6 P Systems Implementation on GPUs

6.1 Introduction

The development of P system simulators is usually driven by the importance of certain models. Usually, these simulators are implemented in a flexible way, allowing not only to simulate a wide variety of P systems but also to help construct simulators for other models. An example of this flexibility is P-Lingua framework[13]. However, for certain applications and models, efficient simulation tools are required. For instance, the simulation of population dynamics P systems is crucial for model validation of real ecosystems and for virtual experimentation. In this case, the faster the simulation tool, the shorter the time to construct a valid model. Another interest behind the development of efficient tools is also for analyzing theoretical aspects of P systems (parallelism, non-determinism, etc.) and how to bridge them with today, *in-silico* technology [48].

There are several ways to accelerate the simulation of P systems: changing the technology where to implement the simulators (e.g., from interpreted languages like Java to compiled ones like C++), increasing the power of the processors where to run the simulations (e.g., increasing the clock frequency, the memory bandwidth and clock, etc.), or using high performance computing (HPC) technologies to implement real parallelism. The main trend when developing efficient simulators has been the last one: taking advantage of the inherent parallelism of the models and mapping it into parallel platforms such as clusters, supercomputers, accelerators, etc.

According to [15], we can define high performance computing (HPC) as *"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"*. This is usually accomplished by means of parallelism, since it is the basis for the acceleration of large and complex real-world applications. The maximum exponent of HPC is known as supercomputing, where the computing power of current technology is being continuously pushed. A ranking of the most powerful

G. Zhang et al., *Membrane Computing Models: Implementations*,
https://doi.org/10.1007/978-981-16-1566-5_6

supercomputers can be consulted at Top500 website [46]. To the time of writing, most of the top 10 supercomputers are based on nodes extended with accelerators.

HPC accelerators are dedicated chips that serve as co-processors, extending in this way the computing power. Examples of accelerators are FPGAs and GPUs. The latter refers to the processors inside the graphics cards, which take over the task of graphics generation in computers. However, with the increasing demand in 3D rendering for gaming and video, and as foreseen by Elster [12] and others, GPUs (Graphics Processing Units) have evolved to a massively parallel processor that is suitable for parallel computing. Today, GPUs are the enabling technology for trending areas such as Deep Learning, Data Science, physics simulation, real-time ray tracing graphics, etc.

Concerning P systems and their applications, GPUs have been shown to be an alternative for accelerating simulations. In [4], the double parallelism of P systems were mapped over the double parallelism inside GPUs. This idea has been refined over the time, in such a way that the simulators are now better adapted to GPU parallelism. We can identify three types of simulators: those developed for very specific P systems or family of P systems (specific simulators) and developed for a wide range of P systems inside a variant (generic simulators) and a hybrid simulator that receives high-level information to be better adapted (adaptive simulators).

In this chapter, we will introduce all the concepts related with GPU computing and its applications. Later, we will go through some P system simulators depending on their type: specific, generic, or hybrid. Finally, we will provide some guidelines on how to develop new simulators for P systems on GPUs.

6.2 GPU Computing

In this section, we will introduce the main concepts of GPU computing, including CUDA and modern GPU architecture. This will provide the required background to understand the design of P system simulators on GPUs.

6.2.1 The Graphics Processing Unit

The first Graphics Processing Units (GPUs) were introduced back in 1999 [11], in order to overcome the bottleneck created by the CPU when generating real-time graphics, such as in videogames and in 3D rendering. Since then, every graphics card has integrated such kind of specific processors. Usually, we refer to GPU and graphics card as synonyms. It is important to remark the place where the GPU is located on a computer. The GPU is connected with the CPU through a data bridge (Northbridge), which is also used to access the main memory (RAM). Currently, modern GPUs are connected through the so called PCI Express bus, which runs at more than 64GB/s (when using 16-lane configuration).

GPUs, since they were born, install processor cores that are specialized for graphics (pixel colorization, etc.). Thus, these processors are able to include more

cores than usual CPUs because they are more specific and, so, lightweight. This GPU architecture has evolved over the time, being both more parallel and flexible. The former means that it included more and more computing cores, and the latter means that these cores were more programmable. In fact, since 2002, the graphics pipeline implemented in GPU hardware became programmable through small programs called shaders. Programming languages such as Cg, GLSL, DirectX Shading languages, etc. are employed for shaders. There were two types of shaders (for vertices and fragments), but in 2007 they were unified (e.g., the U of CUDA stands for Unified).

The evolution of shaders led to a new area called GPGPU (general purpose computing on the GPU), whose name was coined by Mark Harris in 2002. The target of this research area is to develop parallel methodologies to program GPUs for other purposes rather than graphics, such as scientific computing. Today, this is more known as GPU computing and has been settled as an alternative within HPC. That is, GPUs are nowadays an HPC accelerator that can be found in the most powerful supercomputers. GPUs are good at data parallelism. More specifically, they are based on SPMD programming model (Single Program Multiple Data): the GPU processes many independent elements in parallel using the same program [35].

Currently, GPU computing is also a heterogeneous computing system [17], where the GPU is known as device and the CPU is known as host. The host has a role of a master, which takes over the execution and manages the different devices that can be in the system. Devices are co-processors that help to accelerate the algorithms by executing code in a parallel fashion, reducing in this way the overhead on the host. This trend is being consolidated with OpenCL [31], the first free, open standard for multi-platform, heterogeneous parallel programming of modern processors found in PCs, servers, and embedded devices. OpenCL is being used not only for GPUs but also for FPGAs, multicore CPUs, etc. However, the drivers and compilers developed by each manufacturer of chips (NVIDIA, Intel, AMD...) are not up to date and lack full support. This is why a new standard, called SYCL, is being conceived, but it is still experimental (to date).

GPUs can be programmed with both OpenCL and SYCL [44]. Moreover, NVIDIA GPUs can be also programmed with CUDA [32], which is a proprietary technology that is very mature and has lot of functionality. By having a quick look to the literature, it is possible to see that CUDA is the most used platform for GPU computing. On the other side, AMD GPUs can be also programmed with a CUDA-like environment called HIP [34], which is based on RoCm. This allows programmers to translate easily CUDA code to AMD technology. Other standard languages and platforms to program GPUs are based on the graphics pipeline, such as OpenGL, GLSL, and Vulkan [47]. They can also be used for GPU computing in a not very complex way.

In short, GPU computing poses a highly parallel architecture with thousands of lightweight processor cores and high-bandwidth memory that can be programmed with several standard languages. The most evolved one is CUDA, but it works only for NVIDIA GPUs. Since the introduction of CUDA in 2007, many scientific applications have used GPU computing. Their low cost compared to the perfor-

mance offered has made GPUs an attractive alternative. In fact, currently they are the enabling technology (i.e., if you want to tackle these problems, you should be using GPUs) for Deep Learning [18], nanopore DNA sequencing, and high-energy particle trajectory reconstruction in LHCb HLT 1 TODO:CITE PAPER DANIEL CAMPORA.

6.2.2 CUDA Programming Model

In this chapter, we will be focusing on CUDA [32, 33], since, as mentioned, it is the most widely technology for GPU computing, and on top of that, the majority of P system simulators have been developed with CUDA. Let us recall that CUDA, as usual in GPU computing, offers a heterogeneous system to the user, where the CPU is known as the host, and the GPU is the device. The execution flow in a CUDA program is like in any common program; it starts with the CPU main function. At some points, the CPU asks the GPU to allocate memory, transfer memory, launch computation, retrieve results, etc.

CUDA devices take advantage of data-parallel program sections and accelerate their execution. 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 also a unified source code covering both sides.

We call kernels to those functions executed by the device (GPU). When they are requested to be executed by the host, they allocate an execution grid on device. A grid typically populates a large number of execution threads that work in SPMD fashion: they execute the same piece of code (the kernel function) to probably different portions of data. Actually, a kernel is written as a usual function in a programming language (so far, only C++, Fortran or Python) but using special keywords given by the CUDA API. These keywords, such as the thread identifier, might take different values at different threads in run time, so that they can have index different data elements, or even take different execution path (although this is not optimal). 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.

Threads within a grid are arranged in a two-level hierarchy. At the higher level, each grid consists of a two-dimensional array of *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. Moreover, each block is identified by a two-dimensional identifier and each thread within its block by a three-dimensional identifier. To date, a thread block can contain, at most, 1024 threads. Threads within a block can easily cooperate through a special fast memory (see below) and special warp-wide operations (see next section) and be synchronized with a barrier operation.

One kernel is executed by just one grid that, has mentioned, arranges an array of thread blocks, each with the same configuration of threads. It is also possible to launch several kernels at the same time on a GPU with different grids. This is allowed by the so-called CUDA streams. They are concepts similar to lanes, where kernels get executed. As long as there are available resources on the GPU, the kernel executions can be done simultaneously. CUDA streams also allow to overlap execution with memory transfers, designing in this way full computing pipelines.

CUDA programmers also have to explicitly manage the memory layout and hierarchy. GPUs offer different memory spaces arranged in a hierarchy where to store the data of a parallel program. In CUDA, the host and the devices have separate memory spaces (as it is in the real hardware). In order to run a kernel on the device, the data has to be there. Hence, enough memory should be first allocated, and later the relevant data has to be transferred from host to device. Similarly, after the execution of the kernels on the device, the resulting data has to be copied from device to host memory, and finally the allocated device memory should be released. From this point of view, we can assume that CUDA uses static memory allocations. Dynamic memory is already supported at the kernels, but there are restrictions, and it drastically downgrades the performance.

The memory that serves as communication channel between the host and the device is called global (or device) memory. The host can allocate memory and copy memory, and CUDA threads can access it and make modifications. However, this memory is the slowest in the GPU, but the largest one. The best performance is achieved when contiguous threads (according to their identifiers) access contiguous positions of data, in what is called coalesced memory access. This helps to maximize the utilization of the memory bandwidth.

Threads can use a common memory space when they are in the same thread block, which is called shared memory. Accesses to shared memory are very fast when done in a coalesced way. However, it is a small space of up to just kilobytes. On the other side, we can find cached memories. They are memories that automatically speedup the access to repeated data through a cache. Examples of them are constant and texture memories. They are read-only memories for the GPU, and the CPU can just copy data in there, under certain restrictions. Moreover, modern GPUs have two levels of cache memory for accesses to global memory, but this is completely transparent to the code.

In summary, algorithms implemented in CUDA are structured as follows [17]:

1. The host initializes the program, reading the input data.
2. The host allocate enough memory space in global memory for input and auxiliary data.
3. The host copies the input and auxiliary data to the device.
4. The host launches a kernel to the device, with the following syntax: `kernelName <<< numBlocks, numThreads, streamId, sharedMemory >>> (param1, param2,...)`

5. When a kernel is executed, at the device side:
   (a) The threads of each block read its corresponding data portion from global memory to shared memory or to internal variables (also called registers, see next section).
   (b) Threads work with the data directly on the shared memory or with their registers.
   (c) Threads copy these data back to global memory.
6. The host might call more kernels, copy more data, retrieve data, etc.
7. When the algorithm is done, the host copies back the results from global memory of the device.

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.

6.2.3 GPU Architecture

A modern GPU architecture [19] consists of a processor array which has hundreds (even thousands) of *SP* (streaming processor) cores organized in *SM* (streaming multiprocessor). In this sense, every SM contains the following units: SP arithmetic cores, SFU single-precision floating point units (for specific operations such as sine, cosine, reciprocal square root, etc.), double precision units, instruction cache, read only constant cache, read/write shared memory and L1 cache memory, a set of 32-bit registers, and access to the off-chip memory (device/local memory). The arithmetic units are capable to execute several instructions per clock cycle, and they are fully pipelined, running at frequencies around 1 GHz (depending on the GPU). The amount of cores, floating point units, shared memory, etc. depends on the GPU itself.

SMs is able to manage and execute thousands of threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. This execution is done in a *SIMT (Single-Instruction Multiple-Thread)* fashion [19], where threads execute the same instruction on different piece of data. SMs create, manage, schedule, and execute threads in groups of 32 threads (of the same thread block). This set of 32 threads is called *warp*. Each SM can handle several warps. Individual threads of the same warp must be of the same type and start together at the same program address in order to be scheduled simultaneously, but they are free to branch (e.g., an `if then else` clause) and execute independently at the cost of serialization.

