Algorithm 6.4.2 in Chapter 6). In summary, for each $k-1$ of the k choices, we calculate a binomial random number using n and p_i. The number n has to be updated according to the previously calculated random number. Moreover, we need to normalize the remaining probabilities. Finally, the last choice is directly assigned the remaining number n.

The probability mass function of the binomial distribution is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

where $x \in \{0, 1, 2, ..., n\}$ and $0 \leq p \leq 1$, but never $x = 0 \wedge p = 0$. We write the distribution as $B(n, p)$. The expectation, or mean value, is given by $E = n \cdot p$, and the variance is $Var = n \cdot p(1-p)$. A very important property is that the distribution converge to the normal distribution, when n increases. It is a direct consequence of the Central Limit Theorem. The approximation of the distribution is as follows: $B(n, p) \approx N(n \cdot p, n \cdot p(1-p))$. This approximation is accurate enough when $n \cdot p > 10$ [91] (for other sources when both $n \cdot p$ and $n(1-p)$ are greater than 5).

The normal distribution is considered the most prominent probability distribution in statistics. There are three major reasons for this [36]: it is very tractable analytically, the symmetry of its familiar bell shape (Gaussian function) is very useful and it can be used to approximate many other distributions in large samples (by the Central Limit Theorem). It is denoted by $N(\mu, \sigma^2)$, where μ is the expectation (mean), and σ^2 is the variance. The distribution $N(0, 1)$ is called the standard normal distribution. Furthermore, it is possible to relate all normal random variables to the standard normal. That is, if $X \sim N(\mu, \sigma^2)$, then the random variable $Z = \dfrac{(X-\mu)}{\sigma}$ holds $Z \sim N(0, 1)$ [36].

Finally, we will introduce perhaps one of the most important distributions in computing. It is the uniform distribution, which is denoted by $U(a, b)$, what means that it is defined in the interval $[a, b]$. All intervals of the same length on the distribution's support are equally probable.

7.5.2.2 Generation of random numbers

We can think on a *random number generator (RNG)* as a procedure to generate numbers without any pattern, in a unpredictable and unreproducible way. Formally, it generates an infinite stream of random variables that are independent and identically distributed according to some probability distribution [91]. Today, we can use the quantum physical laws as a source of randomness [84]. However, these processes are expensive to handle, and it is still hard to find devices based on this concept. Moreover, physics still have to demonstrate a quote from A. Einstein: *"As I have said so many times, God doesn't play dice with the world"*.

Current random number generators on computers are based on software applications whose algorithms are based on ordinary arithmetics. These are called *pseudorandom number generators (PRNG)*. They generate a deterministic sequence of numbers from an initially requested number, called seed. Although the sequence of numbers is deterministic, the sequence to use is given randomly by the *seed*. This property makes PRNG to be very useful for their speed in number generation and their reproducibility.

Other popular RNGs on computers are quasirandom methods (QRNG). The quasirandom (also known as low-discrepancy) sequences try to fill the space of numbers more uniformly than uncorrelated random points. They are not random at all, since they apply a maximally avoiding strategy, but they are very useful on several simulation scenarios. Therefore, they do not have to be confuse with PRNG methods.

A random number generator is good depending on many factors. Below we list some desirable properties [91]:

- Pass statistical tests supporting the uniformity of random numbers.
- A theoretical base. As stated by D. Knuth (1998), *"Random numbers should not be generated with a method chosen at random"*.
- Reproducible stream of numbers for testing.
- Fast and efficient generation of random numbers.
- A large period of the PRNG. It is defined as the smallest number of steps taken before entering a previously visited state of random numbers. It should be extremely large, on the order of 10^{50}. In order o produce N numbers, some authors consider that the period length should be at least $10N^2$ [91].

- Provide multiple streams to produce sequences in parallel.

Most PRNG algorithms produce uniformly distributed sequences. In these cases, the generator is said to be a uniform random number generator. Random numbers selected from a non-uniform distribution can be generated using a uniform distribution and a function to relate them. Indeed, a standard normal variable can be generated using two uniform variables. This is performed by the box-muller transformation [29]. This method works as follows: consider X_1 and X_2 two independent random variables uniformly distributed in the interval (0,1). Then, the random variables Z_1 and Z_2, defined as $Z_1 = \sqrt{-2 \ln U_1} \cos 2\pi U_2$ and $Z_2 = \sqrt{-2 \ln U_1} \sin 2\pi U_2$, are independent random variables with a standard normal distribution.

Finally, binomial random variables can be generated using several methods [87]. The algorithm BTPE [87] is currently the most used in several standard numerical libraries (e.g. GNU Scientific Library (GSL) [11]). On the contrary, BINV [87] is the algorithm that works better for small number of experiments (or binomial parameter n).

7.5.2.3 The cuRAND library

NVIDIA provides within its CUDA toolkit a library for random number generation. It is called cuRAND [14], and has the ability to generate pseudorandom and quasirandom numbers. The library has passed several statistical tests traditionally used for RNGs. The architecture of cuRAND consists of two parts, for both the host and device sides.

The host can use the cuRAND library for generating random numbers on the host itself, or on the device. The latter means to generate the random numbers on the device global memory, so that both the host or the device can use them for any randomized task. But on the other hand, the device can also use the library to generate random numbers "on the fly", inside the kernels. These are `inline` functions that can be called by the threads of a grid, and generate sequences in parallel.

There are five types of random number generators in cuRAND. One type is for pseudorandom number generation, and it is based on the XORWOW algorithm [14]. The other four types are variants of the SOBOL' quasirandom number generator [14]. The XORWOW generator in cuRAND is estimated to have a period length of 2^{67}. Furthermore, the library supports the pseudorandom number generation for the uniform and the normal distributions. The normal distribution is calculated through the box-muller transformation using the fast float functions on the GPU.

7.5.2.4 The cuRNG_BINOMIAL library

We have developed a new library for generating binomial random variates on the GPU. This is crucial for the correct development of the GPU simulator. This library is called cuRNG_BINOMIAL, and is based on the cuRAND library. The implementation is a header file with `inline` functions for the CUDA threads (as in the cuRAND).

The cuRNG_BINOMIAL library has been designed to run fast, in parallel and saving memory space on the GPU. Therefore, we cannot use very complex algorithms with many loops and conditions, such as the BTPE [87]. This leads to thread predication and warp divergences. The design is then based on the approximation of the binomial distribution by using the standard normal, which is efficiently generated by the cuRAND. However, we need another implementation for small values of the mean. In these cases, the algorithm utilized is the BINV, since it is the most efficient for small values [87] (it requires to generate only one random number, and has a small loop).

When a thread call the function for generating a random number that follows the distribution $B(n, p)$, the following takes place:

- If $n \cdot min(p, 1-p) \geq 10$, then the normal approximation is used. The cuRAND library calculates a normal random variable Z, and the function returns the value $Z\sqrt{np(1-p)} + np$.
- If $n \cdot min(p, 1-p) < 10$, then the BINV algorithm is used. It is applied as shown in Algorithm 7.5.1.

7.5.2.5 Testing the library

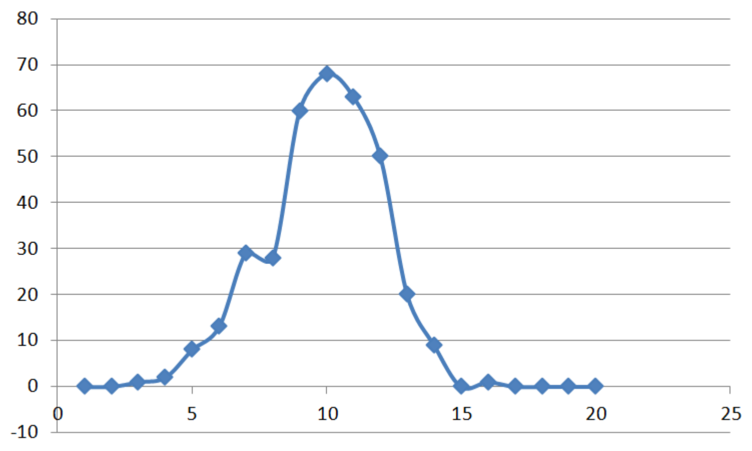
In order to test the cuRNG_BINOMIAL library, we have developed two benchmarks. The first one test the correctness of the library, and the other the performance.

Figure 7.6 shows two tests for the binomial behavior of the library. For the first test (Figure 7.6), 400 random numbers were generated using the binomial variable $X \sim B(20, 0.5)$. Because the mean for this random variable is exactly 10, the library applied the normal distribution for its minimum value.

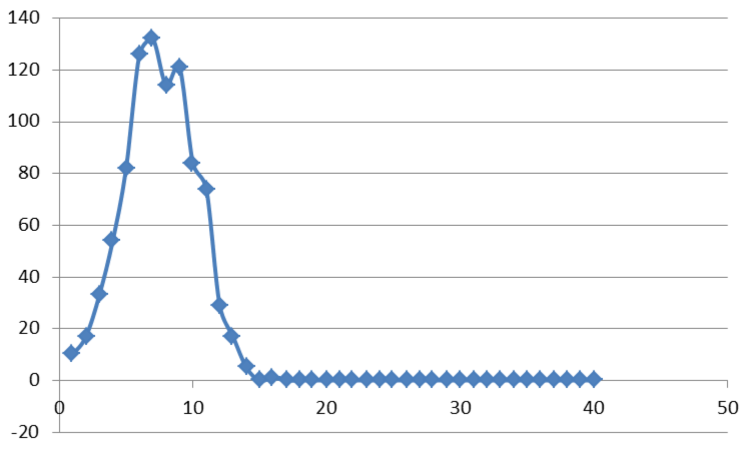
For second first test (Figure 7.6(b)), 600 random numbers were created using the binomial $Y \sim B(40, 0.2)$. In this case, since the mean for this variable is 8, the library used the BINV algorithm. We can see that the resulting graphs meets the binomial form: highest value for the mean, and almost null values for those far from the mean plus variance.

Algorithm 7.5.1 cuRNG_BINOMIAL implementation for the BINV algorithm

```
Require: $n$ and $p$ values.
 1: $q \leftarrow 1 - p$
 2: $s \leftarrow p/q$
 3: $a \leftarrow (n+1) \cdot s$
 4: $r \leftarrow q^n$
 5: $u \leftarrow curand\_uniform$          ▷ (generate a uniform random variable)
 6: repeat
 7:     $u \leftarrow u - r$
 8:     $x \leftarrow x + 1$
 9:     $r \leftarrow (a/x - s) \cdot r$
10:     if $r \leq 0$ then
11:         Break the loop repeat
12:     end if
13: until $u \leq r$
14: if $x > n$ then
15:     $x \leftarrow n$
16: end if
17: if $p \geq 0.5$ then
18:     $x \leftarrow n - x$
19: end if
20: Return $x$
```



(a) $B(20, 0.5)$



(b) $B(40, 0.2)$

Figure 7.6: Binomial random variates generated using cuRNG_BINOMIAL

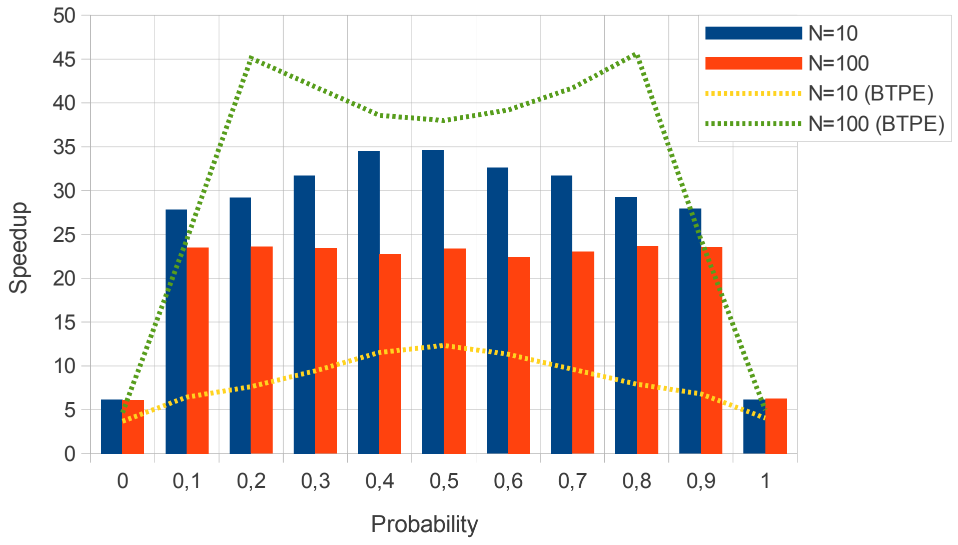


Figure 7.7: Performance of cuRNG_BINOMIAL generating 10^6 random numbers

The second benchmark is focused on the performance of the library, as well as on the correctness. We have executed two tests generating 10^6 random numbers each. They have been carried out on our GPU sever. We have implemented a CPU counterpart of the cuRNG_BINOMIAL library. This CPU version implements both the BINV algorithm and the normal approximation. Moreover, it uses the GSL library to generate the random numbers. The first test is based on the generation of binomial random numbers following the variables $X \sim B(10, p)$, $0 \leq p \leq 1$, and the second is based on the binomial $Y \sim B(100, p)$, $0 \leq p \leq 1$. Note that the library uses the BINV algorithm for the first test, and the normal approximation for the second one.

