J Supercomput (2014) 69:248–272
DOI 10.1007/s11227-014-1150-9

Evaluating the SAT problem on P systems for different high-performance architectures

José M. Cecilia · José M. García ·
Ginés D. Guerrero · Manuel Ujaldón

Published online: 19 March 2014

Abstract Membrane computing is an emergent research area studying the behavior of living cells to define bio-inspired computing devices, also called *P systems*. Such devices provide polynomial time solutions to NP-complete problems by trading time for space. The efficient simulation of P systems poses three major challenging issues: an intrinsic massive parallelism of P systems, an exponential computational workspace, and a non-intensive floating point nature. This paper analyzes the simulation of a family of recognizer P systems with active membranes that solves the satisfiability problem in linear time on three different architectures: a shared memory multiprocessor, a distributed memory system, and a manycore graphics processing unit (GPU). For an efficient handling of the exponential workspace created by the P systems computation, we enable different data policies on those architectures to increase memory bandwidth and exploit data locality through tiling. Parallelism inherent to the target P system is also managed on each architecture to demonstrate that GPUs offer a valid alternative for high-performance computing at a considerably lower cost. Our results lead to execution time improvements exceeding 310× and 78×, respectively, for a much cheaper high-performance alternative.

J. M. Cecilia
Computer Science Department, Universidad Católica San Antonio (UCAM), 30107 Murcia, Spain
e-mail: jmcecilia@ucam.edu

J. M. García
Computer Engineering Department, University of Murcia, 30100 Murcia, Spain
e-mail: jmgarcia@ditec.um.es

G. D. Guerrero
National Lab for High Performance Computing (NLHPC), Center for Mathematical Modeling (CMM), School of Engineering and Sciences, University of Chile, Santiago, Chile
e-mail: gguerrero@nlhpc.cl

M. Ujaldón (✉)
Computer Architecture Department, University of Malaga, 29071 Malaga, Spain
e-mail: ujaldon@uma.es

Keywords Manycore · GPUs · P systems · SAT problem · High-performance computing

1 Introduction

Parallel computing architectures have brought dramatic changes to mainstream computing [4]. This trend is accelerating as the end of the development of hardware following Moore's law looms on the horizon. The number of transistors per die no longer relies on a single chip design, but partitioned among a bunch of simpler cores [6]. Multicore CPUs hold a dozen cores, and manycore GPUs gather a myriad of stream processors. These components are combined to build heterogeneous parallel computers offering a wide spectrum of high-speed processing functions [13] and evaluation methods [24].

A major hurdle for exploiting this raw power is the PCI express bus for communication between the CPU and GPU, as they do not share memory space, and also their different parallel programming approaches and paradigms. These problems amplify when we move to heterogeneous clusters [14]. This article explores this complex situation for a challenging application which requires (1) a dynamic handling of memory space and (2) an exponential workspace growing as our code increases the number of variables involved to run the simulation.

Our work characterizes membrane computing, an emergent research area which studies the behavior of living cells to define bio-inspired computing devices, also called P systems. These devices provide polynomial time solutions to NP-complete problems by trading time for space. This is inspired by the capability of cells to produce an exponential number of new membranes in polynomial time, through *mitosis* and *autopeosis* processes.

Currently, we lack a feasible biological implementation, either in vivo or in vitro, of P systems. The only way to analyze and execute these devices is on silicon-based architectures which are limited by the physical laws. Although some simulators and software applications have been derived [11,12], most of these simulators were developed for sequential architectures using languages such as Java, CLIPS, Prolog or C, where performance is not seen as a critical goal. Trying to exploit as much as possible the parallelism inherent to P system definition, Alonso et al. [3] proposed a circuit implementation for the class of transition P systems. Nguyen et al. [17] proposed an implementation of transition P systems in FPGAs, providing several levels of parallelism, one at the rule level and the other at the region level, releasing a software framework for membrane computing called Reconfig-P. Finally, we developed a generic simulator on GPUs for a family of recognizer P system with active membranes in [8], showing that the double level of parallelism exposed by GPUs represents a valid alternative to simulate P systems.

We implemented a specific simulator for the family of P systems for solving the `SAT` problem in a single GPU in [7], where an analysis to carry out the theoretical simulation of P systems on GPUs was depicted, and some heuristics to accelerate its computation were provided. This work was recently enhanced in [9] with a performance analysis on several GPU multiprocessors.

In this article, we have developed new implementations for shared memory and distributed memory architectures based on current CPUs. We have also developed a new hybrid implementation exploiting the shared memory model at the node level and the message passing model for the communications among nodes. The singularities of P systems for an efficient handling of the workspace lead to different data policies on each of those architectures to increase memory bandwidth and data locality. We have compared all those alternatives against our GPU implementation, analyzing the pros and cons on each hardware platform to put into perspective GPU numbers and emphasize its excellent ratio performance/cost. Overall, GPUs defeat these implementations with gains of factors exceeding 310× and 78× compared to shared and distributed memory architectures, respectively, to represent a much cheaper high-performance alternative.

Section 2 of this article introduces membrane computing and describes the behavior of this biologically inspired way of computation, focusing on computational devices called P systems to solve the satisfiability (SAT) problem. Section 3 describes the parallelism which can be extracted from a P system simulation with active membranes, and once this is learnt, we demonstrate in Sect. 4 how GPUs can accommodate two levels of parallelism in the computational model versus a single level on shared and distributed memory systems. Section 5 analyzes different data policies to increase the memory bandwidth and also to take advantage of the data locality on each architecture via a blocking/tiling algorithm. Finally, Sect. 6 highlights the main ideas presented and provides some directions for future work.

2 Background and related work

2.1 Membrane computing and P systems

Membrane computing [20] is a bio-inspired computing paradigm that has attracted many researchers within natural computing. The model assumes that processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this model are called P systems, which consist of a cell-like membrane structure, where compartments allocate sets of objects (or multisets) with multiplicities associated with the elements.

Syntactic elements of P systems are the following (see Fig. 1): first, a membrane structure consisting of a hierarchical arrangement of membranes embedded in a skin, the frontier between the internal region of the P system and the environment; second, delimiting regions or compartments where multisets (corresponding to chemical substances) and sets of evolution rules (corresponding to reaction rules) are placed. Every membrane has a label and eventually a charge or polarization that can be modified during the computation. From a computational perspective, P systems include two valuable features: inherent parallelism and non-determinism.

A P system computation is a sequence of instantaneous transitions between configurations. The computation begins with an initial configuration, where input data of a given problem are encoded. The transition from one configuration to the next is performed by applying rules to the objects within the regions. This process iterates until no more rules can be applied to the existing objects and membranes. This way,

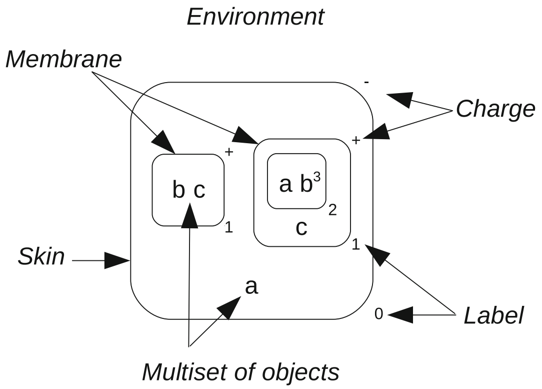

Fig. 1 The structure of a P system

P systems exhibit two levels of parallelism: one for each region (rules are applied in parallel), and another one for the system (all regions evolve concurrently). The objects inside the membranes evolve according to the given rules in a parallel, synchronous and non-deterministic way.

Parallelism and non-determinism can be exploited to solve NP-complete problems in polynomial time, reducing this from an exponential time, but at the expense of using an exponential workspace of membranes and objects which is created in polynomial (often linear) time.

The lack of either in vivo or in vitro implementations of P systems has encouraged researchers to focus on simulators developed in silicon. Those were originally targeted to sequential platforms [11,12], to later address parallelism in different platforms.

2.2 The satisfiability (SAT) problem

Propositional satisfiability problem (SAT) was the first known **NP**-complete algorithm [10]. In computational logic, SAT is a decision problem aimed to determine, for a formula of the propositional calculus in conjunctive normal form (CNF), if there is an assignment of truth values to its variables which that formula evaluates to be true. This is of paramount importance in many computer science areas, including theory, algorithmic, artificial intelligence, hardware design, electronic design automation and verification.