When a grid is created to execute a kernel, the thread blocks are created and assigned to SMs. An SM can handle several thread blocks, but a thread block is assigned only to one SM. Then, the thread block is split into warps and they are scheduled. When a warp is selected, its threads are executed on SPs as long as the

threads are synchronized in the same execution flow of the code. 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.

As shown by the CUDA programming model [17], the GPU contains several memory spaces. First of all, both the GPU and the CPU have separated memory spaces. They are connected through a bus that can be PCI Express $\times 16$ bus standard. Global memory is the largest (up to several gigabytes) but the slowest one. Although there is a two-level cache memory system to speedup repeated access to same data, an access to global memory is around 400 times slower than accessing on-chip memory spaces (such as shared memory or registers). Moreover, in order to fulfill the memory bandwidth, threads should make coalesced accesses.

As mentioned, inside each SM we can find a memory space called shared memory. Its size is measured in kilobytes, but its access is very fast, even close to the accesses to registers. This memory space is also split into shared memory and L1 cache memory. The latter is transparent to CUDA programs, while shared memory is manually managed (one can allocate space and let threads to copy and modify data). There are also many other units for cached memory which is read only for the cores. Finally, SMs incorporate a large amount of registers, whose access is the fastest since they are next to the cores. They are used to allocate the values of single variables declared in the code (e.g., iterators, auxiliary variables, etc.).

6.2.4 Good Practices

CUDA is supported by a wide range of tools [33], including a compiler (called nvcc), the driver for the GPU, libraries, and examples. They are freely available at their website. There is also a vast amount of documentation, books, and literature in this respect. CUDA is not only the most mature platform for GPU computing but also the one with the largest community and support. It is important to know the compiler options for automatic optimizations (like -O3) and to understand and use the libraries (e.g., CuRAND for random number generation, CuSPARSE for sparse matrix representations and operations, etc.).

It is also a good practice to start developing a reference program in sequential C/C++ before starting implementing in CUDA. This is critical in order to first understand the algorithm, secondly to validate the parallel version, and also to run benchmarks and performance analysis.

Finally, let us introduce four ways to accelerate the execution of a program on a GPU with CUDA [35]. They will help to understand the designs of GPU-based P system simulators:

- *Emphasize parallelism*: GPUs prefer to run thousands of lightweight threads. Thus, the algorithms should permit dividing the computation into many independent pieces by decreasing the resources assigned to each thread and avoiding synchronization.

- *Minimize branch divergence*: if a warp is broken because divergence in the path executed by the threads, then there is no real parallelism.
- *Maximize arithmetic intensity*: computation is relatively cheap for today's GPUs, but bandwidth is precious. It is better to maximize the computational operations per memory transaction. Shared memory or registers can help for this purpose.
- *Exploit streaming bandwidth*: on the other side, GPUs and their on-board memory have a peak bandwidth much faster than in CPUs. 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.

6.3 Generic Simulations

In this section, we will introduce a type of simulation of P systems, which is called generic simulation [22, 25]. We will describe how to implement this kind of simulators in CUDA and provide two illustrative examples.

6.3.1 Definition

When implementing a P system simulator, it is important to understand what type of P systems we want to simulate before starting the development. We will say that a generic simulator is a simulator developed for a wide range of P systems belonging to the same variant. If the simulator is able to handle a large variety of P systems (with very distinct rules and alphabets, even designed to solve different problems), then it is generic. Sometimes the types of P systems are restricted somehow for the sake of simplicity, but as long as the simulator accepts P systems from different families (but for the same variant), we will say it is generic and not specific.

In this scenario, it is not possible to know what can happen in the computation at a certain transition step. Therefore, it has to be prepared for any situation, so we need to cover worst-case scenarios when developing such kind of simulators. For example, we need to provide an upper bound of existing objects at a certain step, in order to avoid memory overflows. Furthermore, in principle, all rules might be selected for execution at a certain step (until their applicability is checked). The rules must be stored in memory since we do not know them until the P system model is parsed. As mentioned in Chap. 2, simulators are usually defined by three modules: input parser, simulation engine, and output module. Generic simulators can be designed to reproduce either a single computation or all computations of the input P system.

The memory layout is also an essential part of a simulator, since P system simulators have been demonstrated to be memory and memory bandwidth bound. When storing the information of P system configuration, we can use either [25]:

- *Sparse representation*: using a large array to store multiplicities, with a position per each possible object (all objects defined in the alphabet). The access is direct since the object identifier is the index where to access the array. However, if many objects are not present at a certain moment, the array will be full of zeroes.
- *Dense representation*: using a double array with a component for the object identifier and the other for multiplicity. We need to search for the object, unless we track them and we know exactly where they are store at any moment. This can help to drastically reduce the size because objects with multiplicity zero can be discarded.

Generic simulators usually use sparse representations, since, in this way, they can identify objects very efficiently, in $O(1)$. The object identifier is employed as the index to access the array where storing the multisets, and by representing the whole alphabet, we make sure of an upper bound for the worst-case scenario.

6.3.2 Simulating P Systems with Active Membranes

In this section we will depict the very first GPU simulator for P systems ever developed. It was a generic simulator and helped to understand how to better map the parallelism of P systems on the parallelism on GPUs. The simulated models were of the variant P systems with active membranes and elementary division.

The original work is published in [4, 21, 22]. 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 official website http://sourceforge.net/p/pmcgpu [45] or the repository https://github.com/RGNC/pcuda.

6.3.2.1 Recognizer P Systems with Active Membranes

Families of cell-like P systems whose membrane structures does not grow, that is, there is no rules producing new membranes in the system, only can solve in polynomial time and uniform way, problems in class **P**. Therefore, new ingredients are needed in order to be able to provide efficient solutions of computationally hard problems by making use of an exponential workspace, expressed in terms of number of membranes and number of objects, created in linear time. In [36], a new computing model, called *P system with active membranes*, was introduced. In these systems, the membranes have associated with electrical charges and make use of *division rules*, inspired from the *mitosis* and *meiosis* processes, as a mechanism to generate in linear time, an exponential workspace. Polynomial time and uniform solutions to **NP**-complete problems were given by using families of the new computing model.

Next, P systems with active membranes and division rules only for elementary division are formally defined.

Definition 6.1. 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 and H are finite alphabets such that $\Gamma \cap H = \emptyset$;
2. μ is a rooted tree with q nodes labeled by elements from H (the root is labeled by $i_s \in H$);
3. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ belongs to $M(\Gamma)$, that is, all of them are multisets over Γ;
4. $\mathcal{R}$ is a finite set of rules, of the following forms:
   (a) $[\, a \rightarrow u \,]_h^\alpha$, for $h \in H$, $\alpha \in \{+, -, 0\}$, $a \in \Gamma$, $u \in M(\Gamma)$ (object evolution rules);
   (b) $a\,[\;\;]_h^{\alpha_1} \rightarrow [\, b \,]_h^{\alpha_2}$, for $h \in H \setminus \{i_s\}$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$ (send-in rules);
   (c) $[\, a \,]_h^{\alpha_1} \rightarrow b\,[\;\;]_h^{\alpha_2}$, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$ (send-out rules);
   (d) $[\, a \,]_h^{\alpha} \rightarrow b$, for $h \in H \setminus \{i_s, i_{out}\}$, $\alpha \in \{+, -, 0\}$, $a, b \in \Gamma$ (dissolution rules);
   (e) $[\, a \,]_h^{\alpha_1} \rightarrow [\, b \,]_h^{\alpha_2}\,[\, c \,]_h^{\alpha_3}$, for $h \in H \setminus \{i_s, i_{out}\}$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $a, b, c \in \Gamma$ (division rules for elementary membranes);
5. $i_{out} \in H \cup \{env\}$, where $env \notin \Gamma \cup H$.

A P system with active membranes of degree $q \geq 1$, $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, can be viewed as a set of q membranes, injectively labeled by elements of H, arranged in a hierarchical structure μ given by a labeled rooted tree (called membrane structure) whose root is called the *skin membrane* (labeled by i_s), such that (a) each membrane has associated with an electrical charge from the set $\{+, -, 0\}$; (b) $\mathcal{M}_1, \ldots, \mathcal{M}_p$ represent the finite multisets of *objects* (symbols of the working alphabet Γ) initially placed in the q membranes of the system; (c) $\mathcal{R}$ is a finite set of rules over Γ associated with the labels; and (d) $i_{out} \in H \cup \{env\}$ indicates the output zone. We use the term *zone* i to refer to membrane i in the case $i \in H$ and to refer to the "environment" of the system in the case $i = env$. The leaves of μ are called *elementary membranes*.

Next, the semantics of the new computing model is described. A *configuration* (or *instantaneous description*) C_t at an instant t of a P system with active membranes is described by the following elements: (a) the membrane structure at instant t and (b) all multisets of objects over Γ associated with all the membranes present in the system at that moment.

An *object evolution rule* $[\, a \rightarrow u \,]_h^\alpha$ is *applicable* to a configuration C_t at an instant t, if there exists a membrane labeled by h with electrical charge α, in C_t, such that contains object a. When applying such a rule to such a membrane, one object a is consumed and objects from the multiset u is produced in that membrane.

A *send-in communication rule* $a\,[\;\;]_h^{\alpha_1} \rightarrow [\, b \,]_h^{\alpha_2}$ is *applicable* to a configuration C_t at an instant t, if there exists a membrane labeled by h with electrical charge α_1, in C_t such that h is not the label of the root of μ and its parent membrane contains object a. When applying such a rule to such a membrane, one object a is consumed

from the parent membrane, and object b is produced in the corresponding membrane labeled by h. Besides, the charge α_1 of that membrane h is replaced by α_2.

A *send-out communication rule* $[\, a \,]_h^{\alpha_1} \rightarrow b\, [\ ]_h^{\alpha_2}$ is *applicable* to a configuration C_t at an instant t, if there exists a membrane labeled by h with electrical charge α_1, in C_t such that it contains object a. When applying such a rule to such a membrane, one object a is consumed from such membrane h, and object b is produced in the parent of such membrane (in the case that such membrane is the skin, then object b is produced in the environment). Besides, the charge α_1 of that membrane h is replaced by α_2.

A *dissolution rule* $[\, a \,]_h^{\alpha} \rightarrow b$ is *applicable* to a configuration C_t at an instant t, if there exists a membrane labeled by h with electrical charge α, in C_t, different from the skin membrane and the output zone, such that it contains object a. When applying such a rule to such a membrane, one object a is consumed, membrane h is dissolved, and one object b and the remaining objects of the membrane where the rule is applied are sent to its parent (or the first ancestor that has not been dissolved).

A *division rule for elementary membrane* $[\, a \,]_h^{\alpha_1} \rightarrow [\, b \,]_h^{\alpha_2}\, [\, c \,]_h^{\alpha_3}$ is *applicable* to a configuration C_t at an instant t, if there exists an elementary membrane labeled by h with electrical charge α_1, in C_t, different from the skin membrane and the output zone, such that it contains object a. When applying such a rule to such a membrane, the membrane with label h is divided into two membranes with the same label; in the first copy, one object a is replaced by one object b; in the second one, one object a is replaced by one object c; all the other objects are replicated and copies of them are placed in the two new membranes. Besides, the charge α_1 of the first created membrane h is replaced by α_2, and the charge α_1 of the second created membrane h is replaced by α_3.

In P systems with active membranes, the rules are applied according to the following principles:

- At one transition step: (i) one object and one membrane can be used by only one rule, selected in a non-deterministic way, and (ii) at most a rule of types (b)–(e), selected in a non-deterministic way, can be applied to a membrane, and then it is applied once.
- Object evolution rules can be simultaneously applied to a membrane with one rule of types (b)–(e). If it is the case, object evolution rules will be applied in a maximally parallel manner.
- If an object evolution rule and a division rule are applied to a membrane at the same transition step, then we suppose that first the evolution rule is applied, and then the division is produced. Of course, this process takes only one transition step.
- The skin membrane and the output membrane, if any, can never get divided nor dissolved.

Given a P system with active membranes, $\Pi = (\Gamma, H, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, the *initial configuration* of Π is $C_0 = (\mathcal{M}_1, \cdots, \mathcal{M}_q)$. A configuration is a *halting configuration* if no rule of the system is applicable to it. We say that configuration

C_1 yields configuration C_2 in one *transition step*, denoted $C_1 \Rightarrow_\Pi C_2$, if we can pass from C_1 to 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 in one transition step
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 zone i_{out} in the halting configuration.