The results are shown in Figure 7.7. The speedups for the two tests are represented by bars ($N = 10$ for $X \sim B(10, p)$ and $N = 100$ for $Y \sim B(100, p)$, respectively). The library achieves better performance in the first tests, which is based on the application of the BINV algorithm. However, the approximation of the normal distribution (second test) is a bit less fast than the other approximation. The reason is that generation normal random numbers and its conversion to binomial implies more special arithmetic functions on the GPU (box-muller transformation), what is worst than generating just one random number and applying a small loop (BINV). Nevertheless, the GPU is able to generate those numbers much faster (with a maximum of 35x of speedup) than the CPU counterpart.

We have also expanded the two tests to compare our library, which implements a normal approximation, with the BTPE implementation on the GSL library. This is represented as lines in Figure 7.7. For the first test, we can see that the BINV algorithm behaves worst than the BTPE for small values, and therefore, the GPU achieves the half of the original speedup. However, for high values, as in the second test, the GPU normal approximation works better than the BTPE in terms of performance, increasing the performance up to 45x.

Recall that the graph is symmetric, so that the values for probabilities below and upper 0.5 are the same (we consider the minimum of p and $1-p$). Concerning the correctness, we have test that the 10^6 numbers generated meets the expected mean and variance for both random variables X and Y. This test includes the cuRNG_BINOMIAL library, as well as those based on the GSL library.

7.5.3 Performance results of the simulator

In order to test the performance of our simulators, we constructed a random generator of PDP systems (designated *pdps-rand*). These randomly created PDP systems have no biological meaning. The purpose is to stress the simulator in order to analyze the implemented designs with different topologies. *pdps-rand* is parametrized in such a way that it can create PDP systems of a desired size.

We benchmark our *pdp-gpu-sim* and *pdp-omp-sim* (for 1, 2 and 4 cores) by first analyzing the scalability when increasing the size of the system in several ways. We then profile the simulators, showing the percentage of time taken by each phase separately. All experiments are run on our GPU server: Linux 64-bit server, with a 4-core (2 GHz) dual socket Intel i5 Xeon Nehalem processor, 12 GBytes of DDR3 RAM and two NVIDIA Tesla C1060 graphics cards (240 cores at 1.30 GHZ , 4 GBytes of memory). GPU cores are typically slower than CPU cores.

Figure 7.8 shows the scalability of the simulator when the number of different objects appearing in the LHS (cooperation degree (see Definition 6.2)) increases. We can assume that, the greater the cooperation degree, the greater the number of competing blocks generated by *pdps-rand*. The figure shows the simulation time (in milliseconds) for one computation step running 50 simulations of PDP systems with 10 environments, 50000 rule blocks and 5000 different objects. The randomly generated PDP systems are sorted by the mean LHS length, showing that *pdp-gpu-sim* works better for lengths smaller

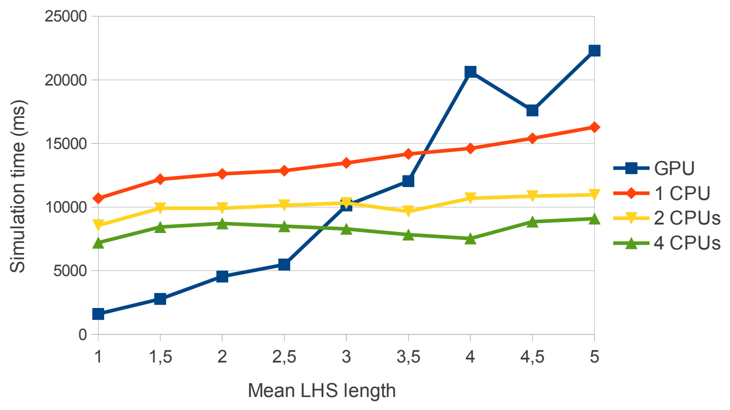


Figure 7.8: Scalability when increasing the mean LHS length of rules.

than 3. The speedup achieved by *pdp-gpu-sim* is 6.6x and 2.3x for lengths of 1 and 2 against *pdp-omp-sim* with one core, and 4.5x and 1.9x against *pdp-omp-sim* with 4 cores, respectively.

The second test analyses the performance when increasing the parallelism level of the CUDA threads within thread blocks, that is, the number of rule blocks. The speedup achieved by *pdp-gpu-sim* versus *pdp-omp-sim* is shown in Figure 7.9. The number of simulations is fixed to 50, and the environments to 20 (hence, a total of 1000 thread blocks). The number of objects is proportionally increased together with the number of rule blocks, in such a way that the ratio for number of rule blocks and number of objects is always 2. The mean LHS length is 1.5 (this is normal value for many real ecosystem models, as seen in the literature). The speedup gets stable to around 7x on the number of rule blocks for the GPU versus CPU. For the multicore versions with 2 and 4 CPUs, the speedups are maintained to 4.3x and 3x, respectively. In our experiments, this number is also achieved when running with 10^6 rule blocks.

The third test is for the second parallelism level in CUDA, concerning thread blocks. It is directly related with the number of environments and simulations. The result is shown in Figure 7.10. In this experiment, the number of rule blocks is fixed to 10000, the number of objects is fixed to 7024 and the mean LHS length is 2. The number of environments is fixed to 1 when increasing the simulations, and vice versa. As it can be seen, for low values, the speedup is demoted below 1. These values come from the fact of insufficient number of thread blocks to fulfill the GPU resources. Another trend shown is that when the number of simulations increases, the advantage of

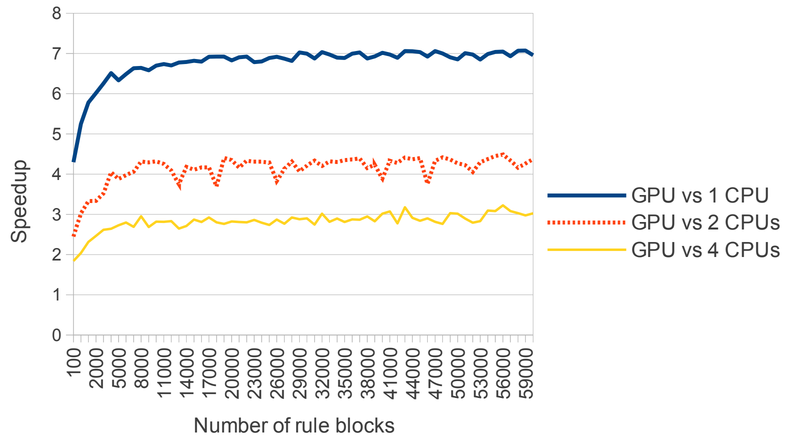


Figure 7.9: Scalability of the simulators when increasing the number of rule blocks.

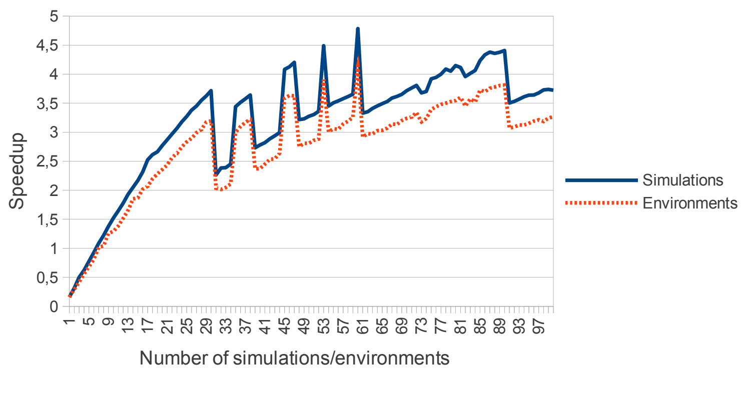


Figure 7.10: Scalability of the simulators when increasing the number of simulations and environments.

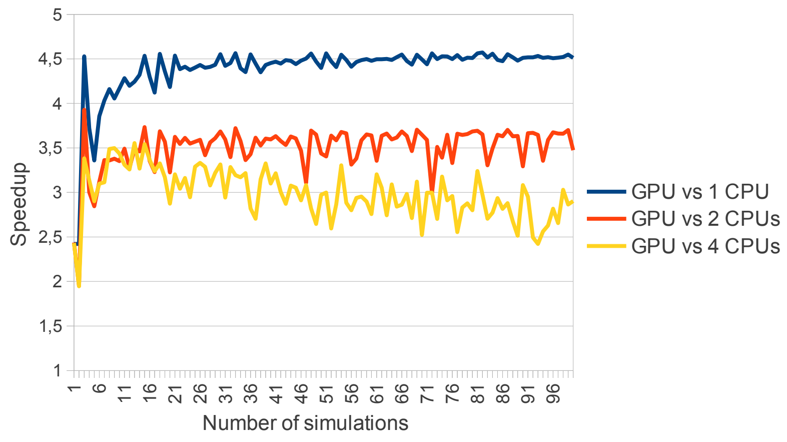


Figure 7.11: Scalability of the simulators when increasing the number of simulations.

parallelizing by simulations increases. The same effect is observed for environments. This trend is stabilized to 3.5x for high values. However the parallelism over simulations is better carried out by the GPU, giving lower speedups for environments.

As stated in [103], parallelizing by simulations yields the largest speedups on multicore platforms. Therefore, we finalize the first benchmark by comparing these results with the GPU. Rule blocks are fixed to 50000, environments to 20, objects to 5000 and mean LHS length to 1.5. As shown in Figure 7.11, the GPU achieves better runtime than the multicore implementations. The speedup is maintained to 4.5x using one core, 3.5x for 2 cores, and 2.7x for 4 cores.

The results of the second benchmark are shown in Table 7.3. This profile has been calculated running the simulator with 10000 rule blocks, 20 environments, 50 simulations, 5000 objects and two different mean LHS lengths, 1.5 (test A) and 3 (test B), respectively. Phase 1 is the most complex part in the simulation (taking more than 50% of the runtime on the CPU). In test A, the GPU implementation offers for phase 1 up to 14x of speedup. Therefore, the percentage of the execution time is decreased to 30%.

Following with Test A, Phase 2 takes only the 12% of the execution time on CPU. However, the GPU can only accelerate this phase by 2x. Therefore, this phase becomes the most expensive when executing the simulator on the GPU (47%). Our novel implementation, *ph2-simorder-dyncomp*, is close (time-wise) to the sequential implementation. Indeed, as mentioned above, this phase

	Test A (mean LHS length 1.5)			Test B (mean LHS length 3)		
	% CPU	% GPU	Speedup	% CPU	% GPU	Speedup
Phase 1	53.7%	30.1%	14.23x	55.3%	12%	8.52x
Phase 2	12.6%	47%	2.13x	18.4%	82.8%	0.4x
Phase 3	22.6%	13.7%	13.2x	14%	2.2%	11.72x
Phase 4	11.1%	9.2%	9.7x	12.3%	3%	7.43x

Table 7.3: Profiling the simulators for GPU and 1 core CPU.

is the most challenging to parallelize. Special efforts have to be considered here. On the other side, Phases 3 and 4 are relatively lightweight and so, successfully accelerated (up to 13x and 9.7x, respectively). Hence, our library random binomial generation based on CuRAND is well suited for Phase 3.

Finally, as shown in Figure 7.8, the performance of *pdp-gpu-sim* decreases as the mean LHS length is increased. For Test B, the overall speedup decreases from 7.9x (Test A) to 1.8x (Test B). The percentage of time consumed by Phase 2 is dramatically increased for the GPU, taking up to 83%. Thus, the competition degree of rule blocks is a limiting factor in performance, which fully correlates with the achieved results.

In conclusion, Phases 1, 3 and 4 were efficiently executed on the GPU, however Phase 2 was poorly accelerated, since it is inherently sequential.

7.6 Conclusions

In this chapter we have introduced the developed implementations for the two simulation algorithms for PDP systems, DNDP and DCBA. DNDP has been implemented within the *pLinguaCore* simulation framework. DCBA has been implemented inside *pLinguaCore*, and also in a stand-alone framework called ABCD-GPU (a subproject of PMCGPU [17]).

- DNDP implementation: it is made in Java, within *pLinguaCore*. Java threads are used to speedup the simulation. Each environment of the simulated PDP system is represented by a thread. However, results indicate that Java thread are not achieving the expected speedup.

- DCBA implementation: the main challenge for the implementation of DCBA is the representation of the distribution table, since it is sparse (a majority of null values) and can be large.