We assume a formula to be in CNF when it is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation (the negation of an expression can be reduced to negated variables by De Morgan's laws). For example, a_1 is a positive literal and $\neg a_2$ is a negative literal.

Considering a CNF formula φ with n variables $(x_1...x_n)$ and m clauses $(C_1...C_m)$, the time spent by all known deterministic algorithms to solve the SAT problem is exponential depending on the size of the input $(\max\{m, n\})$ in the worst case. Several works have been done in the SAT problem resolution [5,16]. However, with the help of membrane systems, we are able to find the solution at linear time, but at the expense of creating an exponential workspace.

The P system simulation algorithm to solve the SAT problem is based on the P system computation described in [21], which can be summarized as the following list of stages:

1. **Generation.** Membranes are organized into a rooted tree with a single branch. The root node is the *skin membrane*, and the second node is called *internal membrane*. Truth assignments to the variables are generated by using division rules and encoded in the internal membranes through an ordered execution for the set of P system rules already described in [21]. This way, 2^n internal membranes are created, each containing a truth assignment to the variables of the formula, exhaustively searching the entire configuration space of the problem.
2. **Synchronization.** The objects encoding a true clause (a partial solution to the CNF formula) are unified in the membrane.
3. **Check out.** The goal here is to determine how many (and which) clauses are *true* in every internal membrane by the assignment that it represents.
4. **Output.** Internal membranes encoding a solution send an object to the skin. If the skin has/does not have such object from some membrane, the object Yes/No is sent to the environment.

A detailed discussion of P system computation to solve the SAT problem is beyond the scope of this paper. We refer the reader to [8,21] for a comprehensive overview.

Algorithm 1 summarizes the sequential code based on the last description. *Generation* and *Synchronization* stages create an exponential workspace of membranes in a synchronous way, also unifying the objects that codify a partial solution. These two stages execute in the same function, which is referred to as *Generation* from now on. At each iteration of *Generation*, each membrane runs in parallel, with a global synchronization required between iterations.

Check out and *Output* stages are performed once the workspace is created. First, they determine the clauses that are true in every internal membrane and then check whether there is a solution for the SAT problem. Hereafter, we combine these two stages into a joint *CheckOut* function.

Algorithm 1 The sequential pseudocode of the P system simulation algorithm for the SAT problem with n variables.

```
Require: $n \geq 0$
  {Start Generation and Synchronization stages}
  repeat
    $Generation$
  until $n$
  {Start Check out and Output stages}
  $CheckOut$
```

3 The parallel version for the P system solving the SAT problem

The P system for the SAT problem gathers all computational features of the recognizer P systems with active membranes [19]. Among them, we highlight the theoretical

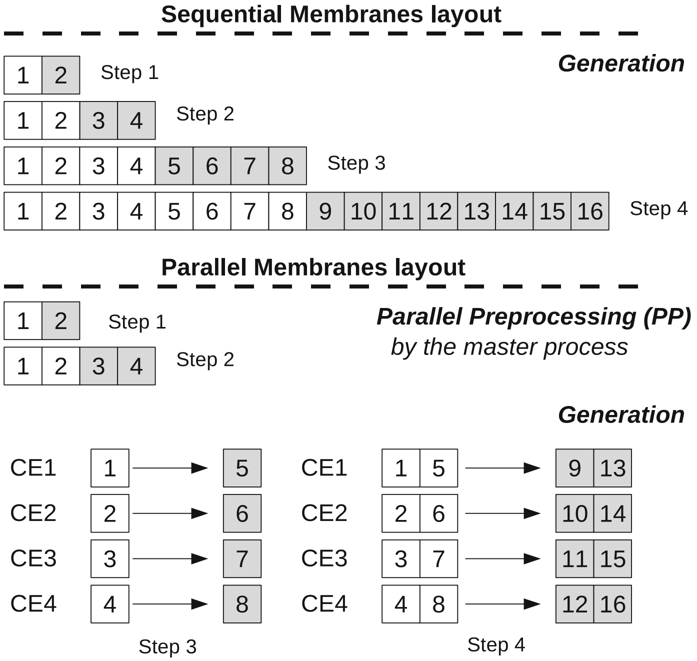

Fig. 2 Sequential and parallel membranes generation on four compute elements (*CE*). *Parallel Preprocessing (PP)* is required to set up the parallel execution

double level of parallelism and non-determinism that makes P systems a computational tool to solve NP-complete problems in polynomial time.

The first level of parallelism for the SAT P system is found among membranes, that is, by executing each membrane in parallel along the computation. The second level of parallelism is found within each membrane. That way, the first level is coarse grained and can be characterized by an intertask parallelism and exploited by the number of processors available in the parallel system, whereas the second level of parallelism is fine grained and intra-task is exploited by the number of cores within each processor, on multi- or manycore architectures.

Figure 2 illustrates parallelism among membranes, with the execution of the *Generation* function for the SAT P system in a sequential and parallel architecture composed of four compute elements (*CE*). In a parallel architecture, a set of membranes is initially created by the master process, whose size is equal to the number of *CE*s available during the execution. Then, a membrane is sent to each *CE* by the master processor in a step called *Parallel Preprocessing (PP)*, developed just before the *Generation* starts the computation on each *CE*. Here, a *CE* represents a processor (die) on each hardware platform, which can later be eventually decomposed into multi- or manycores when exploiting intra-task parallelism.

Figure 2 also shows that each membrane is always generated by the same membrane and in the same computational step regardless of the target architecture. For instance, membrane two is always generated by membrane one in the first computational step, membrane three is always generated by membrane one in the second step, and so on.

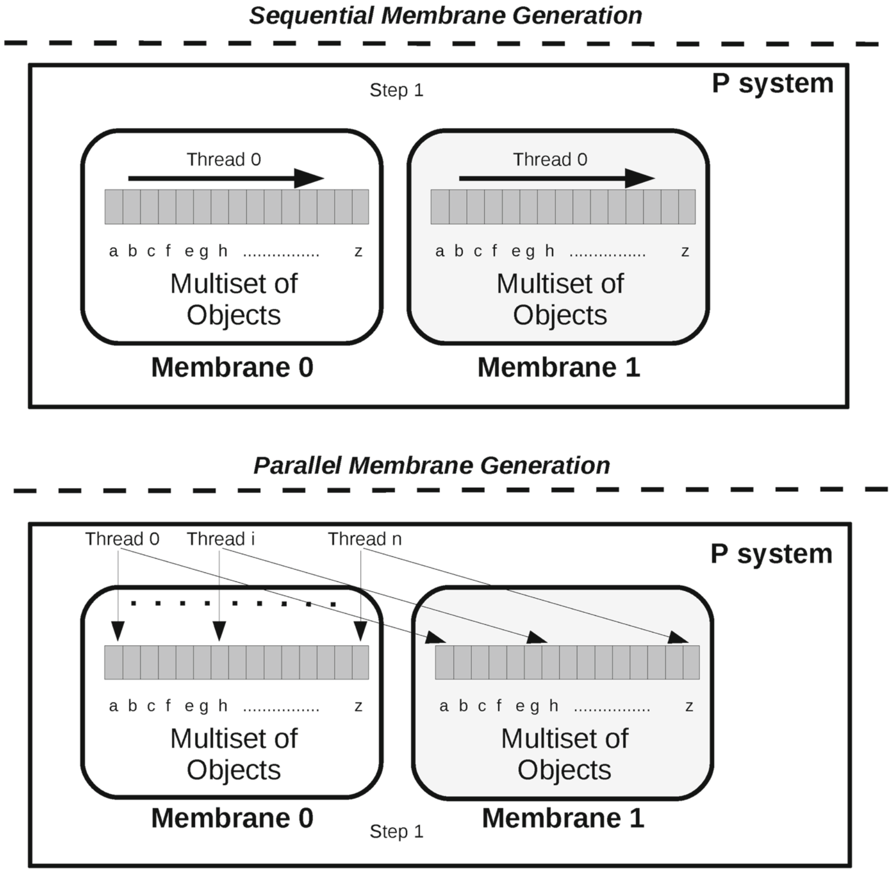

Fig. 3 Sequential and parallel execution when creating the exponential workload

Finally, each node sends the partial response back to the master to produce the final result of the P system.