Let us notice that these P systems have some important features: (a) They use three electrical charges; (b) the polarization of a membrane can be modified by the application of a rule; (c) the label of a membrane cannot be modified by the application of a rule; and (d) they do not use cooperation neither priorities.

Decision problems are associated with languages in such manner that *solving* a decision problem is defined by *recognizing* the language associated with it. For that, *recognizer membrane systems* were introduced in [41] (called *decision P systems*), and complexity classes associated with these systems were introduced in [40]. Over the last few years, the previous methodology for addressing the **P** *versus* **NP** problem has been applied in the framework of *Membrane Computing*.

A computing model in the paradigm of Membrane Computing (generically called *membrane system*) is said to be a *recognizer system* if it has the following syntactic and semantic peculiarities: (a) the working alphabet has two distinguished objects (`yes` and `no`); (b) there exist an input alphabet strictly contained in the working alphabet and an *input membrane*; (c) the initial content of each compartment is a multiset of objects from the working alphabet not belonging to the input alphabet; (d) all computations of the system are halt; and (e) for each computation, either object `yes` or object `no` (but not both) must have been released to the environment and only at its last step. Recognizer membrane systems have the ability to accept or reject multisets over the input alphabet. Specifically, given a recognizer membrane system Π, for each multiset m over the input alphabet, a new initial configuration is obtained by adding the multiset m to the content of the input compartment at the initial configuration of Π (the system Π with this new initial configuration associated with m is denoted by $\Pi + m$). Then, we say that system Π *accepts* (respectively, *reject*) the input multiset m if and only if all computations of the system $\Pi + m$ answer `yes` (resp. `no`).

Unlike a Turing machine where there is an infinite tape, all the elements that make up a recognizer membrane system have a finite description. Therefore, while a decision problem (with an infinite set of instances) can be solved by a single Turing machine, an infinite family of recognizer membrane systems is necessary to solve it.

Following [40], we say that a family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ of recognizer membrane systems solves a decision problem X in *polynomial time and uniform*

way if the following holds: (i) the family $\mathbf{\Pi}$ can be generated by a deterministic Turing machine working in polynomial time; and (ii) there exists a pair (cod, s) of polynomial-time computable functions (over the set of instances of 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 $n \in \mathbb{N}$, the set $s^{-1}(\{n\})$ is a finite set; and (c) the family $\mathbf{\Pi}$ is *polynomially bounded*, *sound*, and *complete* with regard to (X, cod, s) (see [40] for details).

Given a computing model $\mathcal{R}$ of recognizer membrane systems, $\mathbf{PMC}_{\mathcal{R}}$ denotes the set of decision problems solvable by families from $\mathcal{R}$ in polynomial time and uniform way. This complexity class is closed under complement and under polynomial-time reduction [40]. Hence, if X is a complete problem for a complexity class $\mathcal{K}$ and $X \in \mathbf{PMC}_{\mathcal{R}}$, then we deduce that $\mathcal{K} \cup \mathbf{co}\text{-}\mathcal{K} \subseteq \mathbf{PMC}_{\mathcal{R}}$.

P systems with active membranes (without dissolution and using division rules only for elementary membranes) have been successfully used to design polynomial time solutions to (weak and strong) **NP**-complete problems (e.g., `SAT` [42], `Subset Sum` [38], `Knapsack` [39], `Partition` [14], etc.). It is important to note that some of these solutions only make use of two polarizations in their design.

6.3.2.2 Simulation Algorithm

The simulator is based on the sequential simulator for P systems with active membranes provided in pLinguaCore [13]. In this design, the simulation process is a loop divided into two stages: *selection stage* and *execution stage*. 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, since it is possible to have more than one selectable rule but only one can be executed. For example, two evolution rules 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. 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, the simulator uses the two stages and a temporary data structure to store information of the selection of rules.

6.3.2.3 Sequential Simulator

As previously mentioned, CUDA programming model [33] is based on the C/C++ language [16]. 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 [13]. This sequential (or single-threaded) simulator is programmed in JAVA, so the first step was to translate the code to C++.

The first version of the 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 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. However, we found that this drastically slowdown the simulation, since objects get created and consumed at every step, and hence, we are continually allocating and destroying memory, what is very time consuming.

Therefore, a simulator using static structures that get allocated at the beginning of the simulator was developed and shown to be 160 times faster than the first version [21]. These structures are the same also for the parallel simulator, so they are replicated at both sides in order to achieve fair comparisons. In summary, the memory layout to represent the P system is based on the following data structures:

- *Multisets*: an array storing the multisets of the objects using a sparse representation. Since, for simplicity, it is assumed that the simulated P system can contain only two levels in the memory hierarchy (a skin membrane and elementary membranes), the representation of the environment, skin, and elementary membranes are separated. The amount of elementary membranes is set initially by the user.
- *Charges*: an array storing the charge of each membrane.
- *Rule sets*: an array storing rules information. It is indexed by using a membrane label, a charge, and finally an object index. Given that it is possible to have more than one rule associated to the same object, and assuming that the P system is confluent, only one rule of each type is stored.

One major problem to overcome 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 the simulator. The format is depicted in [21]. pLinguaCore 2.0 [13] is able to translate a P system written in P-Lingua language into a binary file. First, we define the P system into P-Lingua. pLinguaCore translates 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 one provided in pLinguaCore.

6.3.2.4 Parallel Simulation on CUDA

Whenever we design algorithms in the CUDA programming model, the 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 [43]. Each thread block accesses to one different set of input data and assigns a single or small constant number of input elements to each thread.

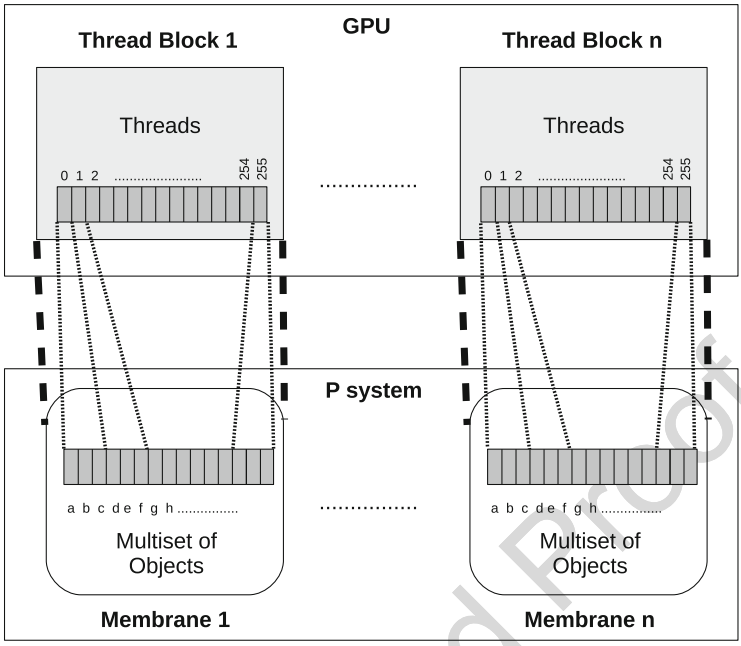

Fig. 6.1 Basic design of the parallel simulator on the GPU. From [4, 21]

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 the designs requires successive kernel calls even to the same kernel.

Figure 6.1 shows the overall design of the simulator on the GPU [4]. Thread blocks and threads are distributed as follows. Each membrane of the simulated P system is attributed to each thread block. In this way, the parallelism between membranes by using the parallelism between thread blocks is identified. However, this is tricky. 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) [40].

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 [21]. 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, and send-in).

Algorithm 2 shows the pseudo-code of the simulator. First of all, the data needed for the computation is moved 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 explained before, this process iterates until the maximum number of steps is reached or the system returns an answer. Finally, the result data is copied back to the CPU.

6.3.2.5 Performance Comparative Analysis

In this section, the performance of the developed simulators is compared. This is done by a very simple example, with the aim of studying the behavior of the CUDA kernels. In order to evaluate the performance of the simulator, a family of P systems was designed, named test P system, where it is easy to vary the number of membranes as well as the number of objects [4]. 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

Algorithm 2: Parallel simulator of P systems on the GPU, from [21]

```
configuration ← initialConfiguration
selectedRules ← ∅
step ← 0
isFinalConfiguration ← false
CopyDataFromCPUtoGPU(configuration)
CopyDataFromCPUtoGPU(rules)
while step < maxStep ∧ NOT isFinalConfiguration do
    kernelSelection(rules,configuration,selectedRules)
    if DISSOLUTION ∈ selectedRules then
        kernelDissolution(rules,configuration,selectedRules)
    end if
    if DIVISION ∈ selectedRules then
        kernelDivision(rules,configuration,selectedRules)
    end if
    if SEND-OUT ∈ selectedRules then
        kernelSendOut(rules,configuration,selectedRules)
    end if
    if SEND-IN ∈ selectedRules then
        kernelSendIn(rules,configuration,selectedRules)
    end if
    step ← step + 1
    isFinalConfiguration ← checkFinalConfiguration(configuration)
end while
CopyDataFromGPUtoCPU(configuration)
```


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.

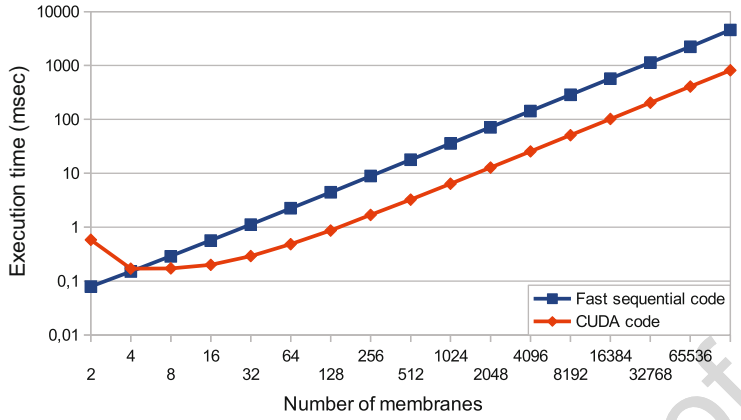

Fig. 6.2 Comparing the execution time for one step of the fast sequential and parallel simulators, by increasing the number of membranes in the system and using a total of 2560 objects in the alphabet. From [4, 21]

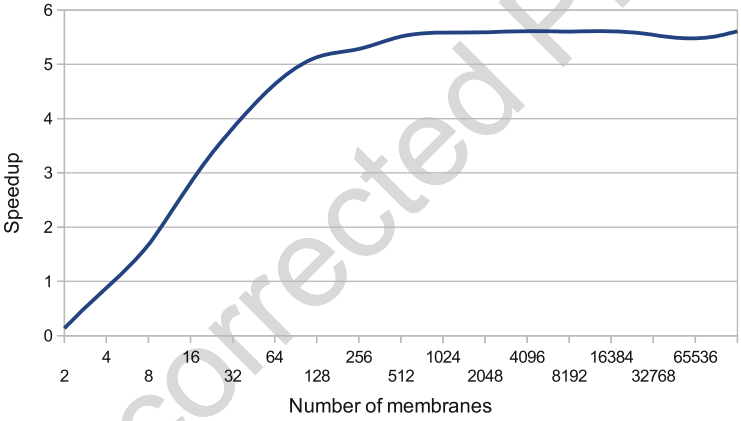

Fig. 6.3 Comparing the speedup for one step of the fast sequential and parallel simulators, by increasing the number of membranes in the system and using a total of 2560 objects in the alphabet. From [4, 21]

Figures 6.2 and 6.3 show the results obtained for the parallel simulator versus the sequential version. Notice that in both graphs the Y-axis is also represented in a logarithmic form. The benchmark covers the parallelism between membranes by exponentially increasing the number of membranes. It can be seen that the CPU simulator also increases its time exponentially from the beginning (with four membranes) until reaching the final configuration (with 32768 membranes). The 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 5.7×, and this difference enlarges with the number of membranes (from 1024), because the resources of the GPU are fully utilized.

6.3.3 Simulating Population Dynamics P Systems

In this section, a simulator for Population Dynamics P (PDP) systems is revisited. It is a generic simulator implementing the DCBA algorithm for PDP systems.