- pLinguaCore: a hash table is utilized for the distribution table, saving memory and time in the simulation process. However, performance is degraded since the chosen platform is in Java and inside *pLinguaCore*, and efficiency can not be totally controlled.
- ABCD-GPU: the table is avoided through a solution called virtual table. The operations over the table are transformed to operations over the rule blocks information, but using some more iterations. This solution has been implemented using C++ and static memory representation. Two parallel platforms have been also used to accelerate the simulation:
  - Multicore CPUs and OpenMP: the different simulations performed to the same model, and the environments of the model, are utilized to distribute parallelism over the cores of a CPU. Experiments show that the simulation time is half than the sequential version using four cores.
  - Manycore GPUs and CUDA: again, the different simulations and environments are used to distribute work along the two-dimensional grid of CUDA thread blocks. Threads work with individual rule blocks in each phase of the DCBA. The challenge of implementing binomial random variates on the GPU has been solved through simple approximations with normals. However, phase 2 is inherently sequential, and becomes the main bottleneck of the GPU simulator. Experiments show that the performance can be improved up to seven times, but depends on the simulated PDP system.

The provided platform for PDP systems is flexible (can handle the full class of PDP systems), but performance and scalability is degraded. However, the achieved performance is enough for running many sizes of models that we can encounter today. Nevertheless, the topology of the simulated model affects directly to the performance. Moreover, experiments show again that P systems simulations (this time, PDP systems) are memory bound: more cores does not improve performance as expected.

Part IV

Thesis Results

"The visions we offer our children shape the future. It matters what those visions are. Often they become self-fulfilling prophecies. Dreams are maps."

Carl Sagan (1997)

8 Conclusions

This chapter puts an end to the document by summarizing and analyzing the presented work. Firstly, we provide a summary of the whole document, chapter by chapter, highlighting the main contributions. Second, the main results obtained during the work are listed and analyzed. Third, we show overall conclusions taken from the main results. Fourth, we elaborate a list of guidelines to efficiently simulate P systems on the GPU. Finally, we analyze future works and open research lines.

8.1 Summary

P systems are computational devices defined in the area of Membrane Computing. They are based on the abstraction of the compartmentalized structure and parallel processing of biochemical information in biological cells (Chapter 1). However, they are yet to be fully implemented *in vivo*, *in vitro*, or even *in silico*, because of their massively parallel, distributed, and non-deterministic nature. Thus, practical computations of P systems are driven by silicon-based simulators, even though their potential results are compromised by the physical limitations of silicon architectures. They are often inefficient or not suitable when dealing with some P system features, such as the exponential workspace creation, non-determinism and massive parallelism.

Silicon-based simulators for Membrane Computing have traditionally been implemented on latency-oriented CPUs architectures (Chapter 2), which lack the possibility of exploiting their massively parallel nature. In order to improve their efficiency, it is necessary to exploit current technologies, leading to solutions in the area of High Performance Computing (HPC), such as accelerators or many-core processors (Chapter 3). In this respect, Graphics Processing Units (GPUs) have been consolidated as accelerators thanks to their throughput-oriented highly-parallel architecture. Therefore, they are good candidates for decreasing the gap between the practical and theoretical computations of P systems as they have been so on other massively parallel applications.

The aim of this work is to analyze GPUs as parallel platforms to accelerate the simulation of Membrane Computing models. For this purpose it is necessary to define new simulation algorithms for P systems, what allows us to better reproduce the semantics of the models. Moreover, they have to be efficiently implemented on GPUs. The GPUs utilized for the development and experimentation processes are of the NVIDIA brand. The simulators are then implemented using the CUDA programming model, and the specific GPU used for the experiments is the NVIDIA Tesla C1060 (which contains 240 cores and 4 GB of memory). This thesis presents the first attempts simulating P systems on CUDA. The document was divided in four parts: preliminaries (three chapters), parallel simulation applied to efficient solutions of computationally hard problems (two chapters), parallel simulation applied to computational models in biology (two chapters), and conclusions (one chapter).

Chapter 1 provided a detailed introduction to the interdisciplinary fields of Bioinspired and Natural Computing. The introduction was specially focused on Membrane Computing. The models defined in the area (P systems) and their major features were depicted. Two P systems models were then described: cell-like P systems with active membranes and tissue-like P systems with cell division. Moreover, two linear-time solutions to SAT problem were given using these P systems variants. The chapter ended highlighting the necessity of simulators for P systems.

This led us to Chapter 2, which outlined the state of the art in simulating P systems. A discussion on the simulation development process was provided, together with a list of the most important software tools available to date. The second chapter also focused on the simulation framework P-Lingua, and the Java library pLinguaCore, which is the seed of our work. Another discussion on the necessity of accelerating the simulation was given. An analysis of the parallel simulation of P systems followed this discussion, and it finished with a summary of the existing parallel simulators.

The introduction block concluded with Chapter 3, that introduced the corresponding concepts of High Performance Computing and Parallel Computing. They constitute the base of the development process for parallel simulators in our work. This chapter made special attention to accelerators, specially the GPU. An introduction of GPU computing was provided, focused on CUDA and NVIDIA GPUs architectures.

Chapter 4 was the starting point of the presentation of the developed work. The first simulator implemented on CUDA was presented here. It is designed for recognizer P systems with active membranes. The simulation algorithm used for the simulator is the same than the existing in the pLinguaCore library. Moreover, pLinguaCore constructs the input of the parallel simulator; that is, translate a P-Lingua file into a parsed binary file. The performance of the created simulators was also analyzed by two case studies: one based on a very simple test P system, and the other regarding the solution to SAT problem with active membranes. The experiments showed that the first case study reports better speedups than the second. A characterization of the simulators was also provided to conclude the chapter.

Chapter 5 introduced the next natural step taken in our work, which is the development of *ad-hoc* simulators for the family of P systems solving the SAT problem. The first simulator was created for the solution with active membranes. Although it is a less flexible simulator than the one described in Chapter 4, the achieved acceleration with the GPU is multiplied. This simulator has been also improved by better adapting to new GPU architectures, multi-GPU systems and to supercomputers. Finally, a simulator for the solution based on tissue-like P systems was also implemented, and compared with the one based on cell-like P systems. The experiments showed that the cell-like based simulator is more efficient than the tissue-like P systems, and this property was analyzed at the end of the chapter.

Chapter 6 started the description of the work on the simulation of Population Dynamics P systems (PDP systems). It described the variant of PDP systems, and the applications in real ecosystems models. After discussing the main features of simulation algorithms for this variant, the first simulation algorithm, BBB (Binomial Block Based), was introduced. Then, the two new simulation algorithms contributed in our work, called DNDP (Direct Non-Deterministic distribution with Probabilities) and DCBA (Direct distribution based on Consistent Blocks Algorithm), were presented. They were also analyzed by simple test case studies, and finally, the DCBA and DNDP were experimentally validated by a real ecosystem model.

Chapter 7 finished the description of the work by presenting the simulators implemented for PDP systems. The DNDP and DCBA were implemented in several platforms. First, a version was included in the pLinguaCore library, as alternative inference engines for PDP systems. After that, the DCBA was implemented in a stand-alone simulator based in C. From this starting point, versions in OpenMP and CUDA were developed. The main challenges of the implementations in the parallel platforms were explained, such as the creation of binomial random-variate generation on the GPU. The carried out experiments reported better acceleration using the GPU than a multicore platform. However, more improvements can be made to the corresponding simulators.

8.2 Results

In this section we will list the main results obtained in the work. The results we are reporting are the developed simulators, designed simulation algorithms, and other transversal results.

8.2.1 Parallel simulators

Next, we show all the parallel simulators we have developed in this work. These simulators are included in the project PMCGPU [17], and available under the GNU GPLv3 software license. Each simulator conforms a sub-project itself within PMCGPU.

P systems with active membranes on CUDA

The first project initiated was PCUDA. It is a C++/CUDA based simulation framework for the class of recognizer P systems with active membranes (see Chapter 4). The simulators are based on the simulation algorithm used for the active membranes engine in pLinguaCore. It performs only one computation in order to avoid non-determinism, so it is interesting for confluent P systems. Thus, the simulation algorithm is mainly a loop over the transition steps, that reproduces one computation path of the tree. We can take advantage of this property by selecting a low-cost path for the simulator. This cost is measured in terms of membranes number and communication.

Each transition is carried out in the simulators by two stages: selection and execution. Selection is the most time-consuming stage, and choose the rules that can be applied for a given P system configuration, together with the number of times to apply each one. The rules that are low-cost for the simulator are chosen here. The information from this stage is used for the next one, which is the execution of them; that is, updating the P system configuration. The division in two stages helps to synchronize the application of rules within and among membranes.

The CUDA simulator takes advantage of the double-parallel nature of GPUs to speedup the simulation of the double-parallel nature of P systems. Each CUDA thread block is assigned to each elementary membrane, and each thread is assigned to a portion of the objects defined in the alphabet. It is a naive solution to the problem of representing the P system in CUDA. The CUDA simulator assumes by default that all the defined objects can be placed within each membrane, allocating memory space for all of them. Although this is the worst case that the simulator has to deal with, it does not take place in the majority of P systems to be simulated. Thus, the performance of the simulator completely depends on the simulated P system.

Two case studies has been used to test the performance: a simple test P system designed to stress the simulator (A), and other regarding the solution to SAT problem with active membranes (B). The experiments report up to 7x of speedup using the case study A (for 512 objects and 1024 membranes), and 1.67x for case study B (SAT(12,17), implying 914 objects and 4096 membranes). We can state that the first case study reports better speedups than the second. Keeping in mind the results, we have identified three indicators that affect performance: density of objects per membrane, rule intensity and communication among membranes.

In conclusion, PCUDA offers a parallel framework to simulate P systems with active membranes. The simulators are highly flexible, but has low performance and low scalability.

Family solving SAT with active membranes on CUDA

The project PCUDASAT is a continuation of PCUDA. It aims to develop optimized simulators for a family of P systems with active membranes solving SAT in linear time (see Chapter 5). A sequential and two parallel (CUDA based) simulators are included. Their simulation algorithm is based on the stages that can be identified in the computation of any P system in the family: generation, synchronization, check-out and output. The code is tailored to them, saving space in the definition of, for example, auxiliary objects.

The simulators receive as input a DIMACS CNF file, which codifies an instance of SAT through a CNF formula. The output is a summary of the codification, and the answer: is `yes` or `no`. Therefore, they merely act as a SAT solver based on a P systems based solution.

The CUDA simulator design is similar to the one used in PCUDA: it assigns a thread block to each elementary membrane (which encodes a truth assignment to the CNF formula). However, the number of objects to be represented inside each membrane in memory has been decreased. In this case, it is enough to store only the objects appearing in the input multiset (which is a literal of the CNF formula), so that the rules can evolve them, one by one. Therefore, the CUDA design assigns a thread to each object of the input multiset. A second CUDA simulator was developed to improve the usage of GPU resources. It is a hybrid solution, inasmuch as it reproduces the execution of the first stages, but the last ones are more efficiently executed on the GPU.

The experiments carried out report up to 63x of speedup for the CUDA simulator against the sequential, considering data management (with 256 objects and 4 M membranes). Furthermore, the hybrid CUDA simulator outperforms the CUDA simulator in 9.3x.

Although the PCUDASAT simulators are less flexible, the performance and scalability has been much increased compared to PCUDA simulators, for this special case study.

It is noteworthy to mention further simulators developed to be better tailored to the GPU idiosyncrasy. Better efficient strategies were performed, and newer GPU cards, multi-GPU and supercomputers were utilized.

Family solving SAT with tissue-like on CUDA

A correlated project to PCUDASAT is TSPCUDASAT. Its objective is to simulate a family of tissue-like P systems with cell division solving SAT in linear time (see Chapter 5). The simulation algorithm is based, as in PCUDASAT, in the 5 stages of the computation of the P systems: generation, exchange, synchronization, check-in and output.

The CUDA simulator design is similar to the one used in PCUDASAT. Each thread block is assigned to each cell labeled by 2. However, the number of objects to be placed inside each cell in the memory representation is increased. The simulator requires to store $2n + 4 + |cod(\varphi)|$ objects per cell, being n the number of variables in the CNF formula, and $|cod(\varphi)|$ the size of the input multiset encoding the input formula φ. That is, it requires $2n+4$ more objects per cell than the PCUDASAT simulators, but it does not need to store charges. Threads are used differently in each stage, maximizing their usage in each case.

Experiments show that the CUDA simulator outperforms the sequential one by 10x (for 256 objects and 4 M cells). It can be seen that solving the same problem (SAT) under different P system variants leads to different speedups using CUDA. Indeed, we show that the usage of charges can help to save space devoted to objects.

DCBA on OpenMP

After defining a new simulation algorithm for PDP systems, called DCBA, the challenge was to efficiently implement it somehow. The first step was an implementation within pLinguaCore. However, the simulation was slow. Then, the solution was to develop a version in C++ version, in a stand-alone simulator. This projects was called ABCD, but now is ABCD-GPU (see Chapter 7).

The implementation in C++ reproduces the stages of DCBA, selection and execution, plus the three micro-stages for selection: phase 1 (distribution), phase 2 (maximality) and phase 3 (probability). The main problem arises when simulating large PDP system models. The static table required in selection phase 1 is too large, and also it is sparse. Therefore, ABCD-GPU simulators save on memory by avoiding the creation of a static table. It is carried out by translating the operations over the table to operations directly to the rule blocks information.