The second level of P system parallelism (internal to membranes) is shown in Fig. 3. Basically, the simulation algorithm itself uses scatter/gather parallelism, with no intertask communication or coordination during the Generation step. The computation starts once the initial data have arrived at the *CE* after the *Parallel Preprocessing* step is over. Then the P system rules are applied for the SAT problem depicted in [21], and resources on each *CE* can be exploited at its peak to cooperate for speeding up the computation of the *Generation* and *CheckOut* functions. These resources are essentially hardware cores on shared memory, distributed memory and GPU platforms, but only GPUs are manycore, which can handle this level of parallelism at large scale using hundreds of streaming processors (see Table 1). An alternative offered by CPUs would eventually use a fine-grain level of parallelism to map on each CPU core the code simulating the behavior of each membrane.

Table 1 CUDA and hardware features for the Tesla C1060 and Tesla C2050 GPUs used during our experimental survey

Feature	Tesla C1060	Tesla C2050
Multiprocessors (SM)	30	14
Streaming processors (SP)/SM	8	32
Total number of SPs	240	448
32-Bit registers/SM	16,384	32,768
Shared memory/SM	16 kB	48 kB
Threads/SM	1,024	1,536
Threads/block	512	1,024
Threads/warp	32	32
Device (video) memory available	4 GB	3 GB

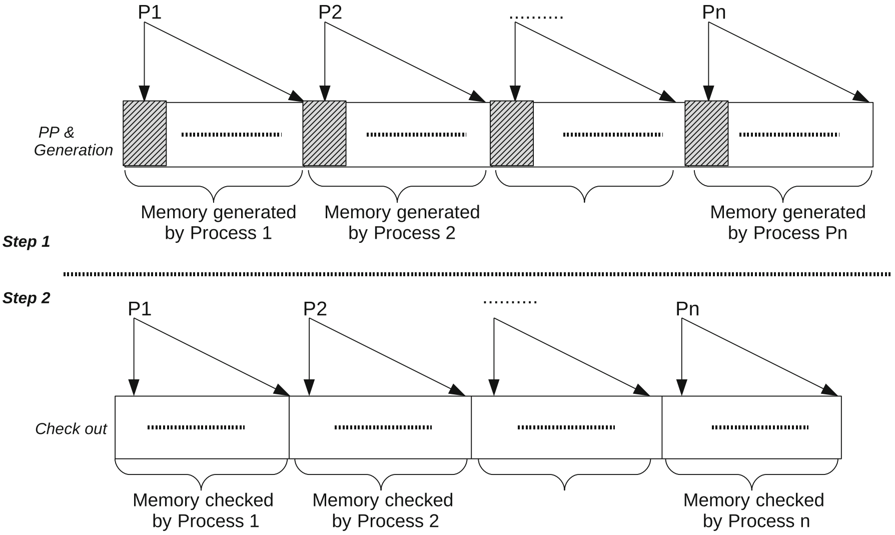
P1
P2
....
Pn
PP &
Generation
Memory generated by Process 1
Memory generated by Process 2
Memory generated by Process Pn
Step 1
Step 2
P1
P2
....
Pn
Check out
Memory checked by Process 1
Memory checked by Process 2
Memory checked by Process n

Fig. 4 Initial data placement for our shared memory implementation

4 Data policies description

Our P system simulator for the SAT problem organizes data depending on the features of the underlying architecture. We now describe those data policies.

4.1 The shared memory implementation

The simulator was implemented on the shared memory system using OpenMP [2]. Figure 4 shows the first data layout used by our simulator. The shared memory space is equally distributed among the *n* processes considered, and the master process performs the *Parallel Preprocessing* step by creating as many membranes as number of processors are involved in the computation. Membranes are placed at the beginning

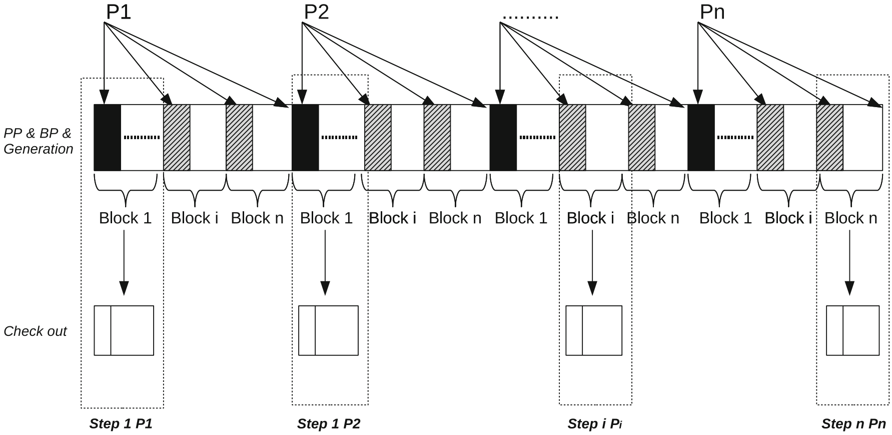

Fig. 5 The shared memory implementation for n processes using our block-based data layout

of the memory space assigned to each process (see gray squares in Fig. 4). Now, the *Generation* step is carried out by each individual process, writing the information on its own memory fragment. Once the membranes workspace has been created by the *Generation* stage, the *CheckOut* stage follows, where membranes are read again by processes to eventually produce the system response.

This data policy does not take advantage of data locality when the *Generation* and *CheckOut* stages are performed, thus producing many caches misses (in particular, read misses) that hit the simulator performance. Locality was improved through a block-based data layout as shown in Fig. 5. Again, the master process starts with the *Parallel Preprocessing* step, which generates as many membranes as processes (represented by black squares in Fig. 5). But now each process performs a local preprocessing step (called *Block Preprocessing*, (*BP*)) on its own memory space before starting the *Generation* stage itself. *Block Preprocessing* pursues a tiling or blocking execution between different stages of the simulation. Each process creates as many membranes as number of blocks, placing them at the beginning of each block position (represented by gray squares in Fig. 5). Then, the *Generation* stage only creates *blocksize* membranes before the *CheckOut* stage starts. Once a block has been checked out by the *CheckOut* stage, processes start again with the *Generation* stage of the following block which has been assigned to them.

The block-based data policy increases the time required by preprocessing, including a new *BP* stage, but a shorter data block can be placed in higher levels of the memory hierarchy, which benefits from data locality. However, there is a trade-off between preprocessing computation (*PP* and *BP*) and the data locality benefits for the *Generation* and *CheckOut* stages, being affected by the block size chosen.

4.2 The distributed memory implementation

The P system simulator for the SAT problem on the distributed memory system was programmed using MPI [1]. Again, we compare here a preliminary non-blocking version with an enhanced version based on a blocking data policy.

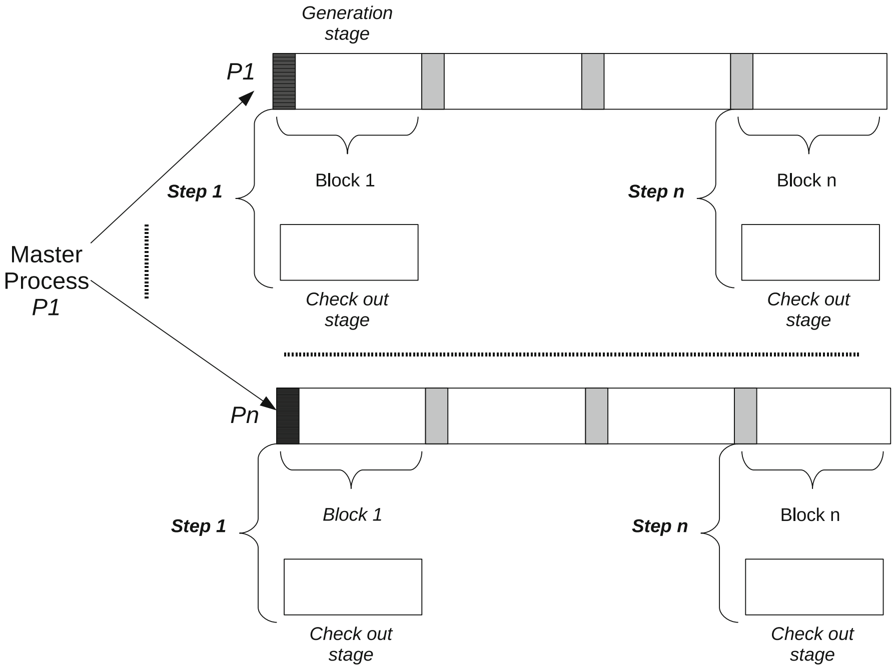

Fig. 6 P system simulation on a distributed memory architecture

In this case, each process allocates memory on its own and private memory space. The master process also performs the *Parallel Preprocessing* step, creating as many membranes as number of processors are involved in the computation. Then, membranes are sent to processors by using the MPI `Scatter` instruction.

