P systems simulations on massively parallel architectures *

José M. Cecilia
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
chema@ditec.um.es

José M. García
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
jmgarcia@ditec.um.es

Ginés D. Guerrero
Computer Engineering and Technology Department
University of Murcia
30100 Murcia, Spain
gines.guerrero@ditec.um.es

Miguel A. Martínez–del–Amor
Computer Science and Artificial Intelligence Dept.
University of Seville
41012 Seville, Spain
mdelamor@us.es

Mario J. Pérez–Jiménez
Computer Science and Artificial Intelligence Dept.
University of Seville
41012 Seville, Spain
marper@us.es

Manuel Ujaldón
Computer Architecture Department
University of Malaga
29071 Malaga, Spain
ujaldon@uma.es

ABSTRACT

Membrane Computing is an emergent research area studying the behaviour 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 challenges in three different aspects: an intrinsic massively parallelism of P systems, an exponential computational workspace, and a non-intensive floating point nature. In this paper, we analyze the simulation of a family of recognizer P systems with active membranes that solves the Satisfiability (SAT) problem in linear time on three different architectures: a shared memory system, a distributed memory system, and a set of Graphics Processing Units (GPUs). 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: Considering the largest problem size we were able to run on the three parallel platforms involving four processors, execution times were 20049.70 ms. using OpenMP on the shared memory multiprocessor, 4954.03 ms. using MPI on the distributed memory multiprocessor and 565.56 ms. using CUDA in our four GPUs, which results in speed factors of 35.44x and 8.75x, respectively.

Keywords

Multicore, Manycore, GPUs, P systems, SAT problem, High Performance Computing

*This work was supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Spanish MEC and MICINN under project TIN2009-13192, by the FEDER funds of the European Community, under grants CSD2006-00046 and TIN2009-14475-C04, and by the Junta of Andalucia of Spain under projects P06-TIC02109 and P08-TIC04200. We would also like to acknowledge the support of the Centro de Supercomputación de la Región de Murcia where our experiments were conducted.

1. INTRODUCTION

Parallel computing architectures have brought dramatic changes to mainstream computing. 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 are no longer relying on a single chip design, but being partitioned among a bunch of simpler cores. Multicore CPUs are holding a dozen of cores, and manycore GPUs gather a myriad of stream processors. These components are being combined to build heterogeneous parallel computers offering a wide spectrum of high speed processing functions. Major hurdles to exploit this raw power are the PCI express bus to communicate the CPU and the GPU as they do not share the memory space, and also their different parallel programming approaches and paradigms. These problems amplify when we move to heterogeneous clusters.

This paper 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 simulation characterizes Membrane Computing, an emergent research area which studies the behaviour 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 of 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 [8, 7], most of these simulators were developed for sequential architectures using languages such as Java, CLIPS, Prolog or C, where performance is hardly compromised.

Section 2 of this article introduces Membrane Computing and describes the behaviour of this biologically inspired way of computation, focusing on computational devices called P systems to solve the Satisfiability (SAT) problem. This behaviour is simulated on different architectures, namely, a shared-memory architecture (HP Superdome), a distributed-memory machine (cluster of HP Blades), and finally a set of Nvidia Tesla GPUs. 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 Section 4 how GPUs can accommodate two levels of parallelism in its computational model versus a single level on shared and distributed memory systems.

The nature of P system computation creates an exponential workspace leading to polynomial time solutions for NP-complete problems. Section 5 analyzes different data policies to increase the memory bandwidth, and also to take advantage of the data locality on each architecture by providing a blocking/tiling algorithm. We also get a glimpse of the memory limitations on each system to simulate larger datasets and benchmarks. The GPU memory is very limited compared to the other alternatives, and the only way to include more GPU memory is actually adding more GPUs to the system. Finally, Section 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

Gh. Păun introduced Membrane Computing in 1998 [12], and since then, this bio-inspired computing paradigm has attracted research activities within Natural Computing. The model starts with the assumption 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 multisets of objects, that is, sets of objects with multiplicities associated to the elements.