This C++ implementation is parallelized in three ways: 1) simulations, 2) environments and 3) a hybrid approach. All of them are implemented using the parallel standard library for multicore platforms, OpenMP. The experiments were ran on two multi-core processors: the Intel i5 Nehalem and i7 Sandy Bridge. We achieve runtime gains of up to 2.5x by using all the cores of a single socket 4-core Intel i7.

Experiments indicate the simulations are memory bound and the portion of the code we parallelized consumes over 98% of the runtime in serial. From this initial work, we conclude that parallelizing by simulations or hybrid techniques yields the largest speedups. Also, using hardware that has more memory bandwidth is an easy way for scientists to improve the speed of our simulator. It can also be concluded that performance tuning to decrease data movement is important for P-system simulators.

The simulator is flexible and scalable enough to run a wide range of PDP systems, but the performance is still low.

DCBA on CUDA

The major expansion of the ABCD-GPU project came with the creation of a CUDA simulator (see Chapter 7). The simulation algorithm is the DCBA, together with the optimizations made in the previous simulators in ABCD-GPU. Moreover, the memory utilization was improved (saving up to 27% of memory) for both the GPU-based version and the previous OpenMP-based version.

The CUDA design for the GPU part of the simulator is as follows: environments and simulations are distributed through thread blocks, and rule blocks among threads. Phases 1, 3 and 4 were efficiently executed on the GPU, however Phase 2 was poorly accelerated, since it is inherently sequential. Furthermore, phase 3 requires the creation of random binomial variate generation. For this purpose, a new CUDA library was developed, called cuRNG_BINOMIAL. It uses the normal approximation for large parameters values, and the BINV algorithm for low ones. The cuRAND library was utilized to generate the required uniform and normal random numbers.

We benchmarked a set of randomly generated PDP systems (without biological meaning), achieving speedups of up to 7x for large sizes (running 50 simulations, 20 environments and more than 20000 rule blocks) on NVIDIA Tesla C1060 GPU over the multi-core version.

8.2.2 Simulation algorithms

One of the assets of PDP systems framework is the ability to conduct the simultaneous evolution of a high number of species, as well as the management of a large number of auxiliary objects. Moreover, the compartmentalized structure, both as a directed graph (environments) and as a rooted tree (membranes), allows to differentiate multiple geographical areas.

The development of efficient algorithms able to of capturing the semantics described by the framework is a challenging task. These algorithms should select rules in the models according to their associated probabilities, while keeping the maximal parallelism semantics of P systems. In this scenario, the concept of *rule blocks* arises. A rule block is a set of rules sharing the same left-hand side. On each step of computation one or more blocks are selected, according to the semantics associated with the modeling framework. For every selected block, its rules are applied a number of times in a probabilistic manner according to their associated probabilities, also known as *local probabilities*.

The way in which the blocks and rules in the model are selected depends on the specific simulation algorithm employed. These algorithms should be able to deal with issues such as the possible competition of blocks and rules for objects. So far, several algorithms have been developed in order to capture the semantics defined by the modeling framework. One of these algorithms is the Binomial Block Based algorithm (BBB).

The BBB algorithm has a drawback, regarding a distorted selection of blocks and rules. Indeed, instead of blocks and rules being selected according to their probabilities in a uniform manner, the selection process is biased towards a random choice imposed to blocks.

Direct Non-Deterministic distribution with Probabilities (DNDP)

In order to solve the drawback of the BBB algorithm, we have inspired on the DND algorithm [116] to define a new simulation algorithm, called DNDP, performing object distribution, maximal consistency and calculation of probability functions (see Chapter 6).

The DNDP algorithm is divided in two phases, one for selection and other for execution of rules. The first one is also divided in other two micro phases, a first selection phase which generates a consistent multiset of applicable rules, and a second selection phase that changes the previous multiset for maximal application. We have analyzed that this algorithm partially solves the drawbacks presented in BBB.

Direct distribution based on Consistent blocks Algorithm (DCBA)

Although the DNDP algorithm is able to refine the behavior of the variant, its selection process is still biased towards those rules with highest probabilities. Therefore, a deterministic behavior for blocks of rules competing for the same resources is not defined. Then, a new approach was introduced, which is called the Direct distribution based on Consistent Blocks Algorithm (DCBA). It is a simulation algorithm which addresses that inherent non-determinism of the variant by proportionally distributing the resources (see Chapter 6).

This new algorithm performs an object distribution along the rules that eventually compete for objects. The main procedure is divided into two stages: selection and execution. Selection stage is also divided into three micro-phases: phase 1 (distribution), where by using a table and the construction of rule blocks, the distribution process takes place; phase 2 (maximality), where a random order is applied to the remaining rule blocks in order to assure the maximality condition; and phase 3 (probability), where the number of application of rule blocks is translated to application of rules by using random numbers respecting the probabilities.

The DNDP and DCBA are experimentally validated towards a real ecosystem model related to the Bearded Vulture in the Pyrenees (NE Spain), showing that they reproduce similar results as the original simulator written in C++ and actual data collected for 14 years.

8.2.3 pLinguaCore simulators

The two introduced simulation algorithms for PDP systems, DNDP and DCBA, have been implemented inside the pLinguaCore library. They are part of the new features of pLinguaCore version 3.0.

DNDP on pLinguaCore

Many simulation algorithms are defined inside pLinguaCore for the different P system models. Initially, one of the implemented simulation algorithms for PDP systems is the BBB. In order to provide improved approaches, we have included DNDP algorithm as well (see Chapter 6).

The key aspect of this implementation is the provided parallel solution. The process of selecting and executing rules is carried out in parallel for each environment without compromising the concurrency when accessing to common objects. However, the selection and execution stages have been globally synchronized, so that the execution phase cannot start before finishing the selection in every environment.

Experiments show that using the Java Virtual Machine for executing the DNDP algorithm in parallel is not efficient enough (1.72x). The speedup is not as much as expected when simulating 16 environments over a 4-core processor. Although being powerful enough for simulating most of the defined PDP systems based models, the new models under development requires a huge amount of resources, much higher than the resources in a personal computer using Java.

DCBA on pLinguaCore

PLinguaCore has been also upgraded to provide an implementation of the DBCA, thus extending its existing probabilistic model simulation algorithms support (see Chapter 7). This implementation aimed to validate the DCBA against small PDP system models. Indeed, after validating the algorithm towards a real ecosystem model related to the Bearded Vulture in the Pyrenees (NE Spain), the DCBA implementation in pLinguaCore is used to run small models. In the near future, this implementation will be changed for a wrapper that executes our C++/CUDA version.

However, we have made efforts to develop an implementation as efficient as possible. In this respect, the key optimization was focused on phase 1, where the static and dynamic tables are used. As mentioned before, these tables are very large for large models, and they are sparse (storing a majority of null values). Therefore, the implementation uses a system based on hash tables to store only the non-null values.

8.2.4 Computational resources and software licensing

The webpage `http://sourceforge.net/p/pmcgpu` provides technical information and documentation about the developed simulators, together with all the related source code of them. Every software application and library here presented is available for the scientific community under the free software license GNU GPL [2].

Additionally, the development of the work presented in this thesis has required the setting up of a GPU server. It has been installed inside a room prepared for that purpose. It has been configured with the operating system Linux Ubuntu server edition 10.04, and it has been expanded with 2 GPUs NVIDIA Tesla C1060. Moreover, the system has been provided of a job queue manager to let many users use the GPUs. One GPU has been devoted for fast testing and debugging, and the other for time and profiling analysis (see Appendix B).

8.3 Conclusions

P systems are an alternative approach to model biological phenomena in the field of computational Systems Biology and Population Dynamics based on the functioning of living cells, providing massively parallel and compartmentalized models. However, one needs efficient simulators in order to experimentally validate the models.

GPUs are being established as a massively parallel processor where programmers can accelerate scientific applications. They are good alternative to conventional CPUs to simulate membrane systems due to the double parallel nature that both GPUs and P systems present. Their shared memory system also helps to efficiently synchronize the simulation of the models.

Using the power and parallelism provided by GPUs to simulate P systems is a new concept in the development of applications for Membrane Computing. GPU features can help researches to accelerate their simulations by using a cheap and scalable parallel architecture. Moreover, GPU computing is broad enough to adapt the simulators to specific P system models. For example, P systems exploiting exponentially their workspace easily reach GPU memory limitations, but they can be scaled to multi-GPU systems or GPU clusters. P systems with probabilistic behavior can be also simulated on the

GPU without losing performance. Libraries such as cuRAND, and the new cuRNG_BINOMIAL, can calculate random numbers in a high rate.

However, our results show that P systems simulations are memory bound: the performance of the simulation is relatively low compared with the resources available in the GPU. The main causes are that simulating P systems requires a high synchronization degree (e.g. the global clock of the models, rule cooperation, rules competition, etc.), and the number of instructions to execute per memory portion is small. These restrict the design of parallel simulators. A parallel simulators designer has to be careful with the representation and management of each P system ingredient. A bad step taken on GPU programming can easily break parallelism, and so, performance. We refer to the guidelines provided at Section 8.4.

Recall that having flexible simulators affects to performance. Flexible parallel simulators are intended to take advantage of P systems parallelism. If the design of the P systems to simulate does not have a large degree of parallelism, the performance is, in fact, dramatically downgraded. Therefore, when working with highly flexible simulators, the P systems design has to be reconsidered to achieve performance, in such a way that they execute as many rules as possible in each computation step. On the contrary, the less flexible the simulators are, the more optimizations can be performed. Thus, *ad-hoc* simulators can run more efficiently on GPUs.

Last, but not least, it is necessary to define new simulation algorithms that better fit the expected behavior of P systems computations. For PDP systems, it is very important to reach this, so that the real ecosystems models can better reproduce the population dynamic processes. Furthermore, simulation algorithms should also be defined taking into account the performance, so that they can be easily implemented on parallel platforms.

We can finally conclude that the advent of the accelerators in High Performance Computing offers fresh avenues for developing new and efficient simulators for P systems.

8.4 Guidelines for designing parallel simulators

In this section we collect some guidelines from our experiences concerning the design of parallel simulators for P systems, specially, using GPUs. We will consider many P systems ingredients and explain how they can be handled on the design of the simulator.

8.4.1 Work assignation

Threads and thread blocks

The work assignation to CUDA thread blocks and threads is a critical design decision. It will drive the design and the development process, and the data representation may differentiate. This assignation is the way to really implement parallelism on CUDA.

We highlight the following heuristics to carry out the work assignation:

- *Thread blocks*: they may be assigned to independent blocks of information related to the purpose of the kernel. These blocks should not communicate among them. For example, elementary membranes for the selection stage of P systems with active membranes, or environments for the selection of PDP systems. The number of blocks should be huge (e.g. greater than 100) to fill all the SMs in the GPU. Recall that one or more thread blocks are assigned to each SM.

- *Threads*: they should be assigned to small information units within the corresponding block that can be executed in parallel at a certain moment. They can cooperate and communicate because threads can be easily synchronized. For example, rules for both P systems with active membranes and PDP systems. The number of these units should be below 512. If not, they can be distributed along the threads. The operations over these units have to be uniform; that is, they have to be as similar as possible. If they differentiate in some point, they must have the possibility to resynchronize as soon as possible. The number of threads should be small, between 64 and 256, and they should not be too much overloaded.

Multi-GPU systems

If the representation size of a P system model is too large for one GPU, a multi-GPU system should be used. First, one will need to design a work assignation, in such a way that the number of thread blocks can be distributed along the GPUs. Another approach could be to identify a higher level of parallelism, where each GPU can take care.

After that, it is required to define a master process managing the execution on each GPU. The communication among GPUs should be avoided (to save memory transactions). However, the new technology GPUDirect of CUDA 5 can speedup that communication.

8.4.2 Ingredients representation

Multisets of objects

The representation of object multisets should be carefully designed. The way they are indexed affects to the overall performance (how the threads access the information), and to the space required to store the information of the full system. Work assignations where threads are assigned to objects are more sensitive to this. A high number of represented objects per membrane can lead to more amount of inactive threads. Here is where the term density of objects per membrane becomes important.

There are two major strategies for representing multisets of objects:

Strategy A. *Static memory approach*: the representation provides enough space for an upper bound of number of objects. This solution will waste space for non-appearing objects, but it gives the opportunity to implement fast accesses by using known indexes.

Our naive representation in flexible simulators was to allocate space for the whole alphabet of objects inside each membrane. Although there are empty spaces, the objects are accessed directly through their IDs. For *ad-hoc* simulators, we could decrease the total size through a known upper bound, and the objects are also easily accessed. By doing this, we were able to increase the density of objects per membrane (not increasing the appearing objects, but decreasing the total considered amount).

Strategy B. *Dynamic memory approach*: objects are dynamically allocated during the simulation. Although there exists dynamic memory support in recent GPUs (Fermi architecture and later), this can be implemented by a very large array and hash based algorithms. This solution will require to implement a searching strategy, dynamic reallocation of objects, empty positions management, etc. Also note that hash algorithms are still inefficient to execute on GPUs. Therefore, we do not recommend this strategy when implementing on GPUs, neither in sequential platforms (dynamic memory has been shown in our experiments to be more slow than using static memory, see Chapter 4).