Once the initial data arrive to each node, the P system computation was developed as in the shared memory case. For the non-blocking data policy, the *Generation* is fully performed before the *CheckOut* starts its computations. For the block-based data policy, the *Block Preprocessing* is required for a blocking or tiling execution. Figure 6 shows the data layout for the block-based data policy. Finally, a reduction is applied using the MPI `Reduce` instruction to end up with the system answer.

To take advantage of all hardware resources currently available on modern large-scale systems, we also provide a hybrid implementation of the simulator, which is programmed using both OpenMP and MPI programming models [15,23].

Similarly to the previous, the master process performs the *Parallel Preprocessing* step, creating as many membranes as number of *nodes* are involved in the computation. However, each node is now a shared memory platform, where several OpenMP processes collaboratively perform the simulation by also using blocking to exploit data locality.

With this implementation, the distributed architecture can also enable fine-grained parallelism, taking advantage of the multiple CPU cores available on each node, and thus simulating the behavior of each membrane.

4.3 Implementation on GPUs

The simulator sets a CUDA thread block for each membrane and a CUDA thread per object (or set of objects) in the multiset.

This time, the first attempt for the SAT P system simulation on GPUs, the *Generation* stage, is encoded as a CUDA kernel, and it starts right after the *Parallel Preprocessing* step. Once membranes have been generated, the *CheckOut* stage starts its execution. Each thread block loads a membrane from global memory and then each thread checks the rules associated with this stage. Finally, each block returns whether its associated membrane makes the CNF formula true or not. For these stages, all threads within a CUDA thread block cooperate with coalesced access to device memory (threads of the same warp access the same memory segment either for reading or writing).

Blocking can also be exploited on GPUs, taking advantage of the on-chip shared memory by using tiles with the aim of increasing the bandwidth to device memory (see Fig. 7). The simulation has to perform the *Block Preprocessing* step, which is implemented through a CUDA kernel where a set of membranes are partially created, placing them apart from each other at a block size distance.

An additional kernel is created this time at the end of the simulation. This kernel performs the *Generation* locally to each block, followed by the *CheckOut* stage. Each thread on a thread block cooperates for an efficient load from global memory to shared memory of the initial membrane generated by the *Block Preprocessing* step (represented by black squares in Fig. 7). Then, the *Generation* stage interacts with shared memory, saving expensive loads/writes from/to global memory which are around 400 times slower.

Finally, the *CheckOut* stage is performed over the data stored in shared memory after a block-level synchronization. This checks whether a clause makes true the CNF formula and writes its result into device memory.

Figure 8 shows the data policy used by the simulation of the P system for the SAT problem on a GPU-based platform. This simulator arranges data according to the "best practices" existing for CUDA-enabled devices [18]. Nevertheless, those guidelines are mainly focused on arithmetic intensive applications on a single GPU, and it remains to be seen whether they are valid on architectures like GPU-based clusters with a much higher degree of parallelism.

Within a GPU-based cluster of up to four GPUs such as those available on current motherboards for gamers, GPUs cannot interact with each other, and a CPU process has to be created to monitor each GPU independently. Note that this does not force us to use parallelism at CPU core level as long as we have exactly four CPUs in our system which can individually host each of the required processes. This way, our three implementations lack using the multithread capabilities of CPU cores.

Figure 8 shows how the master thread creates four CPU threads (CPU context) to invoke the execution on each GPU and manage its resources (i.e., allocate device memory, move data to/from the GPU, and so on). Resources created on each CPU thread are not accessible by any other thread, and there is no explicit initialization function for the runtime API [18], which makes it hard to measure time in a reliable manner, particularly on multi-GPU environments.

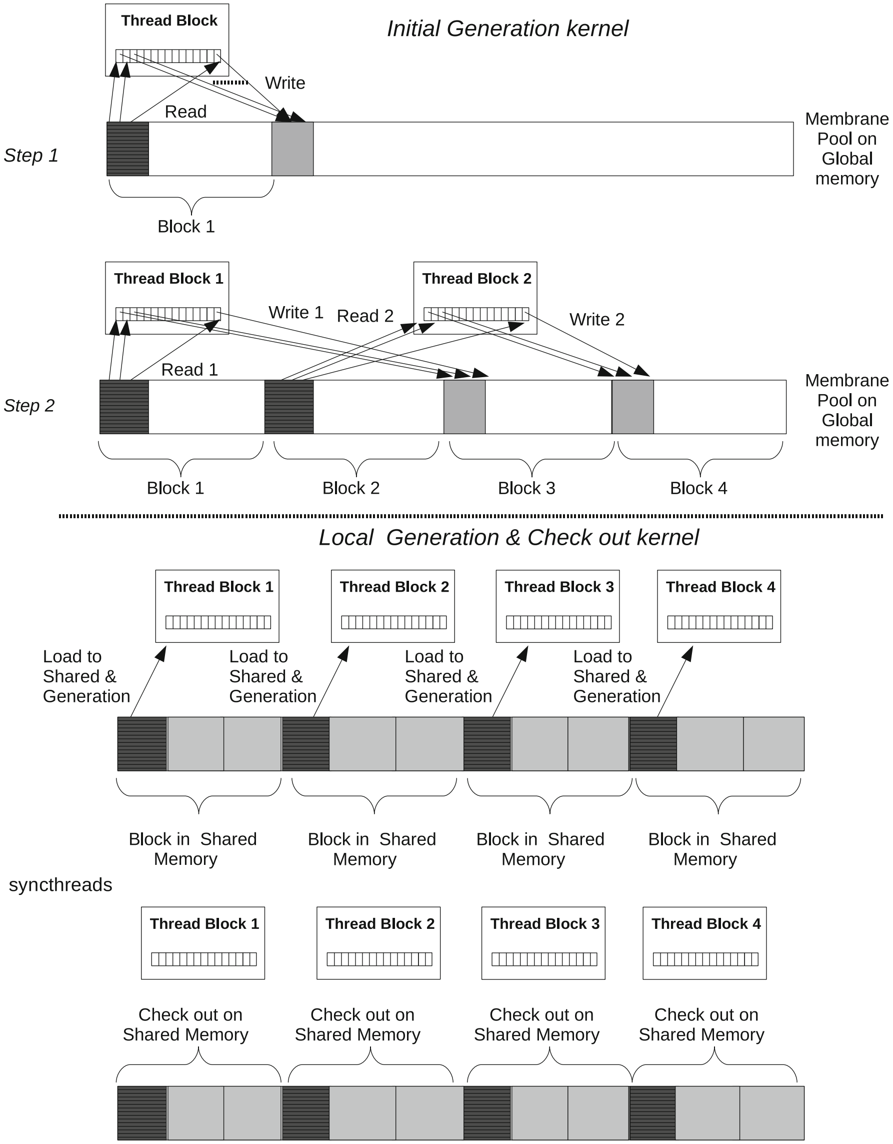

Fig. 7 P system simulation on a single GPU

For the GPU case, the master process performs the *Parallel Preprocessing* step as usual, generating as many membranes as GPUs are involved in the simulation and performing the assignment.