P systems have several syntactic elements (see Figure 1): First, a membrane structure consisting of a hierarchical arrangement of membranes embedded in a skin membrane, which delimits the internal region of the P system from the environment. Second, delimiting regions or compartments where multisets of objects (corresponding to chemical substances) and sets of evolution rules (corresponding to reaction rules) are placed. Every membrane has associated an unchangeable label, and depending on the P system model, it may also contain a charge or polarization that can be modified during the computation. Besides, P systems possess two valuable features: inherent parallelism and non-determinism.

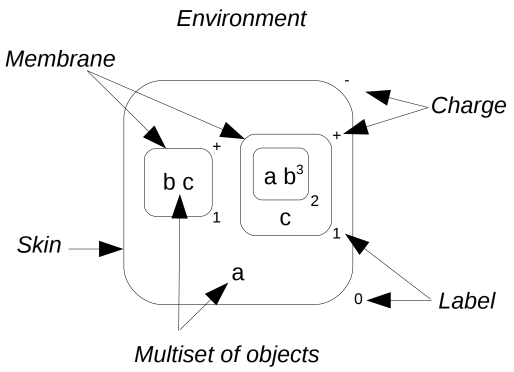

Figure 1: The structure of a P system.

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

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

The two level parallelism and non-determinism can be used 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.

Up to date, there have not been *in vivo* nor *in vitro* implementations of P systems, and researchers have focused on simulators developed in silicon whose initial versions were targeted to sequential platforms [7, 8]. From this departure point, the main challenge for the simulations of P systems in general is to find the right platform to exploit massively the parallelism inherent to the definition of P systems.

In this respect, several efforts have been done implementing this massively parallelism on parallel architectures. For instance, Alonso et al. [3] proposed a circuit implementation for the class of transition P systems. Moreover, Nguyen et al. [9] proposed an implementation of transition P systems in FPGAs, providing several levels of parallelism, one at rule level and other at region level, releasing a software framework for Membrane Computing called Reconfig-P. A generic simulator on GPUs for a family of recognizer P system with active membranes was presented in [5], showing that the double level of parallelism exposed by GPUs represents a valid alternative to simulate P systems.

2.2 The Satisfiability (SAT) problem

Propositional Satisfiability problem (`SAT`) was the first known **NP**-complete problem, as proven by Stephen Cook in 1971 [6]. 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 for which that formula evaluates to 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 of the size of the input $(max\{m,n\})$ in the worst case. 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 [13], which can be summarized as the following list of stages:

1. **Generation.** Membranes are structured within a rooted tree with a single branch. The root node is the *skin membrane*, and the second node is called *internal membrane*. All possible truth assignments to the variables are generated by using division rules, and they are encoded in the internal membranes by executing step by step the set of P system rules already described in [13]. In this way, 2^n internal membranes are created such that each one encodes a truth assignment to the variables of the formula.

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 (that is, by the assignment that represents).

4. **Output.** Internal membranes encoding a solution send an object to the skin. If the skin has such object from some membrane, the object Yes is sent to the environment. Otherwise, the object No is sent.

Algorithm 1 summarizes the sequential code based on previous stages. First, *Generation* and *Synchronization* are the stages creating an exponential workspace of membranes in a synchronous way, and also unifying the objects that codify a partial solution. Both stages are executed in the same function, which is referred to as *Generation* from now on. Note that each membrane runs in parallel at each iteration of *Generation*, but a global synchronization is required by different iterations.