Charges

P systems with charges have several disjunct sets of rules (one set per each polarization) that are applicable at a certain step in the computation. Thus, the number of defined rules is bigger than the number of selectable and applicable rules. In this case, these sets of rules should be easily and efficiently discriminated during the selection process. However, it leads to loose space when storing the output of selection phases; in other words, there are many empty positions related to non-selected rules because of charges. If there are three polarizations, and the rules are in of the same proportion depending on each polarization, then the selectable rules are $\frac{1}{3}$ of the total.

If this problem negatively impacts performance, then the elimination of the auxiliary data structure for selected rules should be considered, such as not using selection and execution phases (see the Synchronization subsection). It is here where the importance of the term rule intensity arises. If the threads are assigned to rules, the more applicable rules, the better it is accelerated.

Nevertheless, the usage of charges has a semantic meaning for the computation in some specific models (e.g. in SAT solutions, the charges store the truth assignment). P systems without charges store that information in extra objects, what affects to the space in memory: this increases the upper bound of number of objects imposed in the representation. In *ad-hoc* simulators, charges can help to simplify the representation (if they are semantically used in the model).

Membrane structure

When we have assigned membranes to thread blocks, we have considered only two levels in the membrane hierarchy. The memory accesses made by threads can be hazard: membranes (thread blocks) accessing to objects in their parents can concurrently interact with other threads belonging to a different thread block, without any protection. A solution to this problem could be to identify the levels and assign thread blocks only to membranes of the same level. Thus, each level is treated in consecutive synchronized calls to kernels.

If the skin membrane is executed separately, the term of communication among membranes becomes important, since it will imply to move data between CPU and GPU. However, if the work assignation assumes that each thread block is assigned to a whole P system (e.g. environments), the problem is solved automatically inasmuch as threads inside blocks can be easily synchronized.

Rule cooperation degree

P systems having a rule cooperation degree greater than one can be hazardous to simulate. They can also lead to object competitions. Rules having more than one object in the left-hand side require to calculate the minimum number of applications available per object. This can be carried out by to strategies:

Strategy A. *Rules (threads) analyze their objects:* threads will loop the objects appearing in the left-hand side, calculating the minimum number along objects. The main problem is that it can cause thread diversification, breaking parallelism inside warps (must be synchronized after this process to reconverge). This strategy can be also tricky to manage when rule competition takes place: the execution can fall into a dead lock.

Strategy B. *Objects (threads) annotate "their availability" to execute the rule:* two steps have to be taken, which are objects annotation, synchronize, minimum calculation. This strategy works better when there is rule competition, and when the cooperation degree is very high, so the loop per thread implicit in strategy A is too slow.

Division rules

The simulator needs to allocate the space for membranes before starting, since the CUDA memory management is based on static memory. We can use again an strategy based on dynamic memory, but as mentioned before, it may be detrimental to performance.

We differentiate two cases and the strategy to follow in each one:

Case A. *We don't know the maximum number of membranes to be created*: in this case the simulator has to allocate as much memory as possible. Moreover it has to annotate the position of the next empty place for membranes. This can be done through a global variable (accessed by atomic operations). The simulator should also launch an error if more membranes are created than the available space.

Case B. *We know the number of membranes to be created*: the simulator should allocate that specific space. In order to avoid the global variable annotating the next empty position, a binary incrementation can be used (if possible). That is, the position of a membrane to create depends directly on the membrane that has issued the division (e.g. $new\_membrane = membrane + total\_existing\_membranes$).

Probabilities

Probabilities associated with rules can be easily represented on the simulator by using an extra information stored apart (e.g. an array). Depending on the nature of the random numbers to generate, the cuRAND and cuRNG_BINARY libraries can help. Threads should be creating the random numbers at the same time to achieve better speedup. Alternatively, the cuRAND has the option of previously generating the random numbers, storing them on the GPU memory.

This can be used only if there is no problem with memory space (depending on the P system representation), and the number of random numbers to generate is not too large. Moreover, CUDA pinned memory can help to asynchronously copy the random numbers from the CPU to the GPU.

8.4.3 Simulation algorithms

Non-determinism

In our simulators we have simplified the management of non-determinism by considering confluent P systems. Otherwise, the whole computation path should have to be analyzed. Two main strategies can be followed:

Strategy A. *Depth-first approach:* only one path is simulated at a time. It requires to annotate the selections made during the path, making possible to going back, and also to select new choices.

Strategy B. *Breadth-first approach:* all the paths are simulated at the same time. Every choice has to be taken, what requires memory for all of them. However, the simulation of every path can be considered as a new level of parallelism.

Rule competition degree

The semantics of the model define how rules compete for common objects. Therefore, the definition of a simulation algorithm capturing this semantics is required, together with a validation process. In our case, we have made two approaches called DNDP and DCBA. We have use a table to uniformly distribute common resources along the rules.

Synchronization

Since every P system has a global clock, the simulator requires synchronizing every transition step (unless for asynchronous P systems). However, our simulation algorithms requires an extra synchronization step regarding two stages, selection and execution: the creation and deletion of objects may not interfere.

A way to avoid this synchronization step is the creation of two copies of P system configuration: one where adding objects, and another where deleting them. It implies an extra step after the selection/execution of rules where adding the remaining objects in the deletion array to the other.

8.5 Future work

This thesis presents the first attempts on simulating P systems using GPUs. Therefore, the work opens new research lines to continue and expand our initial goal: implementing P systems parallelism on High Performance platforms.

Connecting simulators with simulation libraries

In order to effectively use the developed simulators, it is necessary to reconnect them with a more general simulation framework, such as P-Lingua. It is widely used along the Membrane Computing community. Therefore, the parallel simulators can be transparently used by P-Lingua end users, decreasing their time to work with those P systems models, without any knowledge of parallel programming. This opens an important research line regarding the efficient communication between general simulation frameworks and efficient simulators over parallel platforms.

For this purpose, we will require:

- New file formats becoming the input of the CUDA simulators. They should be minimalist and self-contained, providing an efficient language of communication. That way, the same P-Lingua files can contain the input data for the Java simulators inside pLinguaCore and the CUDA simulators. This can help, for example, to validate current PDP systems simulators with real ecosystems models.

- A communication protocol between pLinguaCore and the CUDA simulators for automatically and efficiently connecting them. It should be efficient, fault tolerant, and transparent to end users. Concerning the PDP systems modeling framework, the performance of pLinguaCore can be enhanced by replacing the simulation core through a wrapper executing CUDA simulators. Then, the usage of this simulators would be transparent to end users of both pLinguaCore and MeCoSim platforms.

Simulating other models

Given that P systems simulations are computationally expensive and demanding on memory resources, simulators on CUDA can be developed to speed up the processes by following similar techniques to those described along this thesis. In fact, we have envisioned a starting point for a significant number of applications to benefit from our GPU acceleration methods in the near future.

The concepts and directives utilized here for the design and development of parallel P systems simulators can be applied for other models. However, a previous study of the specific model usage should be made to justify the need of parallel simulators. Some models having the efficiency as an important factor are: multi-compartmental stochastic P systems (having real applications in the field of Systems Biology), spiking neural P systems, tissue-like P systems with cell separation, kernel P systems, etc. A high-performance implementation of those simulation models looks promising on GPUs and we have provided in this thesis some guidelines to succeed by using CUDA.

Additionally, they can be also used for other bioinspired models of computation different from those within Membrane Computing. The key is to study how to adapt the semantics of the models to the GPU architecture.

Improving existing simulators

Further improvements to current parallel simulators can be performed. These simulators are limited by the available resources on the GPU as well as the CPU (RAM, Device Memory, CPU, GPU). In following versions, memory requirements have to be reduced to better utilize the resources of the GPU. There are some techniques that can be studied for this purpose, such as sparse matrices.

Moreover, the implementations can be also tailored to the GPU architecture. Computational primitives, such as scatter/gather, map, reduce and scan, are suitable for this, as seen in many other CUDA accelerated applications.

In addition, the simulators should be extended to remove the limitations of each one. For example, more than two levels in the membrane hierarchy for active membranes can be supported. These expansions should be made without repercussion to the overall performance.

At a higher abstraction level, it would be also interesting to adopt model-oriented heuristics to improve the CUDA design. If the simulators performs a previous analysis of the defined rules, before starting the simulation, more extra information can be provided. This information can be used to select built-in strategies that are better fitted to GPUs, avoiding extra computation or extra memory devoted for situations that may not appear in a specific input P system model. Furthermore, the P-Lingua programming language can be also extended to support parallel directives that the P systems designer can provide to the simulator. For instance, different modules of rules that are executed separately in different moments of the computation.

Improving simulation algorithms

The simulation algorithms utilized to reproduce the semantics of the models can be also improved. We have presented several approaches, such as DNDP and DCBA, but further research can be made in this respect. The enhancements can be twofold:

- Improving behavior: the simulation algorithms for PDP systems, and by extension, of other P systems models, can be refined to better reproduce the expected results from them.
- Improving performance: the simulation algorithms can be better adapted to take advantage of parallel architectures from their initial definition. For example, linear algebra based algorithms can be used to better study the semantics, and can be also efficiently implemented on GPUs.

In this respect, designed simulation algorithms, such as DND [116] and "maximum applicability" [63], can be subject of study. It would be interesting to implement and improve them on the GPU architecture. Although they are focused on transition P systems, they may be extended to other models with similar and extra ingredients.

Another challenge worth to consider concerns the simulation of the maximally parallel application of rules. Maximality is assumed to be done in parallel in the formal model. However, it is simulated through inherently sequential algorithms (as seen in phase 2 of DCBA), so the challenge is to implement actual parallel simulation of maximality on the GPU, with low microsteps and low synchronization.

Using new parallel technology

In our experiments, we have used the NVIDIA Tesla C1060 GPU. It has 240 cores, 4 GBytes, and supports compute capability 1.3. The release date of this GPU was 2008. The newest generation of many-core GPU architectures, such as NVIDIA Kepler (1600 cores), are enhanced by using more cores, more memory and on-chip L2 cache. The peak-performance is increased also for single and double precision operations. Furthermore, the new GPUs of AMD are bursting in the field of GPGPU (providing up to 1800 cores per card). They are programmed in OpenCL, and are harvesting the best peak performance in the GPU computing arena. It would be very interesting to study their behavior to optimize resources in the simulation of P systems.

Moreover, the combination of grid computing, cloud computing and heterogeneous systems can be an alternative for increasing the memory size without sacrificing performance at all. It can pursue a solution to the main problem simulating P systems: memory restrictions. Note that today best supercomputers are based on GPUs (see top 500 website [18]), so the most highly scalable solutions are available for P systems simulations.

Studying models for parallel simulators

Further research can also be carried out concerning the parallel simulation of particular P systems features. We have stated during the thesis, specially in Section 8.4, that there are some P system ingredients well suited to be simulated on GPU (in terms of achieved acceleration). We think that a more in-depth analysis of these ingredients should be considered, identifying which of them can be easily combined and well simulated by the GPU. In this way, novel approximations for parallel simulators development can be carried also at the P systems area. For example, the parallelism inside models should be measured for flexible simulators, since they assume that the model is parallel to achieve better performance. This can be performed through Sevilla Carpets, that have been used as an indicator of the parallelism degree inside models [79].

An approach can be to define a P system model combining all the good features for GPU simulators (let call it *GP systems*, or GPU oriented P systems). Then, the creation of a GPU based simulator for GP systems could be straightforward, and the application of GPU oriented optimizations can also be considered. Finally, it would be important to define a translation protocol from other P systems models to GP systems models. Additionally, it would be interesting to research transformations for some problematic P system ingredients for GPUs to easier ones.

Modeling the GPU

As a theoretical research line, it would be also challenging to turn around and provide an analysis of the computational aspects of GPUs by using P systems. Modeling the GPU using P systems would permit the application of the theory developed in Membrane Computing to characterize this tool. In fact, there are some attempts to model SIMD machines by P systems. This research line would also offer the analysis and improvement of GPU methods from an algorithmic point of view.

Appendices

A
How to Use Guide

The simulators developed in our work are implemented in a stand-alone software application. Each application has their own parameters. In this appendix we describe how to use each of them.

We recall that they can be downloaded from the PMCGPU project website `http://sourceforge.net/p/pmcgpu` [17]. Since the whole project has been developed using CUDA, the computer utilized to run the simulators must have a CUDA capable GPU, together with all the related software. Everything related with CUDA can be checked on the official CUDA website `http://www.nvidia.com/cuda`. Note that some webpages related to CUDA may vary on time. An introduction to CUDA is also provided in Chapter 3 of this thesis.

Finally, it is noteworthy that the system where we have tested the simulators is based on Ubuntu Linux operating system, CUDA release 4.0, and a GPU with CUDA capability 1.3. Newer versions of CUDA (5.0 and newer), Windows operating system, and GPUs with CUDA capability beyond 1.3 have not been tested, and the simulators may not properly compile on such platforms. In that case, it is recommended to adapt the source code.