At a starting point, the simulation barely exploits GPU resources because the computation begins with a single CUDA thread block (which represents the membrane generated by the *Parallel Preprocessing* step). However, the number of CUDA thread blocks grows exponentially in the *Generation* stage along with the number of mem-

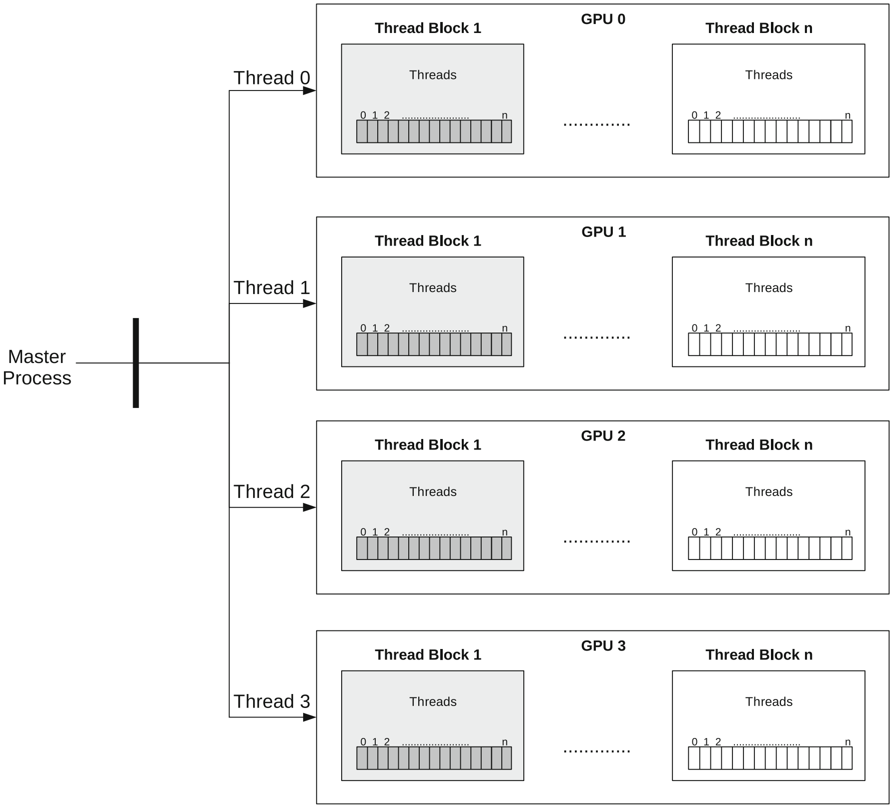

Fig. 8 Data policy on a GPU-based cluster of up to four GPUs. It may be implemented on a cheap motherboard targeted at gamers, where up to four PCI-e sockets are available

branes, and GPU resources are fully utilized at early stages of the simulation. Another alternative consists of creating a larger set of initial membranes in the *Parallel Preprocessing* step to fulfill that GPU resources are occupied right from the beginning, but we have tested that this initial low usage of GPU resources has a negligible impact, even on tiny benchmarks.

5 Performance evaluation

This section evaluates our P systems implementations in three different platforms. The shared memory platform is an HP Integrity Superdome SX2000 endowed with 64 CPUs, Intel Itanium 2 dual-core Montvale (16 kB L1, 256 kB L2, 18 MB L3). Total DRAM memory available is 1.5 TB and interconnection network is a 4× DDR Infiniband. The distributed memory system is an HP BladeSystem which contains up to 102 nodes and each node is a dual socket, each containing a quad-core Intel Xeon E5450 (Nehalem with a 12 MB L2 cache). DRAM memory capacity for the whole system is 1,072 GB. Interconnection network is also a 4× DDR Infiniband (see Table 2).

Table 2 Summary of hardware features for the architectures used during our experimental survey

	Shared memory	Distrib. memory	GPU-based
Hardware	HP Integrity	Hewlett-Packard	4 Intel Xeon E5530 CPU
Platform	Superdome SX2000	Blade System	(+ GPU shown in Table 1)
Number of nodes	1	102	1
CPU sockets per node	64	2	4
CPU cores per socket	2	4	4
CPU cores and speed	128 @ 1.6 GHz	816 @ 3 GHz	16 @ 2.4 GHz
Main memory (DRAM)	1,536 GB	1,072 GB	16 GB (+video memory)
Operating system	Linux 64 bits	Linux 64 bits	Linux 64 bits
Programming model	OpenMP	MPI	CUDA
Compiler	icc Intel 11.1	HP MPI 02.03.01	nvcc Nvidia 3.2

Finally, our GPU-based platform includes a four-socket, quad-core Intel Xeon E5530 (Nehalem with a 8 MB L2 cache), which acts as a host machine for our Nvidia Tesla GPUs, either a C1060 or a C2050, whose details are shown in Table 1.

Data policies and simulation performance are evaluated on each architecture under a set of benchmarks generated by the WinSAT program [22]. WinSAT can generate random SAT problems in DIMACS CNF format file by configuring several parameters: the number of variables (n), the number of clauses (m) and the number of literals per clause (k).

The number of membranes in our P system depends on the number of CNF variables, n (membranes $= 2^n$). We vary n from 13 variables (2^{13} membranes) to 25 variables (2^{25} membranes), whereas the number of literals ($l = m \times k$) is kept constant ($l = 256$ for benchmarks with $n < 22$ and $l = 200$ for benchmarks with $n \geq 22$). Doing so, we reduce memory requirements so that more benchmarks can be simulated on the GPU-based system. Memory requirements for each benchmark can be calculated according to Eq. 1.

$$\text{Size} = 2^n(\text{membranes}) \times l(\text{objects}) \times 4(\text{unit}) \text{ bytes} \tag{1}$$

5.1 The shared memory platform

A performance comparison between the blocking and non-blocking algorithm for 64 membranes per block is shown in Fig. 9. The blocking technique increases performance either with the problem size (i.e., the number of variables in the CNF formula for the SAT problem) or the number of computational processes (OpenMP processes created). The former is needed to hide the *Preprocessing time* (PP and BP), and the latter involves the memory coherence protocol: the network traffic in shared memory systems goes up with the number of cores, but the blocking technique takes advantage of the local data stored on each node to reduce the communications burden versus the non-blocking version.

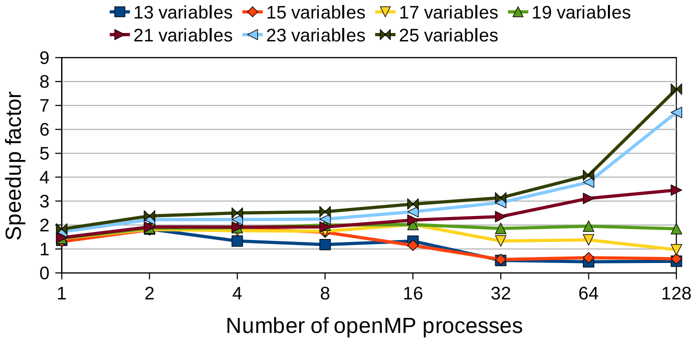

Fig. 9 Speedup factor achieved by the blocking algorithm when varying the number of variables

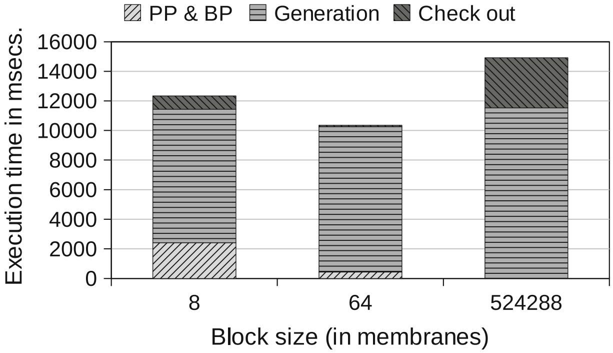

Fig. 10 Breakdown for the total execution time using eight OpenMP processes for a SAT problem composed of $n = 23$ variables and $l = 200$ literals

We now present some results about the simulation performance of the SAT P system, depending on the block size for the block-based data layout in our shared memory system. Figure 10 shows the breakdown for the total execution time in the three main functions performed by the OpenMP simulation, depending on the block size used by the blocking technique. We have checked many different block sizes to find the best configuration, but for the sake of simplicity Fig. 10 only shows three of them for the benchmark with $n = 23$ variables: the largest block size configuration, the shortest one, and finally the one scoring peak performance.

The largest block size ($2^{19} membranes/block$, up to 420 MB according to Eq. 1) is the most time-consuming configuration. The *Preprocessing* (PP and BP preprocessing) step is the least time-consuming for this configuration because only a few initial membranes are required in advance, but the *Generation* and *CheckOut* stages are heavier than in the other two configurations. *CheckOut* starts reading the first membrane right after the $2^{19} membranes$ of a block are generated by each process. Since the L3 cache size for the processor in our shared memory architecture system is 18 MB, many read and write cache misses occur in those stages, affecting the overall simulation performance.