The original work is published in [21, 22, 24, 28]. The framework of generic simulators for PDP systems on GPU is called *ABCDGPU*. It is a subproject of the PMCGPU project and can be downloaded from the official website http://sourceforge.net/p/pmcgpu [45] or the repository https://github.com/RGNC/abcd-gpu.

6.3.3.1 Population Dynamics P Systems

Population Dynamics P systems are a variant of *multienvironment P systems with active membranes* [6–8]. The model consists of a directed graph of environments, each of them containing a P system where electrical charges are associated with membranes. 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 6.2. 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 $V = \{e_1, \ldots, e_m\}$.
- 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 $(x)_{e_j} \overrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$, where $x, y_1, \ldots, y_h \in \Sigma$, $(e_j, e_{j_l}) \in S, 1 \leq l \leq h$, and $p_r : \{1, \ldots, T\} \longrightarrow [0, 1]$ 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 $(x)_{e_j} \overrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ 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 $(x)_{e_j} \overrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ and $u[v]_i^{\alpha} \rightarrow u'[v']_i^{\alpha'}$ having $x \in u$.

- For each $r \in \mathcal{R}$ and $1 \leq j \leq m$, $f_{r,j} : \{1, \ldots, T\} \longrightarrow [0, 1]$ 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 $e_1, \ldots, e_m$ 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, $\Pi_j = (\Gamma, \mu, \mathcal{M}_{1,j}, \ldots, \mathcal{M}_{q,j}, \mathcal{R})$, 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 $1, \ldots, q$. The skin membrane (the root of the tree) is labeled by 1. We also associate electrical charges from the set $\{0, +, -\}$ with membranes.
(c) $\mathcal{M}_{1,j}, \ldots, \mathcal{M}_{q,j}$ 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 provides 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 $(x)_{e_j} \overrightarrow{p_r} (y_1)_{e_{j_1}} \ldots (y_h)_{e_{j_h}}$, is applicable to the environment e_j if it contains object x. When such rule is applied, object x passes from e_j to $e_{j_1}, \ldots, e_{j_h}$ possibly modified into objects $y_1, \ldots, y_h$ 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 $(x)_{e_j} \overrightarrow{p_r} (y_1)_{e_{j_1}} \ldots (y_h)_{e_{j_h}}$, 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.

6.3.3.2 Simulation Algorithm

The simulation algorithms for PDP systems called BBB and DCBA [21, 27] are based on the grouping of rules into blocks. These groups are constructed by looking the left-hand side. Note that rules having the same left-hand side must have associated probabilities summing 1. Specifically, DCBA works using a refined definition of block, called consistent block [21, 27], as shown in Definition 6.3. DNDP [21, 30] does not use the concept of blocks, but it selects rules by a random loop instead.

Definition 6.3. Rules from R and R_E are classified into consistent blocks by either of the following:

(a) The rule block associated with $(i, \alpha, \alpha', u, v)$ is $B_{i,\alpha,\alpha',u,v} = \{r \in R : LHS(r) = (i, \alpha, u, v) \wedge charge(RHS(r)) = \alpha'\}$
(b) The rule block associated with (e_j, x) is $B_{e_j,x} = \{r \in R_E : LHS(r) = (e_j, x)\}$.

The selection of rules in BBB and DCBA relies always first on selecting blocks, calculating a multinomial random variate, and therefore obtaining a selection of rules within each block. In this sense, we can say that rules within a block will not compete among objects when using BBB and DCBA, because they are selected altogether. This, again, does not hold in DNDP, where rules are selected individually according to the probabilities. Block competition takes place whenever two blocks have distinct but overlapping left-hand sides.

DCBA tackles the resource competition issue by performing a proportional distribution of objects among competing blocks. This is done by using the *distribution table*, which is a system-wide time having blocks per columns and pairs (object,region) per rows. Algorithm 3 shows a summary of the algorithm, which can be depicted in [27]. It can be seen that, as usual, each loop iteration is made by two stages: selection and execution. Selection stage consists of three phases: phase 1 distributes objects to the blocks in a certain proportional way, phase 2 ensures *maximality* by checking the maximal number of applications of each block, and phase 3 translates from block to rule applications by calculating random numbers using a multinomial distribution. Finally, execution stage (or phase 4) generates the right-hand side of rules.

Algorithm 3: Sketch of DCBA algorithm for PDP systems

```
1: Initialization of the algorithm: static distribution table (columns: blocks, Rows:
   (objects,membrane))
2: for t ← 0 . . . T do
3:     Selection stage:
4:         Phase 1 (Distribution of objects among rule blocks)
5:         Phase 2 (Maximality selection of rule blocks)
6:         Phase 3 (Probabilistic distribution, blocks to rules)
7:     Execution stage
8: end for
```

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

Finally, DCBA also handles the consistency of rules by defining the concept of consistent blocks [21, 27]: rules within a block have the same left-hand side and the same charge in the right-hand side. There is a further restriction within phase 1: if two non-consistent blocks (having different associated right-hand charge) can be selected in a configuration, the simulation algorithm will return an error, or optionally non-deterministically choose a subset of consistent blocks.

6.3.3.3 Design of the Parallel Simulator

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, which is easily parallelized.

At first glance, these two levels of parallelism (simulations and environments [23]) 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.

Thus, the selection of rule blocks (phase 1) and rules (phase 2 and 3) is further parallelized. Hence, the 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), 256 threads are created, which iterate over the rule blocks in tiles. This design is graphically shown on Fig. 6.4. 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.

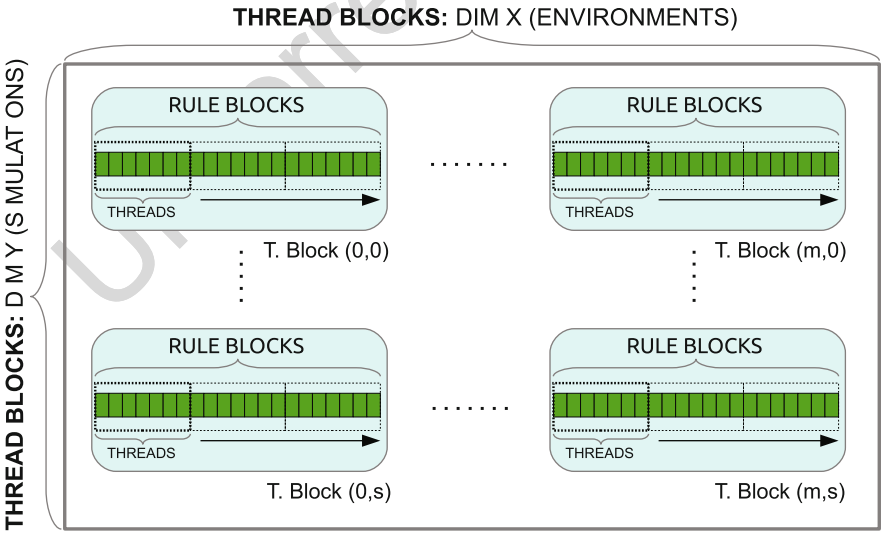

Fig. 6.4 General design of the CUDA-based simulator: 2D grid and 1D thread blocks. Threads loop the rule blocks in tiles. From [21, 28]

6.3.3.4 GPU Implementation of the DCBA Phases

The main challenge at phase 1 is the expanded static table $\mathcal{T}_j$ construction. The size of this table is of order $O(|B| \cdot |\Gamma| \cdot (q+1))$, where $|B|$ is the number of rule blocks, $|\Gamma|$ is the size of the alphabet (total 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 systems and very sparse: competitions for one object appears for a relatively small number of blocks. Thus the expanded static table is implemented by a *virtual table*, which is similar but based on the information of the rules:

- Operations over columns: they can be transformed to operations for each rule block and their left-hand sides.
- Operations over rows: they can be transformed to operations over the left-hand sides of rule blocks and storing the partial results into a global array (one position per row).

Further auxiliary data structures are used to virtually simulate the table [23]:

- $activationVector$: the information of filtered blocks is stored here as Boolean values.
- $addition$: the total sums of the rows are stored using this global vector, one per each pair object and region.
- $MinN$: the minimum numbers per column are stored here.
- $BlockSel$: the total number of applications for each block is stored here.
- $RuleSel$: the total number of applications for each rule is stored here.

The implementation of phase 1 is actually done by means of three kernels, executing one after the other and using the same grid configuration as mentioned in Fig. 6.4. The second and third kernels are executed several times according to parameter A (accuracy) of DCBA [28]:

1. Kernel for Filters: FILTERS 1 and 2 are implemented here.
2. Kernel for Normalization: the two parts (row additions and minimum calculations) of the normalization step is implemented in a kernel. The two parts are synchronized by *synchtreads* CUDA instruction. The work assigned to threads is divergent, 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.
3. Kernel for Updating and FILTER 2. As before, the work of each thread is divergent. Thus, the update of the configuration is also implemented with atomic operations. Moreover, the $BlockSel$ gets updated with the new distribution of selection.

Phase 2 is the most challenging part when parallelizing by blocks. The selection of blocks is performed in an inherently sequential way: we need to know how much a block can consume before checking the next one. At least, phase 2 can be run simultaneously to each environment and simulation. For this phase, a special version of this kernel was designed. This kernel dynamically calculates the competition of blocks, so that the dependencies of blocks are pre-calculated in order to know which blocks can be selected independently of each other, and everything is done in shared memory. $BlockSel$ gets updated with the last selections, and the configuration is also updated to prevent other blocks from being selected.

Phase 3 requires a random number generation system for multinomial distributions that was not existing for CUDA. A dedicated implementation was done, called *CURAND_BINOMIAL*, and it is based on the accelerated uniform and normal random variate generation in CUDA with *CuRAND* and the BINV algorithm. This is therefore used to calculate multinomial distributions per rule block and write to $RuleSel$.

Finally, for phase 4, the rule selection $RuleSel$ is used to generate the right-hand side, by using atomic operations over the configuration. The parallelization is done by using a similar grid configuration as shown in Fig. 6.4, but looping over rules instead of rule blocks.

6.3.3.5 Performance Results of the Simulator

In order to test the performance of the simulators, a random generator of PDP systems was designed (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.

The parallel simulator on the GPU (*pdp-gpu-sim*) and a parallel simulator on multicore CPUs (*pdp-omp-sim*, for 1 (sequential), 2 and 4 cores) are compared. All experiments were run on a 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.

The 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 Fig. 6.5. 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 $7\times$ 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 $4.3\times$ and $3\times$, respectively. In the experiments, this number is also achieved when running with 10^6 rule blocks.

As stated in [23], parallelizing by simulations yields the largest speedups on multicore platforms. In order to test the efficiency of the simulator when

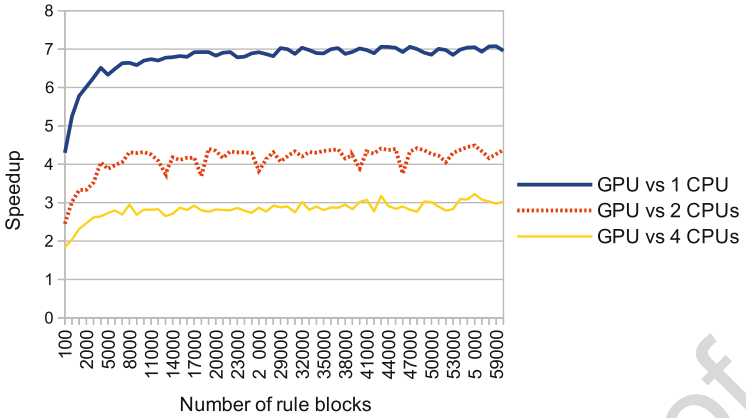

Fig. 6.5 Scalability of the simulators when increasing the number of rule blocks, from [21, 28]

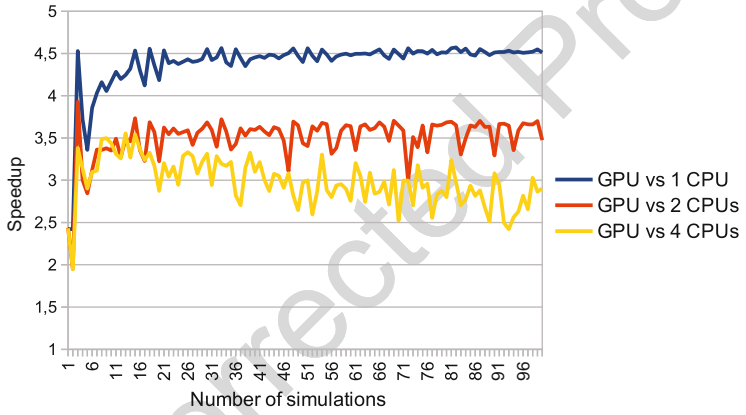

Fig. 6.6 Scalability of the simulators when increasing the number of simulations, from [21, 28]

increasing the number of simulations, rule blocks are fixed to 50000, environments to 20, objects to 5000, and mean LHS length to 1.5. As shown in Fig. 6.6, the GPU achieves better runtime than the multicore implementations. The speedup is maintained to 4.5× using one core, 3.5× for 2 cores, and 2.7× for 4 cores.