A.1 Installing CUDA

Before installing one of our simulators, it is required to check if our system is ready for them. First, it is required to install a CUDA capable GPU. Secondly, the corresponding software has to be configured. Next we summarize the steps to configure the computer before compiling and executing the developed simulators.

1. Check if the computer to run the simulators have a CUDA capable GPU. It can be confirmed visiting `https://developer.nvidia.com/cuda-gpus`.
2. Download the CUDA release version 4.0 (`https://developer.nvidia.com/cuda-toolkit-40`). The latest version of CUDA at the time of typing this thesis is 5.0, but we have tested all the simulators with version 4.0. New versions supporting CUDA version 5.0 will be announced through the PMCGPU website.
3. Install the corresponding driver for the GPU to use.
4. Install the CUDA toolkit 4.0.
5. Install the CUDA SDK 4.0 in the home folder.
6. Download and install the *counterslib* library provided in the PMCGPU website.

A.2 Installing and using PCUDA

In order to install the PCUDA simulators, first download the corresponding *.tar* file from the PMCGPU website. Then, follow the next guidelines.
Installation process:

1. Extract the file inside *~/NVIDIA_GPU_Computing_SDK/C/src*
2. Compile it: *make*

Parameters (or type -h): usage *pcuda <params>*, where <params> can be:

- -vX: Indicates the verbosity level. No verbose activated by default. The verbosity levels are:
  - -v1: Print only the last configuration.
  - -v2: Print only the configuration of the skin in every step, and at the end, the configuration of the remaining membranes.

  - -v3: Print all the information of all the membranes in each configuration, the alphabet and the set of rules.

- -c: Define a configuration file (no implemented yet). Default: config.cfg.
- -l: Define the maximum number of steps to do. Default: 256.
- -i: Define the input binary file.
- -s: Executes only the sequential algorithm. Activated by default until defining some of the next params.
- -f: Launch fast sequential simulation (also -p 1).
- -p: Set the algorithm to be used in fast mode: 1=Fast sequential (also -f), 2=Only selection in parallel, 3(default)=Selection and execution in parallel.
- The next parameters are mandatory when configuring the fast algorithms:
  - -m: Define the maximum number of membranes that the P system will create.
  - -o: Define the maximum number of objects that one membrane can have in one step.
  - -b: Define the number of threads to execute per block.
  - -t: Define the threshold to achieve for executing the parallel algorithm (number of membranes).

Input file preparation and execution:

1. Write the corresponding P system with active membranes in a P-Lingua file ".pli", using the `@model<membrane_division>` model type.
2. Convert it to a binary file, using the jar file "pparser.jar" provided in the folder plingua, or downloading the last version of pLinguaCore from `http://www.p-lingua.org`. Example: *java -jar plingua/pparser.jar plingua/test.pli data/test.bin*
3. Extract the required information from the P system to simulate (from the binary file information). Execute the sequential simulator as follows: *../../bin/linux/release/pcuda -i data/test.bin -s -l 0.*

4. Ensure that the number of objects in the P system can be divisible by a number below 512. If not, it cannot be correctly simulated. If it is, factorize the number, and annotate: the number of objects (*obj*) and the factor number below 512 (*fac*).

5. Get a maximum value of membranes to be created (*memb*). You can previously know it from the P system definition, or running the sequential simulator without the option "-l".

6. Now, you can run either the fast sequential or the parallel simulator, as follows:

   - Fast sequential: *../../bin/linux/release/pcuda -i data/test.bin -t 1 -o obj -b fac -m memb -f*
   - Parallel: *../../bin/linux/release/pcuda -i data/test.bin -t 1 -o obj -b fac -m memb -p 3*

7. Use the value of verbosity (-v 0, 1, 2 ,3) to get the desired information.

A.3 Installing and using PCUDASAT

In order to install the PCUDASAT simulators, first download the corresponding *.tar* file from the PMCGPU website. Then, follow the next guidelines.
Installation process:

1. Extract the file inside */NVIDIA_GPU_Computing_SDK/C/src*

2. Compile it: *make*

Parameters (or type -h): usage *pcudaSAT <params>*, where params must be:

- -f: Define an input file.
- -s: Choose a simulator, that can be 0(default)=sequential version, 1=GPU version, 2=hybrid GPU version.
- -v: Define a verbosity level (not supported yet).

Input file preparation and execution: type a DIMACS CNF file codifying a CNF formula, and run the simulator with the provided parameters.

A.4 Installing and using TSPCUDASAT

In order to install the TSPCUDASAT simulators, first download the corresponding *.tar* file from the PMCGPU website. Then, follow the next guidelines.
Installation process:

1. Extract the file inside */NVIDIA_GPU_Computing_SDK/C/src*
2. Compile it: *make*

Parameters (or type -h): usage *tsp <params>*, where params must be:

- -f: Define an input file.
- -m: Choose a simulator, that can be 2=sequential version, and 5(default)=GPU version. The rest of the versions are experimental.

Input file preparation and execution: type a DIMACS CNF file codifying a CNF formula, and run the simulator with the provided parameters.

A.5 Installing and using ABCD-GPU

In order to install the ABCD-GPU simulators, first download the corresponding *.tar* file from the PMCGPU website. Then, follow the next guidelines.
Installation process:

1. Install the GSL standard library [11].
2. Extract the file inside */NVIDIA_GPU_Computing_SDK/C/src*
3. Compile it: *make*

Parameters (or type -h): usage *abcd <params>*, where params can be:

- -s: number of simulations.
- -a: accuracy in the algorithms (A parameter in the DCBA).
- -t: time steps (T parameter in the DCBA).
- -I: choose the implementations: 0 (default) for OpenMP simulator, and 1 for GPU simulator. Deprecated options: 10 for table based simulator, 11 for sequential simulator, 12 for parallel OpenMP simulator.

- -M: if I=1, M sets the behavior of the GPU simulator: 0 runs without CPU gold code (default), 1 runs with CPU, 2 runs with CPU and phase2 basic kernel.
- if I=12, M is the parallelism level: 0 for environments, 1 for simulations, 2 for hybrid-2s and 3 for hybrid-2e
- -F: unset accurate mode (demotes to float row additions).
- -v: verbosity level, that can be:
  - 0(default)=show only initial information.
  - 1=show a summary of time per step. A final analysis of time is also showed.
  - 2=show all the information: multisets of objects, selected rules, ... (use it only for small PDP systems).
- Special options for the random system generator:
  - -o: number of objects.
  - -q: number of membranes.
  - -b: number of rule blocks.
  - -r: maximum number of rules per block.
  - -l: maximum number of objects in the LHS/RHS.
  - -e: number of environments.
  - -X: select one of the prefixed examples: 0 (default: configured through parameters), 1 (small), 2 (medium), 3 (large).

Input file preparation and execution: input files are still not supported. The simulator runs against random generated PDP systems. Now, you can run either the sequential or the parallel simulators as follows:

- Sequential: *../../bin/linux/release/abcd -X 3 -I 0*
- CPU parallel (example for 4 threads):
  export OMP_NUM_THREADS=4
  ../../bin/linux/release/abcd -X 3 -I 0
- GPU parallel: *../../bin/linux/release/abcd -X 3 -I 1*

B
GPU Server Implementation

In this appendix the technical steps taken to configure and install the GPU server we have used for our experiments are summarized. The hardware and software employed are described, together with the configuration we have implemented. Further descriptions will be focused on the management policy for workloads on the server, and the configuration of the client with the NetBeans IDE. This appendix is intended to serve as a guide for those that would like to configure a single Linux based GPU server, since the taken steps were obtained from different information sources.

B.1 GPU server configuration

The first GPU server we have configured to develop in CUDA contained a Intel Core2 Quad CPU (4 cores), 8 GBytes of RAM, and 1 NVIDIA GPU Tesla C1060. The 32-bit Ubuntu 8.10 server operating system was installed on it. The computer was used to develop PCUDA and PCUDASAT simulators. It was accessed through *ssh* protocol, and the development process was carried out in text mode.

However, after the successful results obtained from the simulators, we have configured a bigger GPU server for our purposes, and the previous server was demoted for Windows users. The new server contains two Intel i5 Nehalem based Xeon E5504 CPUs (a total of 8 cores), 12 GBytes of RAM, and 2

Figure B.1: The GPU server where all the results presented in the thesis were obtained.

NVIDIA Tesla C1060. However, the 64-bit Ubuntu 10.04 server was installed on it. The 64-bit version was installed to effectively use the 4 GBytes of device memory in Tesla C1060 (with the 32-bit driver version, only up to 2 GBytes were possible to allocate).

After solving problems with the power supply of the server, the computer was installed inside an special rack cabinet, which is placed in a special server room maintained at 22^o C approximately. The real server is shown in Figure B.1.

The access to the server was made again through *ssh* protocol. This is controlled in a safe way using the *ufw* firewall, by the command `ufw limit ssh/tcp`. The two GPUs (Tesla C1060) were installed using as reference their corresponding installation guide, and configured using the *NVIDIA CUDA C getting started guide for Linux*. Both guides are available from the official CUDA website `http://www.nvidia.com/cuda`. Following these indications, we have configured the next script to be run in any system startup:

```
#!/bin/bash

modprobe nvidia

if [ "$?" -eq 0 ]; then

# Count the number of NVIDIA controllers found.
N3D=‘/bin/lspci | grep -i NVIDIA | grep "3D controller" | wc -l‘
NVGA=‘/bin/lspci | grep -i NVIDIA | grep "VGA compatible controller"
      | wc -l‘

```

```
N=‘expr $N3D + $NVGA - 1‘
for i in ‘seq 0 $N‘; do
mknod -m 666 /dev/nvidia$i c 195 $i;
done

mknod -m 666 /dev/nvidiactl c 195 255

nvidia-smi -c 1 -g 1

else
exit 1
fi

```

Finally, we have modified the `$HOME/.bashrc` file of each user (by creating the file `/etc/skel/.bashrc`) to configure some environment variables as follows:

- `export PATH=$PATH:/usr/local/cuda/bin`
- `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`
  `/usr/local/cuda/lib64:/usr/local/cuda/lib`

B.2 Workload policy

Given that the server contains 2 GPUs, it is needed to designate a function for each one before setting them up. The GPUs are globally identified by a number, that is in our case, 0 and 1. The properties and configuration of them can be made through the *nvidia-smi* command. In addition, a GPU can be selected directly on the CUDA code through the `cudaSetDevice()` instruction.

We have made the following assignation to achieve a good workload distribution:

- **GPU 0**: it is intended for running test examples from the users. It can be used simultaneously by several users (it has the default compute mode). Moreover, it is designated as the debugger GPU. In order to control the access to the card for debugging, we have prepare a debug scheduling protocol. Users can read and write into a shared filed, where they annotate the date and time they will be debugging. Finally, we have configured *cuda-gdb* application for this purpose. Additionally, *DDD* graphical application can be used together with *cuda-gdb* by typing the following command: `ddd --debugger cuda-gdb BINARY`.

- **GPU 1**: it is intended for running applications separately, one by one, to safely analyze the execution of CUDA applications. It has been configured in exclusive compute mode (only one user can use it at a time), by the command `nvidia-smi -c 1 -g 1`. For this purpose, we have installed and configured a job scheduler application. It offers a queue where to submit the jobs only using device 1. That way, the execution on this GPU is centrally managed by one application, and users do not have to actively wait for using it (the job scheduler automatically controls this).

B.3 Software installation

The job scheduler we have used for our server is the (Oracle) Grid Engine [7] (formerly known as SGE, Sun Grid Engine), which is broadly used in clusters and facilitates "distributed resource management" (DRM). It is based on the idea of having a *master* computer which launch the *jobs* to a set of *hosts*. In our case, the master and the host are the same machine.

The packages required for the installation in Ubuntu Linux 10.04 were: *gridengine-client, gridengine-common, gridengine-master, gridengine-qmon* and *gridengine-exec*. The *qconf* command is the one used to configure the whole system. We have created a new queue for implementing the GPU 1 functionality. We have used the following configuration procedure:

- *qconf -aq*: creates a new queue. It opens a text editor, where to fill the options fields. The filled options were:
  - *"qname"*: the name of the queue (our case, *device1*).
  - *"hostlist"*: the hosts executing the jobs (our case, *localhost*).
  - *"slots"*: allow to execute more than one job at a time in the hosts. For example, it can be used to use all the cores of a processor. We have kept the number to 1, by default. If not, just place *[localhost=c]*, being *c+1* the number of cores.

Finally, we have configure a template script to launch the jobs to the system. The utilized command is *qsub script.sh*.

B.4 Client configuration

As mentioned before, the first simulators of PCUDA and PCUDASAT were remotely developed on text mode interface, using the *vim* editor. However, a big project, such as ABCD-GPU, required the usage of an IDE (Integrated Development Environment). The first option we have considered was the *Eclipse* environment with the *CDT* plugin. But the idea was to use the remote GPU server for compiling and testing, and *Eclipse* does not support it yet. It is noteworthy that since CUDA release 5.0, the Parallel Nsight for Linux is available under *Eclipse*, but only supports local compilation, execution and debug.