Similarly, the smallest block size ($2^3 membranes/block$) shows the highest *Preprocessing* time. Although the *Generation* and *CheckOut* stages behave much better

Table 3 Execution times (in ms) for our P systems simulation on the shared memory architecture (OpenMP code)

Number of membranes	Number of OpenMP processes							
	1	2	4	8	16	32	64	128
2^{13}	132.08	65.49	46.32	32.98	28.67	67.59	123.04	191.92
2^{15}	506.40	265.38	132.96	85.00	75.00	110.36	154.09	209.35
2^{17}	1,887.95	977.63	514.97	288.65	156.93	145.58	158.32	243.96
2^{19}	7,450.16	3,785.42	1,894.35	979.19	539.13	366.85	303.77	322.46
2^{21}	29,281.70	14,766.00	7,447.95	3,745.47	1,916.27	1,023.37	616.87	576.12
2^{23}	78,996.90	39,796.60	20,049.70	10,101.90	5,153.65	2,677.78	1,544.76	997.02
2^{25}	307,756.10	156,617.02	78,758.81	39,794.82	20,227.01	10,428.92	5,812.66	3,307.33

We vary the number of membranes (by rows) and keep constant the number of literals, $l = 256$, for the block-based version

on cache misses, the simulation finds its best configuration for 2^6 membranes (50 kB) per block. This is the turning point between *Preprocessing* time and *Cache misses* (write and read misses) for this architecture.

Finally, Table 3 shows the execution time for the SAT P system simulation with the best configuration under the blocking technique. We executed several benchmarks varying the number of variables of the SAT problem, and also varied the number of OpenMP processes involved in the computation for each benchmark to study the scalability of the system.

The total execution time is given by the following equation:

$$t_{\text{total}} = t_{\text{prepro}} + t_{\text{cpu}} + t_{\text{overhead}} \tag{2}$$

where the first parameter (t_{prepro}) is the preprocessing time spent by the master process to create the initial set of membranes to be distributed among remaining processors. This is *Parallel Preprocessing* plus the preprocessing time required by each process to prepare the blocking execution (that is, *Block Preprocessing*), and it depends on two values: the number of processes and the block size. The second parameter (t_{cpu}) concerns the processing time taken by each node, and depends on the benchmark size. Finally, the last parameter (t_{overhead}) is the extra overhead added to the OpenMP execution time (i.e., synchronizations, loop scheduling, communications among processors, resource sharing, etc...). This parameter increases widely with the number of OpenMP processes.

From the experimental numbers we have obtained (see Table 3), we may draw some valuable remarks:

1. Processing time (t_{cpu} in Eq. 2) predominates over t_{prepro} and t_{overhead} when the problem size increases, so we have decided to present a combined value t_{total} for the sake of clarity.

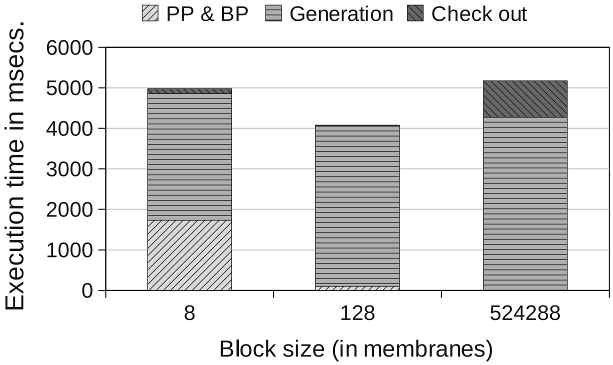

Fig. 11 Breakdown of the total execution time using 8 MPI cores with $n = 23$ variables and $l = 200$ literals

2. The scalability of the system is good on large problem sizes, but weak on smaller benchmarks (upper rows increase the execution time when moving from 64 to 128 OpenMP processes, a situation which does not arise on GPU computing). Nonetheless, these results on shared memory systems are encouraging given our high performance focus, as supercomputing using high-performance platforms is justified only on large-scale problems.
3. Our techniques are more effective on large number of membranes even when the working set for the algorithm exceeds the L3 cache size.
4. The software is more demanding on lower rows, and hardware is more powerful as we move to columns on the right. The last two rows and columns represent the maximum scalability and the best compromise between what the software expects and the hardware can contribute.

Note that this shared memory version only exploits the intra-task parallelism (that is, among membranes). The remaining stages for the simulation are sequentially performed on each node.

5.2 The distributed memory platform

In this case, the maximum speedup obtained by the best configuration for the blocking technique algorithm reaches up to 2× versus the non-blocking alternative, with this peak reached for the case of the $n = 25$ variables benchmark. Memory banks are independent on this platform, so the blocking algorithm takes advantage of data locality to improve memory bandwidth.

Regarding the optimal data block size, Fig. 11 shows the breakdown of the total execution time for the three main functions performed by the MPI simulation for the benchmark with $n = 23$ variables. Again, Fig. 11 shows only the largest, shortest, and best performance block size configurations. The optimal case here corresponds to 2^7 membranes per block (100 kB per block).

Table 4 shows the execution time for the MPI code, taking the best configuration blocking technique and varying the number of variables of the SAT problem and

Table 4 Execution times (ms) for our P systems simulation on the distributed memory architecture

Number of membranes	Number of MPI processes							
	1	2	4	8	16	32	64	128
2^{13}	47.21	20.81	7.42	4.05	2.36	1.83	1.49	1.54
2^{15}	164.42	82.49	29.10	15.22	7.94	4.47	2.80	2.78
2^{17}	454.98	296.61	116.71	69.33	30.68	15.57	8.42	6.00
2^{19}	1,857.39	942.88	496.86	239.82	124.27	61.83	37.70	19.33
2^{21}	7,128.02	3,544.49	1,874.44	953.57	475.95	243.10	154.43	83.89
2^{23}	19,387.70	9,828.17	4,954.03	2,737.77	1335.51	668.66	364.98	220.00
2^{25}	n.a.	37,959.00	19,459.60	9,988.04	5,375.26	2,697.75	1,390.71	720.89

We vary the number of membranes (by rows) and keep constant the number of literals, $l = 256$, for the block-based version. n.a. means "not available" due to device memory constraints

the number of MPI processes. Results on a single core are missing for the largest benchmark (that of $n = 25$ variables), because the memory available on a single node is not enough to run the simulation (the benchmark allocates up to 26 GB and the maximum memory per node is 16 GB).

Similarly to Table 3, the total execution time given by Eq. 2, t_{total}, was used as metric for showing results. Minor differences are seen based on the architectural features of each system, with the overhead being influenced by communications among processors. Data sent to each processor by the master is a single membrane, and the result returned by each node is just a boolean, saying whether or not a solution is found. Note that this version does not exploit the intertask parallelism either: each membrane is sent to a node and simulations are executed sequentially on that node.

Our analysis of these results can be summarized as follows. The system scalability improves again with the problem size, but this time scales much better than in the OpenMP case and does not suffer from small benchmarks or large number of nodes. Only the first row increases slightly when moving to 128 nodes, but the situation immediately reverses. More importantly, large problem sizes and number of nodes keep scalability at high standards, overcoming those weaknesses showed on shared memory problems. Thus, we certify that on memory-bound problems distributed memory architectures are a better alternative to the use of shared memory when the parallelization strategy can find enough number of independent task, and usually at a lower infrastructure cost. Given that GPUs combine small amounts of shared memory within thread blocks with large quantities among multiprocessors (the global memory), it is interesting to see at this point what they have to say concerning performance and scalability.