6.4 Specific Simulations

In this section, we will introduce a type of simulation of P systems, which is called specific simulation [25]. We will describe how to implement this kind of simulators in CUDA and provide two major examples.

6.4.1 Definition

When implementing a P system simulator, it is important to understand what type of P systems we want to simulate before starting the development. We will say that a specific simulator is any simulator developed for just a certain P system or family of P systems. In other words, if we focus on just one P system or a parametrized definition of P systems forming a family, then we need a specific simulator. On the one hand, this is a restricted version of a simulator, since it can handle a reduced variety of P systems, but it also helps to adjust better the simulator to parallel architectures.

Basically, restricting the P systems to be simulated helps us to take a better control of the algorithm by predicting when certain things will happen. On the one hand, by just focusing on a P system variant, we will know which kind of rules and what semantics apply. On the other hand, by knowing exactly the P systems to simulate, we can develop tailored code to certain parts of the model. For example, we can see in the literature that when developing solutions to certain problems, it is very useful to design it by making a scheme of the computation. This means that the computation tree of the designed P system is usually bound and can be divided into stages. For example, in SAT solutions, there is usually a stage where an exponential amount of membranes is generated by applying division rules, and it is known at which moment the stage starts and ends, because it is part of the design of the solution. By making an exhaustive analysis, it would be even possible to predict which objects can appear in which membranes and when.

Therefore, specific simulators can take advantage of that information in order to adapt the code and the data structures. Specific functions and kernels can be written for each stage, and the memory layout to store the P system information can be drastically reduced. In fact, the information of rules (left and right-hand sides) can be encoded in the source code, instead of storing them in memory, because we know the rules.

The memory layout is also an essential part of a simulator, since P system simulators have been demonstrated to be memory and memory bandwidth bound. When storing the information of P system configuration, we can use either of the following[25]:

- *Sparse representation*: using a large array to store multiplicities, with a position per each possible object (all objects defined in the alphabet). The access is direct since the object identifier is the index where to access the array. However, if many objects are not present at a certain moment, the array will be full of zeroes.
- *Dense representation*: using a double array with a component for the object identifier and the other for multiplicity. We need to search for the object, unless we track them and we know exactly where they are store at any moment. This can help to drastically reduce the size because objects with multiplicity zero can be discarded.

Specific simulators can use dense representation, if we can know where each object is at every transition step, so that the rules can access them directly. Moreover, there are many objects that work as counters, for example o_i, $0 \leq i \leq n$. They are distinct objects but at the end they are related. If we know that o_i and o_j, with $i \neq j$ will not be present at the same time, then we can use just one position, or even use a variable (register) to store the subindex.

Finally, it is important to remark that specific simulators must keep being full simulators; that is, if the simulator goes beyond the P system model and skips representing the P system features, then we are simulating something else. In other words, we will say that a program is simulating a P system if we can ask the program, at any transition step, any piece of information of the state of the P system (configuration, rules applied, etc.). Thus, a specific simulator should be developed in such a way that we could extract from it the configuration of the P system or the rules that have been applied.

6.4.2 Simulating a SAT Solution with Active Membrane P Systems

The first specific simulator implemented in CUDA was a family of recognizer P systems with active membranes designed to solve the SAT problem in linear time (but with exponential workspace). In this section, we will discuss the design of this simulator and its performance achieved with CUDA.

The original work is published in [3, 5, 21, 22]. The framework of all these simulators is named *PCUDASAT*, and it can be downloaded from the official website http://sourceforge.net/p/pmcgpu [45] or the repository https://github.com/RGNC/pcudasat.

6.4.2.1 SAT Solution with Active Membranes

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* $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}$, $1 \leq i \leq m$, where $y_{i,i'} \in \{x_j, \neg x_j : 1 \leq j \leq n\}$ are the literals of φ. We can assume that the formula is in simplified expression, i.e., no clause contains two occurrences of the same literal, and no clause can contain, simultaneously, a literal and its negation. The SAT problem is defined as follows: given a Boolean formula in conjunctive normal form (CNF), to determine whether or not there exists a truth assignment to its variables on which the formula evaluates true.

The solution to SAT based on recognizer P system with active membranes is defined as $\Pi_{am-SAT}(\langle m, n\rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, \mathcal{R}, 2)$ of degree 2, for each pair of natural numbers $m, n \in \mathcal{N}$. Specifically:

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

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

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

We also consider a polynomial encoding (cod, s) of the SAT problem in the family $\mathbf{\Pi_{am-SAT}} = \{\Pi_{am-SAT}(t) \mid t \in \mathcal{N}\}$. The function cod associates to the previously described propositional formula φ (an instance of SAT with n variables and m 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 in 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 $\Pi_{\mathbf{am-SAT}}$. 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 system $\Pi_{am-SAT}(s(\varphi))$ with input $cod(\varphi)$ is confluent, and its computation is structured in four phases as follows:

- *Generation phase*: all possible relevant truth assignment is generated for the set of variables of the formula $\{x_1, \ldots, x_n\}$. It is achieved by using division rules in the internal membranes (labeled by 2). This will allow the generation of 2^n membranes that will 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).

6.4.2.2 Sequential Simulator and Data Structures

The sequential simulator design is based on the four main phases of a P system computation from $\mathbf{\Pi_{am-SAT}}$: generation, synchronization, check-out, and output. Thus, the computation of the P system to simulate (from the family $\mathbf{\Pi_{am-SAT}}$) is reproduced by sequentially executing these phases. 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. Since we know the value of n, the simulator knows the amount of membranes to generate before starting the simulation. 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 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, the

only input for the simulator is the CNF formula provided in DIMACS CNF format. We can assume therefore that the simulator behaves as a SAT solver, receiving a propositional formula and giving the corresponding answer. However, this solver is implemented following a solution by means of P systems.

The first challenge is to decrease the sparsity of the data structures storing the configuration of the P system. After an exhaustive analysis of the computation, the upper bound of number of distinct objects appearing in a membrane can be 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 either another or disappears. By definition, send-in, send-out, and division rules do not generate more than one object in the right-hand side.

Hence, 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.

The encoding of objects for the input multiset can be made at a 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 (subindex i).
4. Clause (subindex j).

It is noteworthy that the membrane charges are not stored, since we can observe from the computation that a partition of membranes having positive and negative charges can be done over the array. In other words, 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 encoded directly in source code.

6.4.2.3 Design of the GPU Simulator

The parallel simulator on the GPU was designed to take over the most demanding phases on the computation of $\mathbf{\Pi_{am-SAT}}$, which are the first three phases. The last one (output phase) is developed on the CPU. In order to map the parallelism into the GPU, the simulator assigns a thread block to each membrane, as shown in Fig. 6.7. 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.

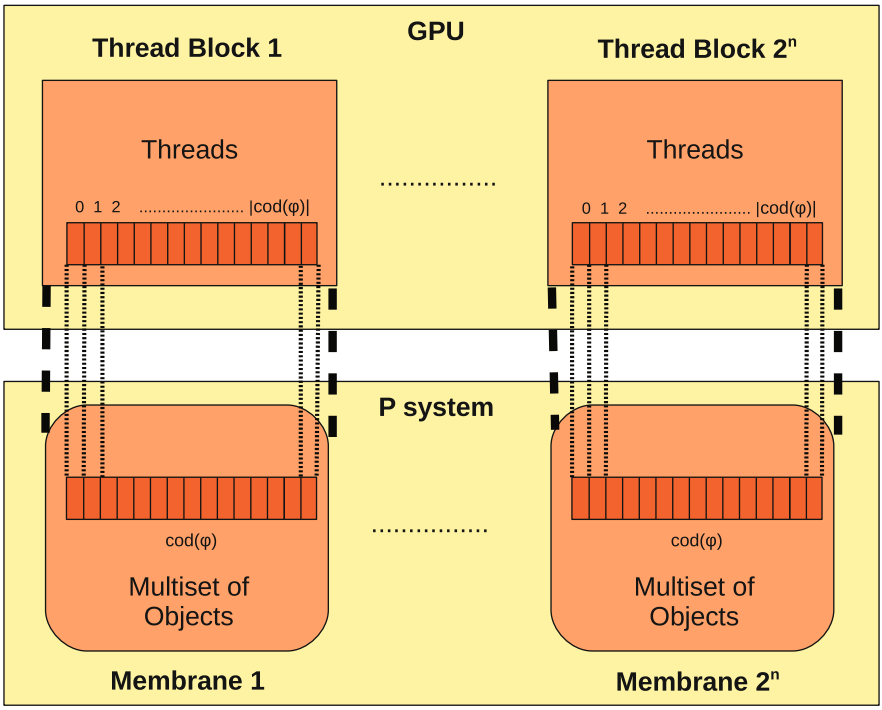

Fig. 6.7 Design of the parallel simulator for $\Pi_{\text{am-SAT}}$. Each generated membrane is assigned to a thread block, and each object (initially, the input multiset) is assigned to a thread. From [3, 21]

Algorithm 4 shows the pseudocode for the host side the simulator. The generation phase is simulated by using three kernels which execute the rules in this phase. This is an iterative process of n steps where the kernels are called n times. A tailored kernel for division is designed to make copies of all membranes executing division rules, whose behavior is shown in Fig. 6.8. There is a double parallelism in the division, as shown in the figure; threads within a block are in charge of making the copies for the new membrane and changing just the corresponding object. But this is also repeated for each thread block. 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 two-dimensional grid of thread blocks.

When the exponential amount of membranes is created, synchronization and check-out phases are executed. This is simulated within just one kernel for both phases, in parallel for each membrane. Global synchronization is not necessary because there is no communication among the internal membranes at these phases. Finally, the output phase is developed on the CPU, checking the conditions and launching the result of the computation.

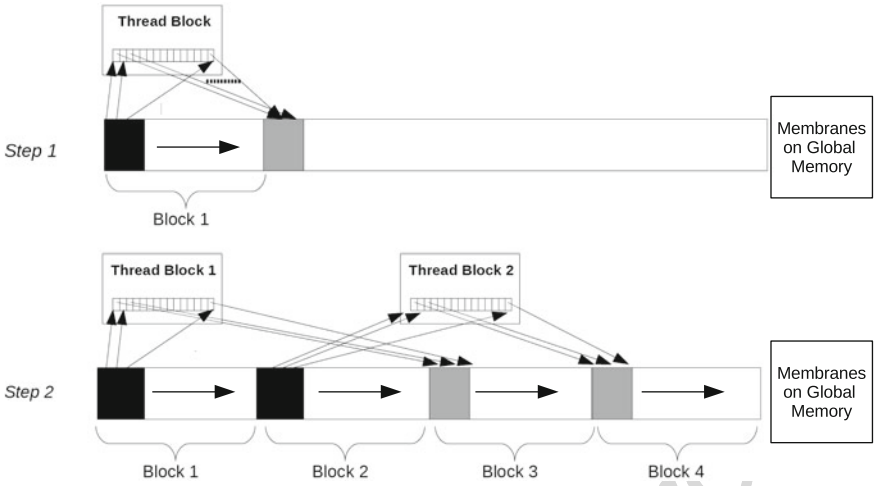

Fig. 6.8 Parallel division on GPU at generation phase in $\Pi_{\text{am-SAT}}$. Each membrane is divided by a thread block, which copies the objects also in parallel by using several threads. If the number of objects is larger than 256, the thread block repeats the process until covering all the objects. From [5]

Algorithm 4: Parallel Simulator of $\Pi_{\text{am-SAT}}$, at host side. From [3, 21]