We then switched to the *NetBeans IDE*. It also supports C/C++ development, and provides a very easy implementation of remote compilation and execution. This option can be configured by *right-click* on the project folder, select the *Set Remote Build Host* option, and configure/select the server. It can be configured to automatically copying the files through the *ssh* protocol.

Finally, the support for CUDA is not complete under *NetBeans IDE*. Although there is a plugin for this (available at the website `http://plugins.netbeans.org/plugin/36176/cuda-plugin`), the IDE has still to be carefully configured. Some taken steps in this regard were the following (note that we are using NetBeans 7.0.1):

- Configure the C/C++ plugin:
  - Select *Tools -> Options*.
  - Select *C/C++* options.
  - In the *Build Tools* tab, configure the tools for the localhost and server machines as follows:
    - Base directory: */usr/local/cuda/bin*
    - C/C++ Compiler: */user/local/cuda/bin/nvcc*
    - Make command: */usr/bin/make*
    - Debugger command: */usr/local/cuda/bin/cuda-gdb*
  - In the *Code Assistance* tab, add the include paths to CUDA for both C and C++ compilers:
    - */usr/local/cuda/include*
    - *$HOME/NVIDIA_GPU_Computing_SDK/C/common/inc*
    - *$HOME/NVIDIA_GPU_Computing_SDK/shared/inc*
  - Also in the *Code Assistance* tab, add to *Macro Definitions*, the reserved CUDA keywords, such as *__device__*, *__global__*, *__host__* and *__shared__*.
  - In the *Highlighting* tab, uncheck the options *Highlighting Syntax Errors* and *Highlight Unresolved identifiers*.

  - In the *Other* tab, add the extensions *.cuh* (header) and *.cu* (C++ files).

- Create a CUDA project selecting the *Cuda project* option, provided by the installed plugin.

- Configure the CUDA project:

  - *Right-click* to the project folder, and select *properties*.
  - In the *Build* option, select the *Configuration* for *CUDA-Toolkit* or *CUDA-SDK*.
  - Again in *Build* option, uncheck the option *Enable Make Dependency Checking*.
  - In *Build / linker* option, configure the *Additional Library Directories* with *../../NVIDIA_GPU_Computing_SDK/C/lib: ../../NVIDIA_GPU_Computing_SDK/shared/lib*
  - We recommend to run the application directly in the terminal inside the *NetBeans IDE*. To do so, in the *Run* option, configure the *Environment* with the variables *PATH* and *LD_LIBRARY_PATH* as mentioned before. Moreover, configure a command for running the application.
  - Note that debugging inside NetBeans is partially supported since we have configured the *Debugger command* option to *cuda-gdb*. For better debugging experience, one can use the *cuda-gdb* or *ddd* applications [62].

Bibliography

[1] B. Barney. Introduction to Parallel Computing. https://computing.llnl.gov/tutorials/parallel_comp. Tutorial of "Using LLNL's Supercomputers" workshop.

[2] GNU GPL license. http://www.gnu.org/licenses/gpl.html.

[3] Inside HPC blog. http://insidehpc.org.

[4] Message Passing Interface forum. http://www.mpi-forum.org. Location containing the official MPI standards documents.

[5] NVIDIA CUDA C Programming Guide 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[6] OpenCL standard webpage. http://www.khronos.org/opencl.

[7] Oracle Grid Engine. http://www.oracle.com/technetwork/oem/grid-engine-166852.html.

[8] P-Lingua release 1.0 website. http://www.gcn.us.es/plingua.

[9] The Bearded Vulture ecosystem model in P-Lingua. http://www.p-lingua.org/wiki/index.php/bvBWMC12.

[10] The Berkeley Open Infrastructure for Network Computing (BOINC) official website. http://boinc.berkeley.edu.

[11] The GNU Standard Library. http://www.gnu.org/s/gsl.

[12] The GNUplot web page. http://www.gnuplot.info.

[13] The GPGPU organization. http://www.gpgpu.org.

[14] The NVIDIA CUDA Random Number Generation library (cuRAND). `https://developer.nvidia.com/curand`.

[15] The OpenMP API specification for parallel programming. `http://www.openmp.org`. Official website.

[16] The P-Lingua web page. `http://www.p-lingua.org`.

[17] The PMCGPU (Parallel simulators for Membrane Computing on the GPU) project website. `http://sourceforge.net/p/pmcgpu`.

[18] The top 500 supercomputer site. `http://www.top500.org`.

[19] M. Qasem. WinSAT website. `http://www.mqasem.net/sat/winsat`, 2009.

[20] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(11):1021–1024, 1994.

[21] A. Alhazov, C. Martín-Vide, and L. Pan. Solving graph problems by P systems with restricted elementary active membranes. In N. Jonoska, G. Păun, and G. Rozenberg, editors, *Aspects of Molecular Computing*, volume 2950 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2004.

[22] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference*, volume 30, pages 483–485, Atlantic City, NJ, April 1967.

[23] A. A. Aqrawi and A. C. Elster. Bandwidth reduction through multithreaded compression of seismic images. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1730–1739, may 2011.

[24] D. Balbontín-Noval, M. Pérez-Jiménez, and F. Sancho-Caparrini. A MzScheme implementation of transition P systems. In G. Păun, G. Rozenberg, A. Salomaa, and C. Zandron, editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin Heidelberg, 2003.

[25] D. Besozzi, P. Cazzaniga, D. Pescini, and G. Mauri. Modelling metapopulations with stochastic membrane systems. *Biosystems*, 91(3):499–514, 2008.

[26] L. Bianco and A. Castellini. Psim: a computational platform for metabolic P systems. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2007.

[27] L. Bianco, F. Fontana, and V. Manca. P systems with reaction maps. *International Journal of Foundations of Computer Science*, 17(1):27–48, 2006.

[28] J. Blakes, J. Twycross, F. J. Romero-Campero, and N. Krasnogor. The infobiotics workbench: an integrated *in silico* modelling platform for systems and synthetic biology. *Bioinformatics*, 27(23):3323–3324, 2011.

[29] G. E. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[30] G. Bravo, L. Fernández, F. Arroyo, and M. A. Peña. Hierarchical master-slave architecture for membrane systems implementation. In *Thirteenth International Symposium on Artificial Life and Robotics 2008*, AROB 13th, 2008.

[31] F. G. Cabarle, H. N. Adorna, and M. A. Martínez-del-Amor. A spiking neural P system simulator based on CUDA. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 87–103. Springer Berlin Heidelberg, 2012.

[32] F. G. Cabarle, H. N. Adorna, M. A. Martínez-del-Amor, and M. J. Pérez-Jiménez. Improving GPU simulations of spiking neural P systems. *Romanian Journal of Information Science and Technology*, 15:5–20, 2012.

[33] M. Cardona, M. A. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A computational modeling for real ecosystems based on P systems. *Natural Computing*, 10(1):39–53, 2011.

[34] M. Cardona, M. A. Colomer, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A P system based model of an ecosystem of some scavenger birds. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 182–195. Springer Berlin Heidelberg, 2010.

[35] M. Cardona, M. A. Colomer, M. J. Pérez-Jiménez, D. Sanuy, and A. Margalida. Modeling ecosystems using P systems: the bearded vulture, a case study. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 137–156. Springer Berlin Heidelberg, 2009.

[36] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury, CA, USA, 2nd edition, 2001.

[37] A. Castellini and V. Manca. Metaplab: A computational framework for metabolic p systems. In *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin Heidelberg, 2009.

[38] A. Castellini, V. Manca, and Y. Suzuki. Metabolic P system flux regulation by artificial neural networks. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 196–209. Springer Berlin Heidelberg, 2010.

[39] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Implementing P systems parallelism by means of GPUs. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2010.

[40] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79(6):317–325, 2010.

[41] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[42] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. P systems simulations on massively parallel architectures. In *Third International Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 17–26, Vienna, Austria, 2010.

[43] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. The GPU on the simulation of cellular computing models. *Soft Computing*, 16(2):231–246, 2012.

[44] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, Ignacio Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P Systems with active membranes on CUDA. In *2009 International Workshop on High Performance Computational Systems Biology, HIBI'09*, pages 61–71, Trento, Italy, 2009. IEEE Computer Society.

[45] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. A massively parallel framework using P systems and GPUs. In *Symposium on Application Accelerators in High Performance Computing*, Illinois, USA, 2009.

[46] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. Enhancing the simulation of P systems for the SAT problem on GPUs. In *Symposium on Application Accelerators in High Performance Computing*, Knoxville, USA, July 2010 2010.

[47] S. Cheruku, A. Păun, F. J. Romero-Campero, M. J. Pérez-Jiménez, and O. H. Ibarra. Simulating FAS-induced apoptosis by using P systems. *Progress in Natural Science*, 17:424–431, 2007.

[48] G. Ciobanu and D. Paraschiv. P system software simulator. *Fundamenta Informaticae*, 49(1):61–66, 2002.

[49] G. Ciobanu and G. Wenyuan. A P system running on a cluster of computers. In *Lecture Notes in Computer Science*, WMC 2003, pages 123–150. Springer-Verlag, 2004.

[50] M. Colomer, A. Margalida, D. Sanuy, and M. J. Pérez-Jiménez. A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222(1):33–47, 2011.

[51] M. Colomer, M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, and A. Riscos-Núñez. A uniform framework for modeling based on P systems. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, volume 1, pages 616–621, September 2010.

[52] M. Colomer, I. Pérez-Hurtado, M. Pérez-Jiménez, and A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem. *Natural Computing*, 11(3):369–379, 2012.

[53] M. A. Colomer, S. Lavín, I. Marco, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, D. Sanuy, E. Serrano, and L. Valencia-Cabrera. Modeling population growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by using P systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2011.

[54] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[55] A. Cordón-Franco, M. Gutiérrez-Naranjo, M. Pérez-Jiménez, and F. Sancho-Caparrini. A Prolog simulator for deterministic P systems with active membranes. *New Generation Computing*, 22(4):349–364, 2004.

[56] T. S. Crow. Evolution of the graphical processing unit. Master's thesis, University of Nevada Reno, `http://www.cse.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf`, 2004.

[57] D. Díaz-Pernil, C. Graciani-Díaz, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. *Software for P systems*, chapter 17, pages 437–454. Oxford University Press, Oxford (U.K.), 2010.

[58] D. Díaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua programming environment for Membrane Computing. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 187–203. Springer Berlin Heidelberg, 2009.

[59] L. Diez Dolinski, R. Núñez Hervás, M. Cruz Echeandía, and A. Ortega. Distributed simulation of P systems by means of Map-Reduce: first steps with Hadoop and P-Lingua. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*, volume 6691 of *Lecture Notes in Computer Science*, pages 457–464. Springer Berlin Heidelberg, 2011.

[60] V. Eijkhout. *Introduction to High Performance Scientific Computing.* 1st edition, 2011.

[61] A. C. Elster. High-Performance Computing: past, present, and future. In J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, P. Raback, and V. Savolainen, editors, *Applied Parallel Computing*, volume 2367 of *Lecture Notes in Computer Science*, pages 433–444. Springer Berlin Heidelberg, 2006.

[62] R. Farber. *CUDA application design and development.* Elsevier, 2011.

[63] L. Fernández, F. Arroyo, J. A. Tejedor, and J. Castellanos. Massively parallel algorithm for evolution rules application in transition P systems. In *7th Workshop on Membrane Computing*, pages 337–343, Leiden, The Netherlands, July 2006.

[64] L. Fernández, V. J. Martínez, F. Arroyo, and L. F. Mingo. A hardware circuit for selecting active rules in transition P systems. In *SYNASC*, pages 415–418, 2005.

[65] R. P. Feynman. There's plenty of room at the bottom. *Engineering and Science*, 23(5):22–36, February 1960.

[66] F. Fontana, L. Bianco, and V. Manca. P systems and the modeling of biochemical oscillations. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 199–208. Springer Berlin Heidelberg, 2006.

[67] I. T. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995. `http://www.mcs.anl.gov/~itf/dbpp`.

[68] I. T. Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), 2002.

[69] M. García-Quismondo, R. Gutiérrez-Escudero, M. A. Martínez-del-Amor, E. Orejuela-Pinedo, and I. Pérez-Hurtado. P-Lingua 2.0: a software framework for cell-like P systems. *International Journal of Computers, Communications and Control*, 4(3):234–243, 2009.

[70] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and

A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[71] M. García-Quismondo, L. F. Macías-Ramos, and M. J. Pérez-Jiménez. Implementing enzymatic numerical P systems for AI applications by means of graphic processing units. In J. Kelemen, J. Romportl, and E. Zackova, editors, *Beyond Artificial Intelligence*, volume 4 of *Topics in Intelligent Engineering and Informatics*, pages 137–159. Springer Berlin Heidelberg, 2013.

[72] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[73] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing.* Pearson Education, Harlow, England, 2nd edition, 2003.

[74] G. D. Guerrero, J. M. Cecilia, J. M. García, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Analysis of P systems simulation on CUDA. In *XX Jornadas de Paralelismo*, pages 289–294, A coruña, Spain, September 2009. Servizo de Publicacións, Universidade da Coruña.