5.3 The GPUs

Figure 12 shows the breakdown of the total execution time for a Tesla C1060 GPU executing the benchmark with $n = 21$ variables and using a tiling version. It shows that

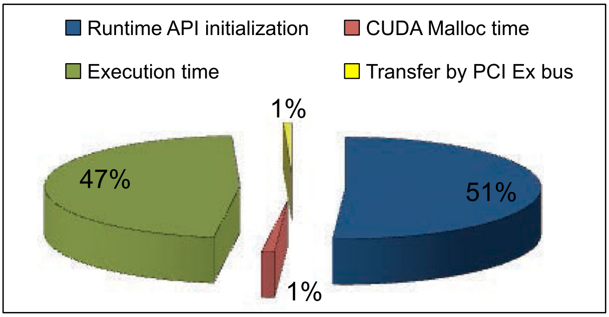

Fig. 12 Breakdown of the total execution time in a Tesla C1060 GPU with $n = 21$ variables

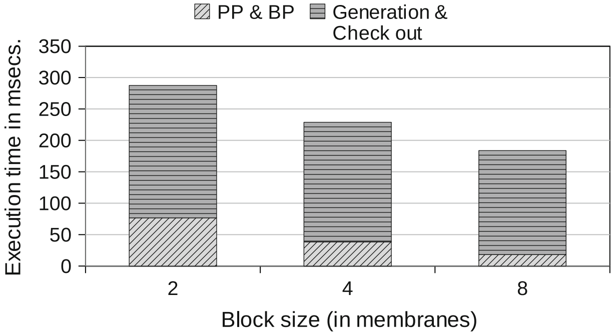

Fig. 13 Breakdown for the execution time on a single GPU with $n = 23$ variables and $l = 200$ literals

the 51 % of total execution time is spent by the runtime API initialization on average and only 47 % corresponds to the actual execution time. The runtime API initialization penalty is not usually considered when timing GPU applications, because it is not stable between different executions or related to the actual GPU computation. But in our case it represents half of the total execution time. Data transfers are not that important here and lose the leadership shown on previous platforms.

We first evaluate the impact of the data block size. Figure 13 shows the breakdown of the total execution time for the two main kernels performed by the GPU simulation. The block size is now limited by the on-chip shared memory space (16 kB for Tesla C1060, 48 kB for Tesla C2050). Simulations are tested for two, four and eight membranes per block, reaching the best performance for the last case.

The number of global memory accesses and the number of iterations in the *Block Preprocessing* kernel intrinsically depends on block size. In particular, eight membranes per block require half of the memory accesses and iterations as compared to the four membranes per block configuration, which, similarly, cut down to a half those required by the two membranes per block case. Figure 13 reflects this fact.

Likewise, memory accesses in the *Generation* and *CheckOut* stages are reduced in a similar way as long as the block size increases. However, the GPU resource occupancy worsens for the eight membranes per block case, because the shared memory usage per block prevents allocating more than one block per SM. As a result, the overall improvement is just 14% over the four membranes per block configuration.

Table 5 Execution times (ms) for our P systems simulation on different CUDA versions and GPU platforms

Number of membranes	GPU enhanced		GPU tiled	
	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050
2^{13}	0.82	0.62	0.64	0.37
2^{14}	1.55	1.20	1.15	0.66
2^{15}	2.90	2.30	2.17	1.24
2^{16}	5.65	4.37	4.23	2.37
2^{17}	11.16	8.71	8.29	4.65
2^{18}	22.06	17.15	16.46	9.19
2^{19}	44.69	33.16	32.79	18.27
2^{20}	88.48	89.03	65.51	36.65
2^{21}	171.04	127.85	124.29	69.76
2^{22}	298.26	n.a.	178.18	n.a.

We vary the number of membranes and keep constant the number of literals, $l = 256$. There are eight membranes per CUDA block in the Tesla C1060, and 16 in the Tesla C2050. n.a. means "not available" due to device memory constraints

Table 5 shows the performance for the tiling version of the GPU simulator with eight membranes per block in the Tesla C1060 GPU and 16 membranes in the Tesla C2050 GPU. We vary the problem size (by rows) to study the scalability for the system. Note that video memory constraints prevent us from executing the code beyond n=23 variables, whose memory requirements are 6,400 MB. Moreover, those times do not account for overheads like *initial* and *final* data transfers between CPU and GPU, GPU memory allocation, and CUDA runtime initialization, which may be significant in practice.

GPUs improve significantly the device memory bandwidth through shared memory usage, which is explicitly used by the CUDA programmer. This way, one can control the number of accesses and the way to access memory-bounded applications like ours. Even though the small size of the shared memory decreases GPU occupancy, the benefit of reducing the number of accesses to device memory is much higher and this strategy is widely rewarded: we see that the tiling technique obtains up to 2.43× speedup factor versus the non-tiling counterpart (see the penultimate row in Table 5).

5.4 Overall comparison

Figure 14 summarizes the performance for all our implementations, where GPU numbers have been extended from 1 to 2 and 4 GPUs. That way, we can study the scalability in all platforms: for the OpenMP case, doubling the number of threads on each step forward that is represented on the X axis within each data set interval; for the MPI case, doubling the number of processes involved; and for CUDA, doubling the number of GPUs. A single context is maintained per physical core to minimize communications and avoid additional overhead for the exponential workspace generated in the simulation. On the other hand, the Y axis depicts the execution time in $\log_{10}$ scale. With

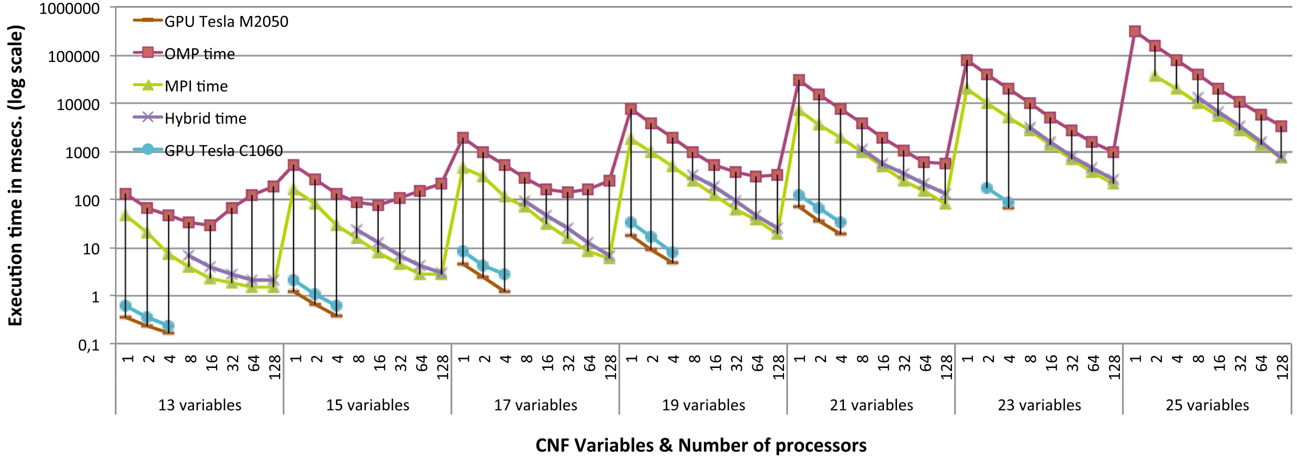

Fig. 14 Execution time for the three different programming models and architectures: CUDA on GPUs, OpenMP on a shared memory system and MPI on a distributed memory platform

log scales being used on both dimensions, it is not that easy to keep track of hardware scalability, but the important issue here is that all architectures draw parallel lines on its evolution when doubling the hardware resources involved, reflecting a similar behavior in this respect.

Essentially, the simulation algorithm itself uses scatter/gather parallelism, with no intertask communication or coordination during the Generation step. This produces good (linear) scaling on distributed memory systems as only a front-end collective scatter and a back-end collective are required to accumulate the final solution. The hybrid version introduces the overhead of creating OpenMP processes within each node, and for this particular problem where the number of communications is too low, this is paid by obtaining even higher execution times than the only MPI case. We can also see that the relative GPU performance is enhanced as we increase the problem size, reaching its peak for $n = 23$ variables, where the problem size exceeds the video memory available for a single GPU.

Starting with the smallest problem size we have run, execution times for $n = 13$ variables are 132.08 ms for a single OpenMP thread, 47.21 ms for a single MPI process and 0.82 ms on a single GPU. The GPU acceleration for this case reaches 161.07× versus OpenMP and 57.57× versus MPI. However, considering the largest problem size and enabling the maximum amount of parallelism, we were able to expose on the three parallel platforms for a fair comparison ($n = 23$ variables and four processors) that execution times were 20049.70 ms using OpenMP on the shared memory multiprocessor, 4954.03 ms using MPI on the distributed memory multiprocessor and 64.51 ms using CUDA in a set of four Tesla C2050 GPUs (actually, a Tesla S2050 server). Those numbers are translated into more than 310× factor versus the distributed memory system and 78× versus the shared memory platform for a much cheaper high-performance alternative (the cost for a Tesla S2050 server is around 8,000 euros).