1: {**Initialization**}
2: $Threads \leftarrow |cod(\varphi)|$ {The number of literals in the CNF formula}
3: $Blocks \leftarrow (1, 1)$ {One block in the 2-dimensional grid}
4: $d \leftarrow 0$ {A counter}
5: $numMembranes \leftarrow 1$ {Number of membranes}
6: $psystem \leftarrow allocateGPUMemory(2^n)$ {Allocate enough memory to represent the P system}

7: {**Generation phase**}
8: **repeat**
9: $Division\_kernel <<< Blocks, Threads >>> (psystem, numMembranes)$
10: $numMembranes \leftarrow numMembranes \times 2$
11: $Blocks \leftarrow AdjustBlocks(psystem, numMembranes)$ {Distribute membranes among blocks}
12: $Send\_out\_kernel <<< Blocks, Threads >>> (psystem, numMembranes)$
13: $Send\_in\_kernel <<< Blocks, Threads >>> (psystem, numMembranes)$
14: $d \leftarrow d + 1$
15: **until** $d < n$ {Repeat n times (number of variables)}

16: {**Synchronization and Check-out phases**}
17: $Syn\_Check\_kernel <<< Blocks, Threads >>> (psystem, numMembranes)$

18: {**Output phase** (executed on the CPU)}
19: $Output(psystem, numMembranes)$

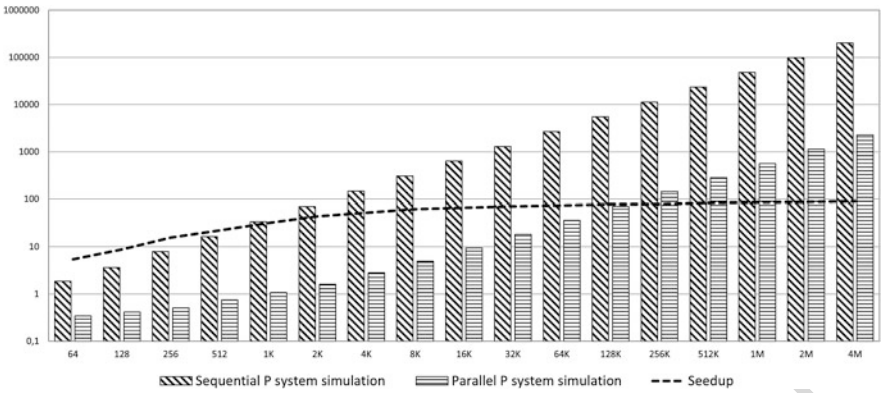

Fig. 6.9 Simulation performance for *am-sat-gpu* vs *am-sat-seq* when increasing the number of membranes (x-axis). From [3, 21]

6.4.2.4 Performance Analysis

Next, we analyze the performance of the simulators above described for $\mathbf{\Pi_{am-SAT}}$: the sequential simulator developed in C++ (from now, *am-sat-seq*) and the GPU parallel simulator on CUDA (*am-sat-gpu*). The experimental results were obtained using a Tesla C1060 GPU.

Figure 6.9 shows the experimental performance of the cell-like simulators (in a log scale) when increasing the number of membranes in the P system (and hence, the number of blocks in the GPU and also the variables in the CNF formula) until reaching 2^{12} membranes. The number of simulated membranes is restricted by the available memory of the system. The number of literals in the formula is fixed to 256, which means 256 threads per block.

It can be seen that once the GPU resources have been fully occupied, the execution time increases linearly with the number of blocks. In this case, we report up to 94× of speedup between *am-sat-seq* and *am-sat-gpu*. However, Fig. 6.9 shows the speedup becomes a constant number of 100× when the number of membranes is greater than 128 K.[1] This is the number of blocks launched in the grid of the GPU.

We finalize the performance analysis by also considering the data management (allocation and transfer) time of the GPU. This is also very important, because it is part of the solution. Figure 6.10 shows the speedup achieved by comparing *am-sat-gpu* (with data management) and *am-sat-seq*. 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 1.23×, and it is increased along with the number of membranes until 64× 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

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

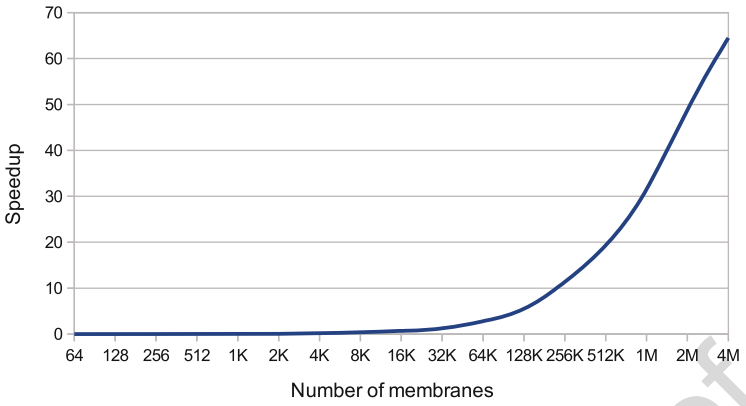

Fig. 6.10 Achieved speedup of *am-sat-gpu* against *am-sat-seq* considering also the GPU data management, when increasing the number of membranes (x-axis). From [3, 21]

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.

6.4.3 Simulating a SAT Solution with Tissue P Systems

In this section, we depict a specific simulator for the family of recognizer tissue P systems with cell division. We first explain the data structures and the phases that compound 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.

The original work is published in [21, 22, 26]. This simulation framework is named *TSPCUDASAT*, and it can be downloaded from the official website http://sourceforge.net/p/pmcgpu [45] or the repository https://github.com/RGNC/tspcudasat.

6.4.3.1 Recognizer Tissue P System with Cell Division

In the paradigm of membrane computing, a new computing model (*tissue P system with cell division*) is introduced by using the biological membranes placed in the nodes of a directed graph, inspired from the cell inter-communication in tissues [20]. Besides, 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.

Definition 6.4. A *tissue P system with cell division* of degree $q \geq 1$, is a tuple $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:

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

A *tissue P system with cell division* $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ 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) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ represent the finite multisets of *objects* (symbols of the working alphabet Γ) initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects initially located in the environment of the system, all of them available in an arbitrary number of copies; (c) $\mathcal{R}$ is a finite set of rules over Γ associated with the cells and the environment; and (d) $i_{out} \in \{0, 1, \ldots, q\}$ indicates the output zone. We use the term *zone* i to refer to cell i, in the case $1 \leq i \leq q$ and to refer the environment in the case $i = 0$.

A communication rule $(i, u/v, j)$ is called a *symport rule* if $u = \lambda$ or $v = \lambda$. A symport rule $(i, u/\lambda, j)$ provides a virtual arc from zone i to zone j. A communication rule $(i, u/v, j)$ is called an *antiport rule* if $u \neq \lambda$ and $v \neq \lambda$. An antiport rule $(i, u/v, j)$ provides two arcs: one from zone i to zone j and another one from zone j to zone i. Therefore, every tissue P system has an underlying directed graph whose nodes are the zones (cells and the environment) of the system and the arcs are obtained from communication rules.

A *configuration* (or *instantaneous description*) C_t at an instant t of a tissue P system Π is a tuple whose components are the multisets over Γ associated with each cell present in the system at moment t and the multiset over $\Gamma \setminus \mathcal{E}$ associated with the environment at moment t.

A communication rule $(i, u/v, j)$ is applicable to zones i, j to a configuration C_t at instant t, if in that configuration the multiset u is contained in one zone i and multiset v is contained in one zone j. When applying such a communication rule to such zones, the objects of the multiset represented by u are sent from zone i to zone j, and simultaneously, the objects of multiset v are sent from zone j to zone i. The *length* of communication rule $(i, u/v, j)$ is defined as $|u| + |v|$.

A division rule $[a]_i \rightarrow [b]_i[c]_i$ is applicable to a configuration at an instant t, if one cell i belongs to that configuration containing object a. When applying a division rule $[a]_i \rightarrow [b]_i[c]_i$ to such a cell i, under the influence of object a, that cell is divided into two cells with the same label i; 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

residing in such a cell i are replicated, and copies of them are placed in the two new cells. The output cell i_{out} and any cell with input degree equal to zero cannot be divided.

The rules of a tissue P system with cell division are applied as follows: communication rules will be applied in a non-deterministic maximally parallel manner as it is customary in membrane computing but with the following important remark: if a cell divides, then the division rule is the only one which is applied for that cell at that step; the objects inside that cell do not evolve by means of communication rules. In other words, before division a cell interrupts all its communication channels with the other cells and with the environment. The new cells resulting from division will interact with other cells or with the environment only at the next step—providing that they do not divide once again. The label of a cell precisely identifies the rules which can be applied to it.

Given a tissue P system with cell division, $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, the *initial configuration* of Π is $C_0 = (\mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$. A configuration is a *halting configuration* if no rule of the system is applicable to it. We say that configuration C_1 yields configuration C_2 in one *transition step*, denoted $C_1 \Rightarrow_\Pi C_2$, if we can pass from C_1 to C_2 by applying the rules from $\mathcal{R}$ following the previous remarks. In tissue P systems with cell division, the concepts of *computation*, *recognizer system*, and *polynomial time and uniform solution* to a decision problem are introduced in a similar way than in P systems with active membranes.

6.4.3.2 SAT Solution with Tissue P Systems

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

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

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

- 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) $(1, e_{2n+2m+nm+3}\ f\ \texttt{no}/\lambda, 0)$.

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), with 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 $\{x_1, \ldots, x_n\}$. This is accomplished 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.
- *Counter 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 is 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 are exchanged for the objects f and g of the 2^n cell 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 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 allow 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 $C_{2n+m+nm+1}$ 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 are 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 $C_{2n+m+nm+m+1}$.
- *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 $C_{2n+m+nm+m+1}$ 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) cannot 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 $C_{2n+m+nm+m+1}$ 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 cell 1. This object permits the application of rule (t), since the objects `no` and f remain in cell 1. In this way, the object `no` is sent in the next computation step, and the computation finalizes.

6.4.3.3 Sequential Simulation and Data Structure

For an easier implementation, the simulation algorithm has been divided into five (simulation) phases, instead of the six phases in $\Pi_{\mathbf{tsp-SAT}}$ since we merge some of them. 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 $\mathbf{\Pi_{tsp-SAT}}$. 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. The input of the simulator is the same than the one used in the simulator for the cell-like solution $\mathbf{\Pi_{am-SAT}}$. 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 as follows [21]:

- As the exchange phase is very simple, it is then implemented after the generation phase loop, within the same function.
- The full synchronization phase is applied to one cell before going to the next one. This allows to exploit data locality in cache memories.
- In the checking phase, the objects r_j, for $1 \leq j \leq m$, are inserted in order 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.

For this solution, the memory layout for the representation of the tissue P system differentiates between cells labeled 1 and 2, having a different data structure representing each type of cell in the system.

First, cell 1 is represented as an array having a maximum dimension of five elements. That is, the multiset for cell 1 has the maximum amount of five objects. These five 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:

- $|cod(\varphi)|$ 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 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 (subindex i).
4. Clause (subindex j).

6.4.3.4 Design of the Parallel Simulator

The design of this parallel simulator is driven by the same structure of phases we have used for the sequential one. Separated CUDA kernels are utilized to speedup the execution of each phase.

The general assignment of work for threads and thread blocks is summarized in Fig. 6.11. 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.