[75] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[76] A. Gutiérrez and S. Alonso. *P systems: from theory to implementation*, chapter 17, pages 205–226. Concept Press Ltd, Hong Kong, 2010.

[77] A. Gutiérrez, L. Fernández, F. Arroyo, and S. Alonso. Hardware and software architecture for implementing membrane systems: A case of study to transition P systems. In M. Garzon and H. Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 211–220. Springer Berlin Heidelberg, 2008.

[78] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A fast P system for finding a balanced 2-partition. *Soft Computing*, 9(9):673–678, 2005.

[79] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. On descriptive complexity of P systems. In G. Mauri, G. Paun, M. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors, *Membrane Comput-*

ing, volume 3365 of *Lecture Notes in Computer Science*, pages 320–330. Springer Berlin Heidelberg, 2005.

[80] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Available membrane computing software. In G. Ciobanu, G. Păun, and M. J. Pérez-Jiménez, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 411–436. Springer Berlin Heidelberg, 2006.

[81] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. On the degree of parallelism in membrane systems. *Theoretical Computer Science*, 372(2-3):183–195, 2007.

[82] T. Head. Formal language theory and dna: an analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[83] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[84] ID Quantique SA. Random number generation using quantum physics, 2010. Quantis white paper 3.0.

[85] S. Jørgensen. *Ecological Modelling. An introduction.* WIT press, Southampton, Boston, 2009.

[86] R. A. Juayong, F. G. Cabarle, H. N. Adorna, and M. A. Martínez-del-Amor. On the simulations of evolution-communication P systems with energy without antiport rules for GPUs. In *Tenth Brainstorming Week on Membrane Computing*, volume I, pages 267–290, Seville, Spain, February 2012. Fénix Editora.

[87] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.

[88] B. W. Kernighan and D. Ritchie. *The C programming language.* Prentice Hall, 2nd edition, 1988.

[89] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[90] K. Krewell. Cell moves into the limelight, 2005. Microprocessor Report.

[91] D. P. Kroese, T. Taimre, and Z. I. Botev. *Handbook of Monte Carlo Methods*. Wiley, New Jersey, USA, 1st edition, 2011.

[92] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[93] H. Li and L. R. Petzold. Efficient parallelization of stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. Technical report, Dept. Computer Science, University of California, 2007.

[94] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[95] M. M. Membrane Computing in Prolog. In *Pre-proceedings of the Workshop on Multiset Processing*, pages 159–175, 2000.

[96] V. Manca. *Fundamentals of Metabolic P Systems*, chapter 19, pages 475–498. Oxford University Press, Oxford (U.K.), 2010.

[97] V. Manca, R. Pagliarini, and S. Zorzan. A photosynthetic process modelled by a metabolic P system. *Natural Computing*, 8(4):847–864, 2009.

[98] M. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M.A. Colomer, and M.J. Pérez-Jiménez. DCBA: Simulating population dynamics P systems with proportional object distribution. In *Proceedings of the 13th International Conference on Membrane Computing (CMC13)*, pages 291–310, Budapest, Hungary, August 2012.

[99] M. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M.A. Colomer, and M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P systems with proportional object distribution. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume II, pages 27–56, Seville, Spain, February 2012. Fénix Editora.

[100] M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, and M. Colomer. A new simulation algorithm for multienvironment probabilistic P systems. In *IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010)*, volume 1, pages 59–68, September 2010.

[101] M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, and F. Sancho-Caparrini. A simulation algorithm for multienvironment probabilistic p systems: a formal verification. *International Journal of Foundations of Computer Science*, 22(01):107–118, 2011.

[102] M. A. Martínez-del-Amor, J. M. Cecilia, G. D. Guerrero, and I. Pérez-Hurtado. An overview of P system simulation on GPUs. In *I Jornadas Jóvenes Investigadores*, pages 2–7, Cáceres, Spain, April 2010.

[103] M. A. Martínez-del-Amor, I. Karlin, R. E. Jensen, M. J. Pérez-Jiménez, and A. C. Elster. Parallel simulation of probabilistic P systems on multicore platforms. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume II, pages 17–26, Seville, Spain, February 2012. Fénix Editora.

[104] M. A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A. C. Elster, and M. J. Pérez-Jiménez. Population Dynamics P Systems on CUDA. In D. Gilbert and M. Heiner, editors, *Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 247–266. Springer Berlin Heidelberg, 2012.

[105] M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, J. M. Cecilia, G. D. Guerrero, and J. M. García. Simulating active membrane systems using GPUs. In G. Paun, M. J. Pérez-Jiménez, and A. Riscos-Núñez, editors, *10th Workshop on Membrane Computing*, pages 369–384, Curtea de Arges, Rumania, August 2009. Marpapublicidad.

[106] M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, J. M. Cecilia, G. D. Guerrero, and J. M. García. Simulation of recognizer P systems by using manycore GPUs. In *7th Brainstorming Week on Membrane Computing*, volume II, pages 45–58, Sevilla, España, February 2009. Fénix Editora.

[107] M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A p-lingua based simulator for tissue p systems. *The Journal of Logic and Algebraic Programming*, 79(6):374–382, 2010.

[108] C. Maxfield. *The Design Warrior's Guide to FPGAs.* Elsevier, 2004.

[109] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[110] M. Minsky and S. Papert. *Perceptions.* MIT Press, 1970.

[111] V. Moya, C. González, J. Roca, A. Fernández, and R. Espasa. Shader performance analysis on a modern GPU architecture. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.

[112] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide.* Addison-Wesley, 1st edition, 2011.

[113] L. Mussi and S. Cagnoni. Particle swarm optimization within the CUDA architecture. In *GPUs for Genetic and Evolutionary Computation (GECCO)*, 2009.

[114] E. Nabil, H. Hameed, and A. Badr. A cloud based P systems algorithm. *International Journal of Computer Applications*, 54(13):26–31, 2012.

[115] V. Nguyen, D. Kearney, and G. Gioiosa. Balancing performance, flexibility, and scalability in a parallel computing platform for Membrane Computing applications. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 385–413. Springer Berlin Heidelberg, 2007.

[116] V. Nguyen, D. Kearney, and G. Gioiosa. An algorithm for non-deterministic object distribution in p systems and its implementation in hardware. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2009.

[117] V. Nguyen, D. Kearney, and G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in Reconfig-P. *Journal of Logic and Algebraic Programming*, 79(6):383–396, 2010.

[118] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[119] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[120] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, Burlington, USA, 1st edition, 2011.

[121] G. Păun and F. J. Romero-Campero. Membrane Computing as a modeling framework. Cellular systems case studies. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Formal Methods for Computational Systems Biology*, volume 5016 of *Lecture Notes in Computer Science*, pages 168–214. Springer Berlin Heidelberg, 2008.

[122] F. Peña-Cantillana, D. Díaz-Pernil, H. A. Christinal, and M. A. Gutiérrez-Naranjo. Implementation on CUDA of the smoothing problem with tissue-like P systems. *International Journal on Natural Computing Research*, 2(3):25–34, 2011.

[123] J. Pérez-Carrasco. Aceleración de simulaciones de sistemas celulares en soluciones del problema sat usando gpus. Master's thesis, Higher Technical School of Computer Engineering, July 2012.

[124] I. Pérez-Hurtado. *Desarrollo y aplicaciones de un entorno de programación para Computación Celular: P-Lingua.* PhD thesis, University of Seville, 2010.

[125] I. Pérez-Hurtado, L. Valencia-Cabrera, M. Pérez-Jiménez, M. Colomer, and A. Riscos-Núñez. MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems. In *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, volume 1, pages 637–643, September 2010.

[126] M. Pérez-Jiménez and F. Romero-Campero. A CLIPS simulator for recognizer P systems with active membranes. In *2nd Brainstorming Week on Membrane Computing*, pages 387–413, Seville, Spain, February 2004. Fénix Editora.

[127] M. Pérez-Jiménez and F. Romero-Campero. A study of the robustness of the EGFR signalling cascade using continuous membrane systems. In *Membrane Computing*, volume 3561 of *Lecture Notes in Computer Science*, pages 268–278. Springer Berlin Heidelberg, 2005.

[128] M. Pérez-Jiménez and F. Romero-Campero. P systems, a new computational modelling tool for systems biology. In C. Priami and G. Plotkin, editors, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in Computer Science*, pages 176–197. Springer Berlin Heidelberg, 2006.

[129] M. J. Pérez-Jiménez and A. Riscos-Núñez. A linear-time solution to the knapsack problem using p systems with active membranes. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 250–268. Springer Berlin Heidelberg, 2004.

[130] M. J. Pérez-Jiménez and A. Riscos-Núñez. Solving the Subset-Sum problem by P systems with active membranes. *New Generation Computing*, 23(4):339–356, 2005.

[131] M. J. Pérez-Jiménez and F. J. Romero-Campero. An efficient family of P systems for packing items into bins. *Journal of Universal Computer Science*, 10(5):650–670, 2004.

[132] M. J. Pérez-Jiménez and F. J. Romero-Campero. Attacking the common algorithmic problem by recognizer P systems. In M. Margenstern, editor, *Machines, Computations, and Universality*, volume 3354 of *Lecture Notes in Computer Science*, pages 304–315. Springer Berlin Heidelberg, 2005.

[133] M. J. Pérez-Jiménez, Á. Romero-Jiménez, and F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3):265–285, 2003.

[134] M. J. Pérez-Jiménez, A. Romero-Jiménez, and F. Sancho-Caparrini. A polynomial complexity class in P systems using membrane division. *Journal of Automata, Languages and Combinatorics*, 11:423–434, 2006.

[135] D. Pescini, D. Besozzi, G. Mauri, and C. Zandron. Dynamical probabilistic p systems. *International Journal of Foundations of Computer Science*, 17(1):183–204, 2006.

[136] B. Petreska and C. Teuscher. A reconfigurable hardware membrane system. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 269–285. Springer Berlin Heidelberg, 2004.

[137] P. Pospichal and J. Jaros. GPU-based acceleration of the genetic algorithm. In *GPUs for Genetic and Evolutionary Computation (GECCO)*, 2009.

[138] C. Prabhu. *Grid and Cluster Computing.* Prentice-Hall, 2008.

[139] G. Păun. P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6:75–90, 1999.

[140] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143; Turku Center for CS–TUCS Report No 208 (1998), 2000.

[141] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, USA, 2010.

[142] A. Regev and E. Shapiro. *The π-calculus as an abstraction for biomolecular systems.* Modelling in Molecular Biology. Springer Berlin, 2004.

[143] F. J. Romero-Campero and M. J. Pérez-Jiménez. A model of the quorum sensing system in vibrio fischeri using P systems. *Artificial Life*, 14(1):95–109, 2008.

[144] F. J. Romero-Campero and M. J. Pérez-Jiménez. Modelling gene expression control using p systems: The lac operon, a case study. *Biosystems*, 91(3):438–457, 2008.

[145] F. Rosenblatt. The perception: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[146] A. Ruíz, M. Ujaldón, J. A. Andrades, J. Becerra, K. Huang, T. Pan, and J. H. Saltz. The GPU on biomedical image processing for color and phenotype analysis. In *7th IEEE International Conference on Bioinformatics and Bioengineering*, pages 1124–1128, Piscataway, NJ, USA, October 2007. IEEE Computer Society.

[147] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE Computer Society, May 2009.

[148] S. Sedwards and T. Mazza. Cyto-sim: a formal language model and stochastic simulator of membrane-enclosed biochemical processes. *Bioinformatics Applications Note*, 23(20):2800–2802, 2007.

[149] T. Stützle. Parallelization strategies for ant colony optimization. In A. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 722–731. Springer Berlin Heidelberg, 1998.

[150] A. Syropoulos, L. Mamatas, P. C. Allilomes, and K. T. Sotiriades. A distributed simulation of transition P systems. In *Workshop on Membrane Computing*, pages 357–368, 2003.

[151] J. A. Tejedor, L. Fernández, F. Arroyo, and G. Bravo. An architecture for attacking the communication bottleneck in P systems. *Artificial Life and Robotics*, 12(1-2):236–240, 2008.

[152] G. Terrazas, N. Krasnogor, M. Gheorghe, F. Bernardini, S. Diggle, and M. Cámara. An environment aware P system model of quorum sensing. In S. Cooper, B. Löwe, and L. Torenvliet, editors, *New Computational Paradigms*, volume 3526 of *Lecture Notes in Computer Science*, pages 479–485. Springer Berlin Heidelberg, 2005.

[153] W. Voorsluys, J. Broberg, and R. Buyya. *Introduction to Cloud Computing*, pages 1–41. Wiley press, 2011.

[154] N. Whitehead and A. Fit-Florea. Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs, 2011.

[155] C. Zandron, C. Ferretti, and G. Mauri. Solving np-complete problems using p systems with active membranes. In *Proceedings of the Second International Conference on Unconventional Models of Computation*, UMC '00, pages 289–301, London, UK, 2001. Springer-Verlag.

[156] X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan, and M. J. Pérez-Jiménez. Matrix representation of spiking neural p systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg, 2011.