Our work identifies GPUs as the best choice by a wide margin for running the simulation of P systems, and scalability benefits large problem sizes which are more realistic within this research area. On the hardware size, the three architectures evaluated here pose different schemes for connecting processors and memories. Results demonstrate that a proper combination of shared memory, strategically placed in small amounts within distributed GPU multiprocessors and propelled by programmer's skills, is a winner card in memory-bound problems when data partitioning finds a clean decomposition in independent blocks. One may think that distributed memory platforms have a clear chance to succeed under this scenario, but the huge number of CUDA blocks deployed guarantees hiding latencies to global memory while keeping all GPU multiprocessors busy, leading to an extraordinary processor occupancy which is unbeatable on a large number of nodes.

6 Summary and conclusions

In this article, we have described the simulation of a family of recognizer P systems with active membranes, solving the satisfiability (SAT) problem, on three different parallel architectures based on shared memory, distributed memory and a set of GPUs. We have also used three different programming models: OpenMP, MPI and CUDA, respectively. Our goal here was twofold: (1) analyze pros and cons of those architectures and programming models for natural computing and (2) compare them using a representative algorithm within this field. Considering the largest problem size we were able to run on the three parallel platforms, execution times were 20,049.70 ms using OpenMP on the shared memory multiprocessor, 4954.03 ms using MPI on the distributed memory multinode and 64.51 ms using CUDA in a Tesla C2050 GPUs. This leads to execution time improvements exceeding 310 and 78×, respectively, for a much cheaper high-performance alternative.

Along our journey, we have also introduced a number of variants in the implementation to enrich our discussion. For example, a blocking/tiling strategy for the data placement takes advantage of data locality and leads to a more effective use of bandwidth available in all targeted systems. Performance varies depending of the memory architecture and the way to manage it, but GPUs are ahead in this respect due to explicit control mechanisms from a programming viewpoint, combined with wider memory buses (256–512 bits) and higher frequency rates (around 2 GHz on GDDR5 video memory technology) on the architectural side.

Additional efforts are required on each specific architecture to reduce the cost of preprocessing steps, with a different reward for each case too.

Remarkable findings also worth mentioning can be summarized as follows:

- **Shared memory architecture.** The blocking technique improves the parallel efficiency, but the OpenMP simulator reaches the lowest performance as the pressure on shared resources increases with the number of processors. On the positive side, this was the only platform where we were able to execute all benchmarks due to higher memory availability, and the fact that we sacrifice DRAM in our P system implementation to provide a more general solution makes memory resources become valuable when simulating large-scale systems.
- **Distributed memory systems.** They exhibit good scalability with the number of processors, which can be partially explained by the low number of communications required by our simulations. This also explains the lack of scalability provided by the hybrid version that exploits all the resources within a node. In general, the MPI solution provides a balance between performance and memory consumption.
- **GPUs.** The two levels of parallelism that P systems exhibit, one at the region level and the other at the system level, were exploited using CUDA to map perfectly into data parallelism at core level and task parallelism among graphics cards leading to the best performance overall. However, we run out of video memory for executing the largest problem sizes, as manycores cannot gather similar amounts of DRAM memory as CPU multiprocessors in the current state of its evolution.

In the near future, we will work in two different directions: evaluating new hardware frontiers and improving the definition of computational models to deal with a broader set of problems. From the hardware side, we expect that the combination of cloud computing and heterogeneous systems will provide us a likely path for increasing the memory size without sacrificing performance at all [25]. This grants us a realistic hope for running larger simulations and more sophisticated algorithms within natural computing in the years to come. Moreover, hardware-based implementations can be a good alternative to consider. From the algorithm point of view, we are studying a hybrid implementation of P systems that uses heuristics to improve the particular solution of the modeled problem.

Acknowledgments This work was jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 15290/PI/2010, the Spanish MEC and European Commission FEDER under grant TIN2012-31345, the Junta de Andalucía under Project of Excellence P12-TIC-1741, the Universidad Católica San Antonio de Murcia (UCAM) under grant PMAFI/26/12 and the supercomputing infrastructure of the NLHPC (ECM-02). We also thank NVIDIA for hardware donation under CUDA Teaching Center 2011-14, CUDA Research Center 2012-14 and CUDA Fellow 2012-14 Awards. Finally, we want to thank all reviewers of this paper for the valuable suggestions they gave us to improve the overall quality of this paper.

References

1. Message Passing Interface (MPI). http://www.mcs.anl.gov/mpi
2. The OpenMP Specification. http://www.openmp.org
3. Alonso S, Fernández L, Arroyo F, Gil J (2008) A circuit implementing massive parallelism in transition P systems. Int J Inform Technol Knowl 2(1):35–42
4. Asanovic K, Bodik R, Catanzaro B, Gebis J, Joseph J, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. University of California, Berkeley, EECS Department
5. Asghar S, Aubanel E, Bremner D (2013) A dynamic moldable job scheduling based parallel SAT solver. In: Proceedings of international conference on parallel processing (ICPP), pp 110–119
6. Borkar S, Jouppin NP, Stenstrom P (2007) Microprocessors in the era of terascale integration. In DATE 07: Proceedings of the conference on design, automation and test in Europe, EDA Consortium San Jose, CA, USA, ACM, pp 237–242
7. Cecilia JM, García JM, Guerrero GD, del Amor MAM, Pérez-Hurtado I, Pérez-Jiménez MJ (2010) Simulating a P system based efficient solution to SAT by using GPUs. J Logic Algebraic Program 79(6):317–325
8. Cecilia JM, García JM, Guerrero GD, del Amor MAM, Pérez-Hurtado I, Pérez-Jiménez MJ (2010) Simulation of P systems with active membranes on CUDA. Brief Bioinform 11(3):313–322
9. Cecilia JM, García JM, Guerrero GD, del Amor MAM, Pérez-Jiménez MJ, Ujaldón M (2012) The GPU on the simulation of cellular computing models. J Soft Comput 16(2):231–246 (Special issue on evolutionary computation on general purpose graphics processing units)
10. Cook SA (1971) The complexity of theorem-proving procedures. In STOC '71: Proceedings of the third annual ACM symposium on Theory of computing, New York, NY, USA, ACM, pp 151–158
11. Díaz D, Graciani C, Gutiérrez-Naranjo MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2009) Software for p systems. In: Paun Gh, Rozenberg G, Salomaa A, (eds) The Oxford handbook of membrane computing, pp 437–454. Oxford University Press, Oxford
12. García-Quismondo M, Gutiérrez-Escudero R, Pérez-Hurtado I, Pérez-Jiménez MJ, Riscos-Núñez A (2010) An overview of p-lingua 2.0. Lecture Notes Comput Sci 5957:264–288
13. Garland M, Kirk DB (2010) Understanding throughput-oriented architectures. Commun ACM 53(11):58–66
14. Hwu W (2011) GPU computing gems: Emerald Edition. Morgan Kaufmann, Los Altos
15. Jin H, Jespersen D, Mehrotra P, Biswas R, Huang L, Chapman B (2011) High performance computing using MPI and OpenMP on multi-core parallel systems. J Parallel Comput 37(9):562–575
16. Meyer Q, Schonfeld F, Stamminge M, Wanka R (2010) 3-SAT on CUDA: towards a massively parallel SAT solver. In: Proceedings of IEEE international conference on high performance computing and simulation (HPCS), pp 306–313
17. Nguyen V, Kearney D, Gioiosa G (2010) An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. J Logic Algebraic Program 79(6):383–396
18. NVIDIA (2011) CUDA Programming Guide 4.0
19. Paun G (2002) Membrane computing. An introduction. Springer, Berlin, pp 9–419
20. Paun G, Centre T, Science C (1998) Computing with membranes. J Comput Syst Sci 61:108–143
21. Pérez-Jiménez MJ, Romero-Jiménez Á, Sancho-Caparrini F (2003) Complexity classes in models of cellular computing with membranes. J Nat Comput 2(3):265–285
22. Qasem M (2009) WinSAT website. http://users.ecs.soton.ac.uk/mqq06r/winsat

23. Rabenseifner R, Hager G, Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: PDP 2009: Proceedings of the 17th euromicro international conference on parallel, distributed, and network-based processing. Computer Society Press, Weimar, pp 227–236
24. Shi Z (2011) Stochastic modeling, correlation, competition, and cooperation in a CSMA wireless network. ProQuest, UMI Dissertation Publishing
25. Trieflinger S (2013) High performance peer-to-peer desktop grid computing: architecture, methods and applications. PhD dissertation. University of Stuttgart