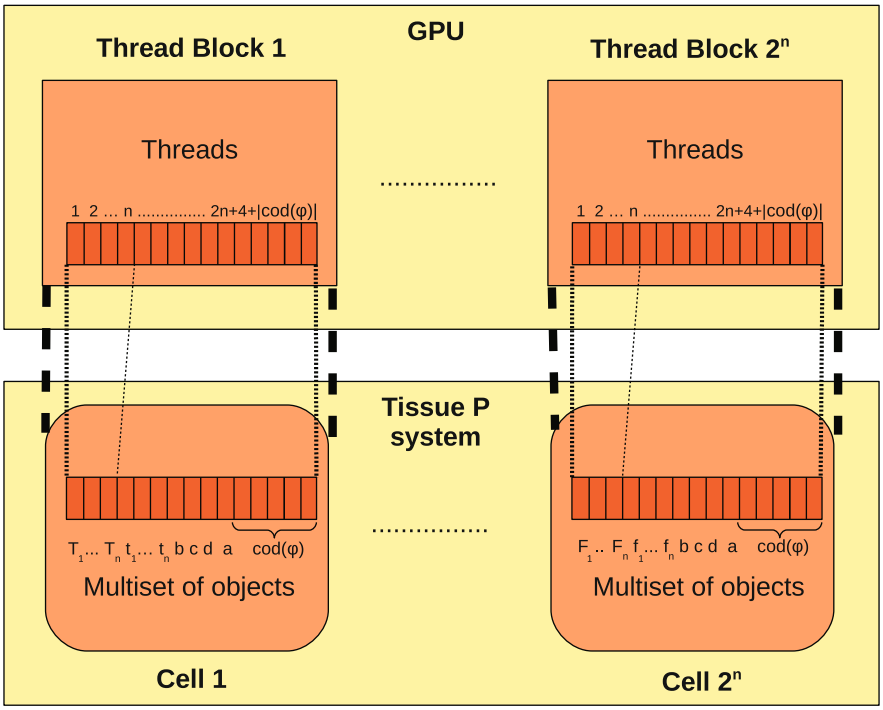

Fig. 6.11 General design of the parallel simulator for $\mathbf{\Pi_{tsp-SAT}}$. From [21, 26]

- *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. 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. The first one is to *emphasize the parallelism*. This optimization aims to increase the number of threads per block (to the recommended amount from 64 to 256), so it allows to fulfill warps and hide latency. 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 Sect. 6.4.2.
- *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 the 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 $n = 32$. 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 fulfil 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 $|cod(\varphi)|$.

6.4.3.5 Performance Analysis

In this subsection, we analyze the performance of the two simulators developed for the family of tissue-like P systems $\Pi_{\mathbf{tsp-SAT}}$: the sequential simulator developed in C++ (from now, *tsp-sat-seq*) and the parallel simulator on the GPU (*tsp-sat-gpu*).

Figure 6.12 shows the results for both simulators when increasing the number of cells (by increasing the number of variables in the input CNF formulas), considering only kernel runtime for *tsp-sat-gpu*. For this case, we can observe that again the kernels of *tsp-sat-gpu* run faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cell 2 created by the system. For 64 membranes, the speedup is of 2×, but for 2 M cells it is of 8.3×.

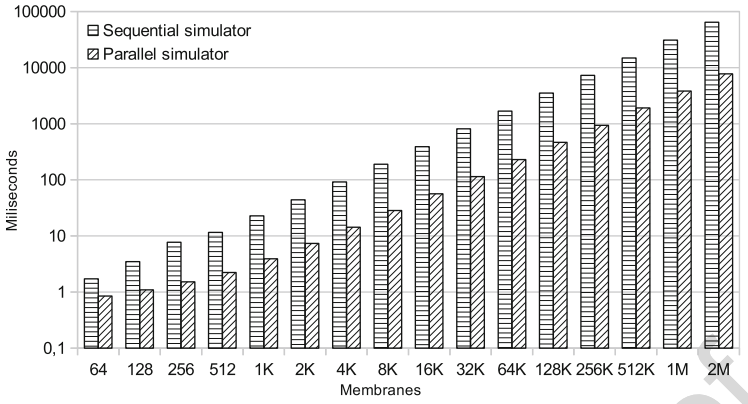

Fig. 6.12 Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu* when increasing the number of membranes (x-axis). From [21, 26]

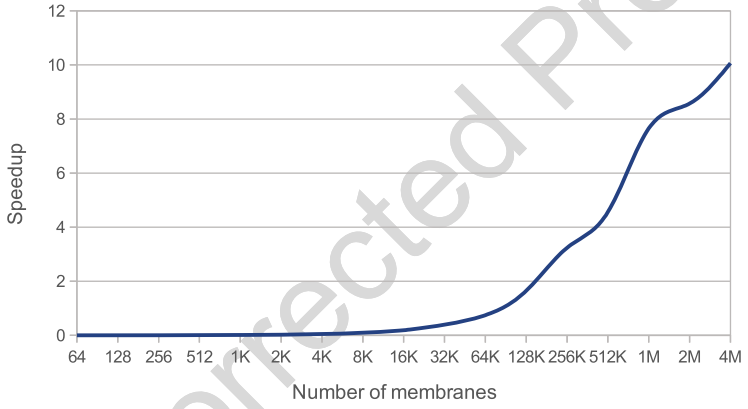

Fig. 6.13 Speedup achieved running Test 2 (256 Objects/Cell) for *tsp-sat-gpu* and *tsp-sat-seq* considering also the GPU data management, when increasing the number of membranes (x-axis). From [21, 26]

Finally, we show the speedup achieved by 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 10× (Fig. 6.13).

AQ1

6.5 Adaptive Simulations

In this section, we will introduce a third type of simulation of P systems, which is called adaptive simulation.

6.5.1 Definition

We have discussed the difference between generic and specific simulators. In this section, we will discuss a hybrid type, which is called adaptative, or simply adaptive, simulation. A simulator of this kind is initially a generic simulator, which is designed to simulate a wide range of P systems within a variant. However, the simulator is provided with high-level information that can be either discarded (then remaining as generic) or used to adapt the simulation to improve its efficiency.

In this sense, an adaptative simulator has the goal of getting closer to specific simulators without losing generality; that is, they are generic simulators with improved performance by taking advantage of extra information provided directly by designers (e.g., modules). For example, if the algorithm scheme of the computation is known by the designer (as it is, as discussed for the specific simulators), then it can be given to the simulator in order to be able to discard rules at selection stage (because the algorithm scheme is known).

Next, we will overview the first adaptive simulator for P systems implemented so far, which is published in [29]. This simulation framework is implemented within *ABCD-GPU*, and it can be downloaded from the official website http://sourceforge.net/p/pmcgpu [45] or the repository https://github.com/RGNC/abcd-gpu/tree/adaptative.

6.5.2 Simulating Population Dynamics P Systems

The idea of adaptative simulators was introduced and analyzed in [29]. It is inspired in the way directives work in common programming languages. They are special syntactic elements that tell extra information to the compiler, allowing to better adapt the code for some purposes if the compiler accepts it (e.g., in OpenMP, one call can easily ask to parallelize the iterations of a loop). This way, a P system model designer can also provide very useful information to the simulator, rather than just the syntactic and/or semantic elements of the P system to simulate, such as the algorithmic scheme of the computation.

Specifically in PDP systems, ecosystem modelers often use algorithmic schemes for their models [6]. This is given as cycle that is repeated (per year, per season, etc.). A cycle in the model is a fixed amount of transition steps where a sequence of modules take place. These modules reproduce certain processes such as reproduction of species, feeding, migration, etc. Moreover, these modules consist of certain rules that are carefully designed to model the corresponding process. Therefore, we can

say that somehow the model designer already knows which rules can be executed in each time step. Thus, if they are able to provide that information, the simulator can take advantage of this to dismiss rules automatically at each step.

The PDP system simulator was turned into adaptative. First, the model designer is able to provide the information of the modules they are defining by using the new P-Lingua 5 software [37]. This new version now includes new syntax elements called *features*. They are written as `@featureName = featureValue` and can be defined globally (for the whole system) or locally (for individual rules). ABCD-GPU takes this information to organize the rules by modules. If the simulator does not recognize the information provided by the features, it can proceed and simulate the system without problems.

In summary, there are two main pieces of information that has to be declared in order to define modules:

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

The simulator precomputes which modules are active in each step within the cycle before starting the simulation. In this way, this information can be used to easily identify the rules that might be applicable at each transition step. For this purpose, the rule blocks and the rules are sorted in order to compact them into modules; rules belonging to the same module are put one after the other. The kernels of ABCD-GPU are expanded to accept extra indexes indicating the modules and where the rules of the modules are. In this way, the threads as distributed in Fig. 6.4 will have a shorter loop, because the rule blocks (and rules) are just those from the module being active. Furthermore, if the solution has parallel modules in a cycle, then they can also run in parallel thanks to CUDA streams. We can launch the kernels for phase 1 also in parallel at different streams, one per module. As for environments and simulations, the behavior remains as before.

6.5.2.1 Analysis of Performance Results

Next, the behavior and performance of the adaptative PDP system simulators for GPU and OpenMP are analyzed. The model employed as benchmark is based on the tritrophic interactions presented in [9, 10]. This is a virtual ecosystem that was defined to illustrate PDP systems as a modeling framework. In this model, three trophic levels are represented: grass, herbivores, and carnivores. These species interact with each other, reproduce, and move along the 10 environments when no food is encountered. Rule block competitions take place. For instance, all herbivores compete for grass that is represented by a single object, *G*.

For benchmarking purposes, the model has been generalized so that the number of species can be changed. The corresponding parameters (probabilities, amount of copies eaten per species, etc.) are generated randomly. This was possible thanks to

the ability of P-Lingua 5 to incorporate calls from the model to random number generation functions. Moreover, the modules of the model are identified by P-Lingua 5 features.

In this section, the benchmark carried out to the adaptative PDP systems simulator is analyzed. The two versions of the simulator are compared: generic and adaptive versions of ABCD-GPU. The extended tritrophic model is used as input. In all experiments, 20 years of the virtual ecosystem are simulated (corresponding to 180 transition steps of the PDP systems). The *A* parameter of DCBA is set to 2. No output was asked, so only the simulation runtimes were measured. The scalability of the simulators is analyzed by increasing the number of species. Specifically, 7 will be used to denote the *base* model, which has in fact 7 species. In order to have an idea of the dimensions of the model, the ratio of rule blocks per species is approximately 22: 21985 rule blocks are generated for 1000 species, being 9990 communication rule blocks and 11,995 skeleton rule blocks. Another parameter affecting scalability is the amount of simulations running in parallel. For this reason, 50 simulations were launched for the tests. The following two configurations of CPU and GPU hardware were used to run the simulations (short names are provided in bold):

- **(i7)** Intel i7-8700 CPU at 3.20 GHz, having 12 logical cores (6 physical)
- **(P100)** Tesla P100 GPU, having 3584 cores at 1.33 GHz

A cross comparison of runtimes and speedups achieved by GPU compared to CPU is shown in Fig. 6.14, which corresponds to the speedups reached by the above simulation times. The GPU is faster, in both *adaptive* and *generic* versions, than the multicore counterparts when handling middle and large models. Only for the

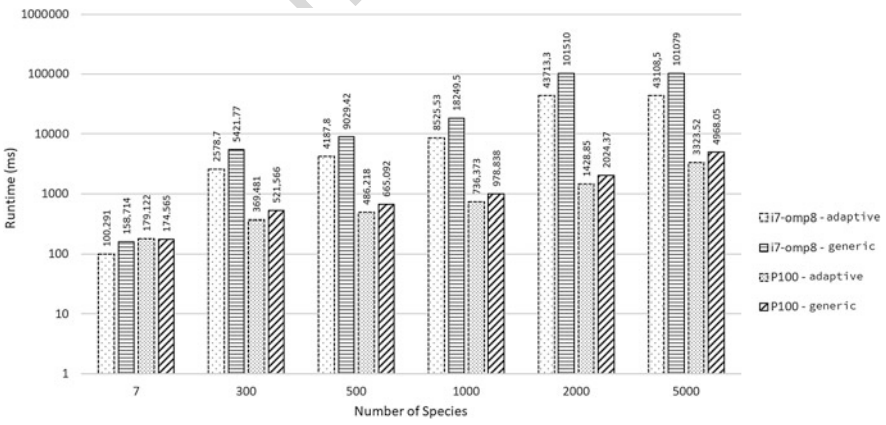

Fig. 6.14 Comparison of P100 versus i7 with 8 threads, for both generic and adaptive versions of *abcd-gpu* tested for different number of species in the model. It shows the corresponding speedups of P100 against i7 for both version. 50 simulations were run. Bar plots use logarithmic scale for y-axis. From [29]

small base model, the GPU is a bit slower (above 0.9×). Speedups are higher with larger models, being around 30× and 50× for *adaptive* and *generic* simulators, respectively, and for 2000 species. When simulating hundreds of species, 6× and 10× accelerations were obtained for *adaptive* and *generic* versions. Finally, the speedup of the GPU is lower when using the *adaptive* version, given that the impact of the *modular* scheme is better for the CPU than for the GPU.

We can conclude that this design helped to improve the performance by 2.5× extra when using a P100 GPU [29].