Once the workspace is created, the *Check out* and *Output* stages are performed. First, they determine the clauses being true in every internal membrane, and then they 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
```

The specific simulation of the family of P systems that solves `SAT` for a single GPU is analyzed in [4], where problems to carry out the theoretical simulation of P systems on GPUs are depicted, and some heuristics to accelerate its computation are provided.

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 [11]. Among them, we highlight the theoretical double level of parallelism and non-determinism that makes P systems a computational tool to solve NP-complete problems in polynomial time.

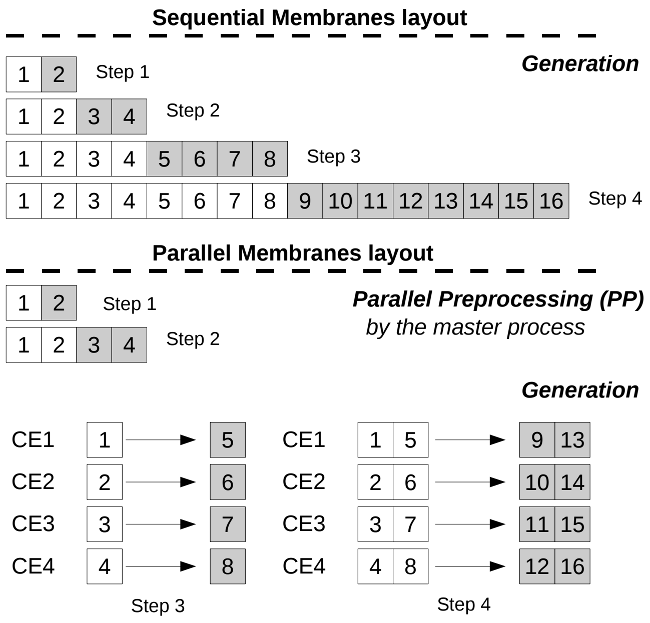
Sequential Membranes layout
Generation
Step 1
Step 2
Step 3
Step 4
Parallel Membranes layout
Parallel Preprocessing (PP) by the master process
Generation
CE1
CE2
CE3
CE4
Step 3
Step 4

Figure 2: Sequential and parallel membranes generation on four Compute Elements (CE). The *Parallel Preprocessing (PP)* is required to set up the parallel execution.

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 inter-task 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 to be exploited by the number of cores within each processor, either on multi- or many-core architectures.

The membrane parallelism is showed in Figure 2. It shows the execution of the *Generation* function for the SAT P system in a sequential as well as a parallel architecture with four Compute Elements (CE). In a parallel architecture, a set of membranes is initially created by the master process, whose size is equal to the number of CEs available during the execution. Then, a membrane is sent to each CE by the master processor. This step is called *Parallel Preprocessing (PP)*, and it is developed just before the *Generation* starts the computation on each CE. This CE is represented by a processor (die) on each hardware platform, which can later be eventually decomposed into multi- or many-cores when exploiting intra-task parallelism.

Furthermore, Figure 2 shows that each membrane is always generated by the same membrane and also in the same computational step on every 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. Finally, each node sends the partial response back to the master in order to produce the final result of the P system.