6.6 Conclusions

GPUs have been established as a massively parallel processor and an enabling technology where programmers currently accelerate scientific applications. They provide a good parallel platform to simulate P 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. Moreover, they are a cheap and scalable parallel architecture that can be seen in current HPC solutions.

However, the results in the literature [22, 48] show that P systems simulations are memory bandwidth bound: they spend more time accessing and updating data (multisets) than executing computation. The main cause is 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 operations to execute per memory access is very small (P systems execute rewriting rules). This restricts the design of parallel simulators. A parallel simulator 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 can identify a taxonomy of simulators developed so far. Generic parallel simulators are intended to be flexible enough to simulate a wide range of P systems within a variant. They also take advantage of P systems parallelism to speedup the simulation. However, 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. Some variants simulated by generic simulators are P systems with active membranes and elementary division and Population Dynamic P systems. Other related works also include spiking neural P systems variants [1, 2].

On the other hand, specific simulators are designed for just certain P systems within a solution or family. This way, the simulator can be designed adapting all parts to the P systems, since their scheme is known at developing time. The performance achieved in these simulators are much higher, but it comes at restricting the P systems to simulate. For example, one cannot define new rules to simulate, since they are already predetermined. In the middle term, we have a new type of simulation called adaptive. Basically, it is a generic simulator but that includes high level information that can be either discarded by the simulator (going generic) or used to adapt the simulation and achieve better performance (adaptive).

We can identify some challenges for the future. For example, concerning the memory bandwidth limit, one challenge is to design P system variants where the model contains a higher computational intensity. Moreover, memory accesses can be partially reduced by improving data structures using a compacted, dense, and well-ordered memory representation of P systems. A challenge is to use a dense representation in an effective way in generic simulators. Finally, a P system model with cooperation in the LHS usually leads to this issue, making it more difficult when the cooperation is larger.

References

1. J.P. Carandang, F.G.C. Cabarle, H.N. Adorna, N.H.S. Hernandez, M.A. Martínez-del-Amor, Handling non-determinism in spiking neural P systems: algorithms and simulations. Fundam. Inf. **164**(2–3), 139–155 (2019). https://doi.org/10.3233/FI-2019-1759
2. J.P. Carandang, J.M.B. Villaflores, F.G.C. Cabarle, H.N. Adorna, M.A. Martínez-del-Amor, CuSNP: spiking neural P systems simulators in CUDA. Rom. J Inf Sci Technol. **20**(1), 57–70 (2017)
3. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulating a P system based efficient solution to SAT by using GPUs. J Logic Algebraic Program. **79**(6), 317–325 (2010). https://doi.org/10.1016/j.jlap.2010.03.008
4. 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 Bioinf. **11**(3), 313–322 (2010). https://doi.org/10.1093/bib/bbp064
5. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, M. Ujaldón, The GPU on the simulation of cellular computing models. Soft Comput. **16**(2), 231–246 (2012). https://doi.org/10.1007/s00500-011-0716-1
6. M.A. Colomer, A. Margalida, M.J. Pérez-Jiménez, Population Dynamics P system (PDP) models: a standardized protocol for describing and applying novel bio-inspired computing tools. PLOS ONE **8**(5), e60698 (2013). https://doi.org/10.1371/journal.pone.0060698
7. M.A. Colomer, A. Margalida, D. Sanuy, M.J. Pérez-Jiménez, A bio-inspired computing model as a new tool for modeling ecosystems: the avian scavengers as a case study. Ecol. Model. **222**(1), 33–47 (2011). https://doi.org/10.1016/j.ecolmodel.2010.09.012
8. M.A. Colomer, A. Margalida, L. Valencia, A. Palau, Application of a computational model for complex fluvial ecosystems: the population dynamics of zebra mussel Dreissena polymorpha as a case study. Ecol. Complexity **20**, 116–126 (2014). https://doi.org/10.1016/j.ecocom.2014.09.006
9. M.A. Colomer, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem, in *IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010)*, vol. 1 (2010). https://doi.org/10.1109/BICTA.2010.5645258
10. M.A. Colomer, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem. Nat. Comput. **11**(3), 369–379 (2012). https://doi.org/10.1007/s11047-011-9289-2
11. T.S. Crow, Evolution of the graphical processing unit. Master's thesis, University of Nevada Reno (2004). http://www.cse.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf
12. A.C. Elster, High-Performance Computing: past, present, and future, in *Applied Parallel Computing, Lecture Notes in Computer Science*, ed. by J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, P. Raback, V. Savolainen, vol. 2367 (2006), pp. 433–444. https://doi.org/10.1007/3-540-48051-X_43

AQ2

13. M. García-Quismondo, R. Gutiérrez-Escudero, M.A. Martínez-del-Amor, E. Orejuela-Pinedo, I. Pérez-Hurtado, P-Lingua 2.0: A software framework for cell-like P systems. Int. J. Comput. Commun. Control **4**(3), 234–243 (2009). https://doi.org/10.15837/ijccc.2009.3.2431
14. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, A fast P system for finding a balanced 2-partition. Soft Comput. **9**, 673–678 (2005). https://doi.org/10.1007/s00500-004-0397-0
15. Inside HPC blog. http://insidehpc.org
16. B.W. Kernighan, D. Ritchie. *The C Programming Language*, 2nd edn. (Prentice Hall, Englewood Cliffs 1988)
17. D.B. Kirk, W.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd edn. (Morgan Kaufmann, San Francisco, 2016). https://www.sciencedirect.com/science/book/9780128119860
18. A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. **25**(2) (2012). https://doi.org/10.1145/3065386
19. E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008). https://doi.org/10.1109/MM.2008.31
20. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríguez-Patón, Tissue P systems. Theor. Comput. Sci. **296**(2), 295–326 (2003). https://doi.org/10.1016/S0304-3975(02)00659-X
21. M.A. Martínez-del-Amor, *Accelerating Membrane Systems Simulators Using High Performance Computing with GPU*. Ph.D. Thesis, Universidad de Sevilla, 2013. http://hdl.handle.net/11441/15644
22. M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Simulating P systems on GPU devices: a survey. Fundam. Inf. **136**(3), 269–284 (2015). https://doi.org/10.3233/FI-2015-1157
23. M.A. Martínez-del-Amor, I. Karlin, R.E. Jensen, M.J. Pérez-Jiménez, A.C. Elster, Parallel simulation of probabilistic P systems on multicore platforms, in *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, ed. by M. García-Quismondo, L.F. Macías-Ramos, Gh. Păun, L. Valencia-Cabrera, vol. II (Fénix Editora, 2012), pp. 17–26
24. M.A. Martínez-del-Amor, L.F. Macías-Ramos, L. Valencia-Cabrera, M.J. Pérez-Jiménez, Parallel simulation of Population Dynamics P systems: updates and roadmap. Nat. Comput. **15**(4), 565–573 (2015). https://doi.org/10.1007/s11047-016-9566-1
25. M.A. Martínez-del Amor, D. Orellana-Martín, I. Pérez-Hurtado, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Design of specific P systems simulators on GPUs, in *Membrane Computing. CMC 2018*, ed. by T. Hinze, G. Rozenberg, A. Salomaa, C. Zandron. Lecture Notes in Computer Science, vol. 11399 (2019), pp. 202–207. https://doi.org/10.1007/978-3-030-12797-8_14
26. M.A. Martínez-del-Amor, J. Pérez-Carrasco, M.J. Pérez-Jiménez, Characterizing the parallel simulation of P systems on the GPU. Int. J. Unconv. Comput. **9**(5–6), 405–424 (2013)
27. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez., M.A. Colomer, M.J. Pérez-Jiménez, DCBA: simulating Population Dynamics P Systems with proportional object distribution, in *Membrane Computing. CMC 2012*, ed. by E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil. Lecture Notes in Computer Science, vol. 7762 (2012), pp. 291–310. https://doi.org/10.1007/978-3-642-36751-9_18
28. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez, Population Dynamics P Systems on CUDA, in *Computational Methods in Systems Biology*, ed. by D. Gilbert, M. Heiner. Lecture Notes in Computer Science, vol. 7605 (2012), pp. 247–266. https://doi.org/10.1007/978-3-642-33636-2_15
29. M.A. Martínez-del-Amor, I. Pérez-Hurtado, D. Orellana-Martín, M.J. Pérez-Jiménez, Adaptative parallel simulators for bioinspired computing models. Future Gener. Comput. Syst. **107**, 469–484 (2020). https://doi.org/10.1016/j.future.2020.02.012

30. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, M.A. Colomer, A new simulation algorithm for multienvironment probabilistic P systems, in *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, Changsha, 2010, vol. 1 (2010), pp. 59–68. https://doi.org/10.1109/BICTA.2010.5645352
31. A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, *OpenCL Programming Guide*, 1st edn. (Addison-Wesley, Reading, 2011)
32. J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA: is CUDA the parallel programming model that application developers have been waiting for? Queue **6**(2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500
33. NVIDIA CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed June 2019
34. N. Otterness, J. Anderson, AMD GPUs as an alternative to NVIDIA for supporting real-time workloads, in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems*, (2020), pp. 10:1–10:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.10
35. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing. Proc. IEEE **96**(5), 879–899 (2008). https://doi.org/10.1109/JPROC.2008.917757
36. Gh. Păun, P systems with active membranes: attacking NP-complete problems. J. Autom. Lang. Comb. **6**, 75–90 (1999)
37. I. Pérez-Hurtado, D. Orellana-Martín, G. Zhang, M.J. Pérez-Jiménez, P-lingua in two steps: flexibility and efficiency. J. Membr. Comput. **1**(2), 93–102 (2019). https://doi.org/10.1007/s41965-019-00014-1
38. M.J. Pérez-Jiménez, A. Riscos-Núñez, Solving the Subset-Sum problem by P systems with active membranes. N. Gener. Comput. **23**(4), 339–356 (2005). https://doi.org/10.1007/BF03037637
39. M.J. Pérez-Jiménez, A. Riscos-Núñez, A linear-time solution for the knapsack problem with active membranes, in *Membrane Computing. WMC 2003*, ed. by C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg, A. Salomaa. Lecture Notes in Computer Science, vol. 2933 (2004), pp. 250–268. https://doi.org/10.1007/b95207
40. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, Complexity classes in models of cellular computing with membranes. Nat. Comput. **2**(3), 265–285 (2003). https://doi.org/10.1023/A:1025449224520
41. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, Decision P systems and the $\mathbf{P} \neq \mathbf{NP}$ conjecture, in *Membrane Computing. WMC 2002*, ed. by Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron. Lecture Notes in Computer Science, vol. 2597 (2003), pp. 388–399. https://doi.org/10.1007/3-540-36490-0_27
42. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, A polynomial complexity class in P systems using membrane division. J. Autom. Lang. Comb. **11**, 423–434 (2006)
43. N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*, Rome, 2009 (IEEE Computer Society, Silver Spring, 2009), pp. 1–10. https://doi.org/10.1109/IPDPS.2009.5161005
44. W. Shin, K.H. Yoo, N. Baek, Large-scale data computing performance comparisons on SYCL heterogeneous parallel processing layer implementations. Appl. Sci. **10**, 1656 (2020). https://doi.org/10.3390/app10051656
45. The PMCGPU (Parallel simulators for Membrane Computing on the GPU) project website. http://sourceforge.net/p/pmcgpu. Accessed June 2019
46. The top 500 supercomputer site. http://www.top500.org
47. A. Torres-Moríño, M.A. Martínez-del-Amor, F. Sancho-Caparrini, GPU-parallel crowd simulation with Vulkan, in *Proceedings of the 18th High Performance Computing & Simulation (HPCS 2020)*. (2020, in press)
48. G. Zhang, Z. Shang, S. Verlan, M.A, Martínez-del-Amor, C. Yuan, L. Valencia-Cabrera, M.J. Pérez-Jiménez, An overview of hardware implementation of Membrane Computing models. ACM Comput. Surv. **53**(4), 90 (2020). https://doi.org/10.1145/3402456

AUTHOR QUERIES

AQ1. Missing citation for “Fig. 6.13” was inserted. Please check if appropriate. Otherwise, please provide citation for “Fig. 6.13”. Note that the order of main citations of figures in the text must be sequential.

AQ2. Please note that the references in the list have been arranged alphabetically and corresponding citations have been changed accordingly. Please check.