Figure 3 shows the second level of P system parallelism (that internal to membranes). Once the initial data has arrived to the CE after the *Parallel Preprocessing* step, it

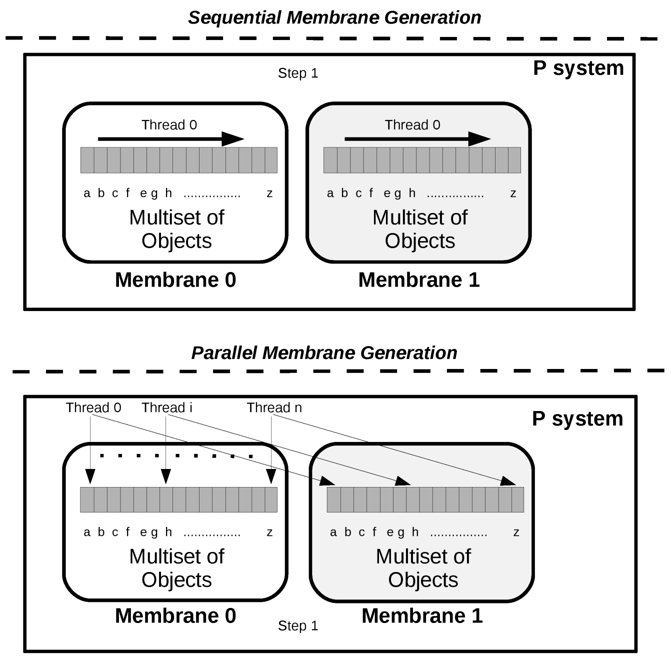

Figure 3: Sequential and parallel execution when creating the exponential workload.

starts the computation according to algorithm 1, and applying the P system rules for the SAT problem depicted in [13]. Then, resources on each CE can be exploited at its peak to cooperate for speeding up the computation of the *Generation* and *CheckOut* functions. This 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).

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 of the memory space assigned to each process (see gray squares in Figure 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 Figure 5.

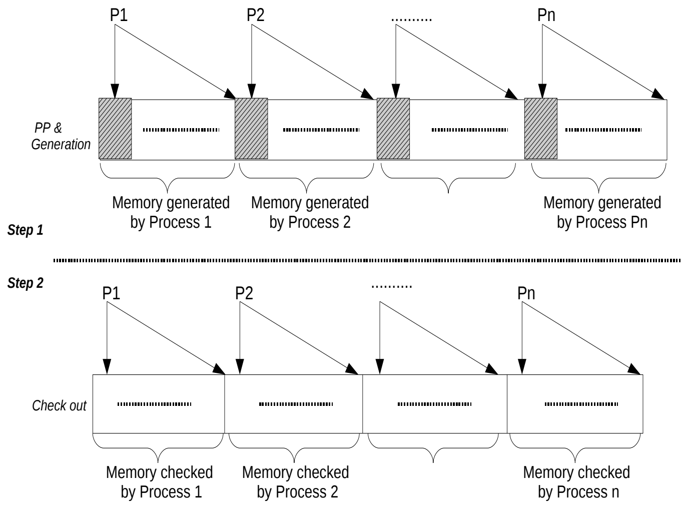
P1
P2
....
Pn
PP & Generation
Memory generated by Process 1
Memory generated by Process 2
Memory generated by Process Pn
Step 1
Step 2
Check out
Memory checked by Process 1
Memory checked by Process 2
Memory checked by Process n

Figure 4: Initial data placement for our shared memory implementation.

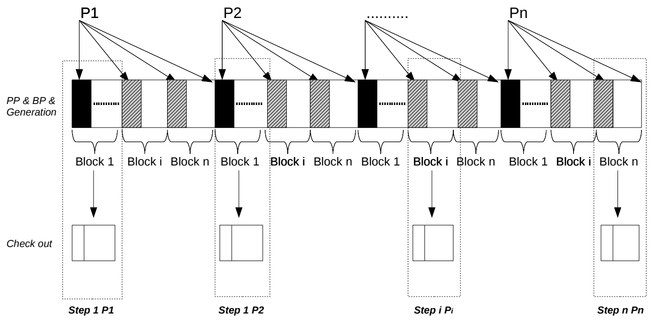

Figure 5: The shared memory implementation for n processes using our block-based data layout.

Again, the master process starts with the *Parallel Preprocessing* step, which generates as many membranes as processes (represented by black squares in Figure 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 Figure 5). Then, the *Generation* stage only creates *blocksize* membranes before the *CheckOut* stage starts. Once a block has been checked by the *CheckOut* stage, processes start again the *Generation* on the following block it has assigned to.

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].

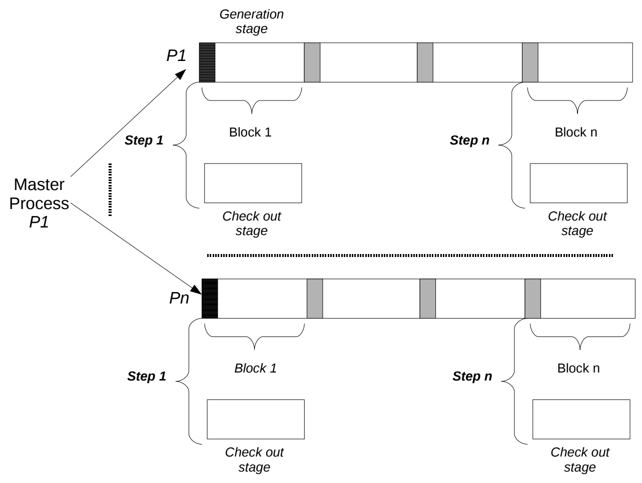

Figure 6: P system simulation on a distributed memory architecture.

Again, we compare here a preliminary non-blocking version with an enhanced version based on a blocking data policy.

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 arrives 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.

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 true the CNF formula 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 Figure 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

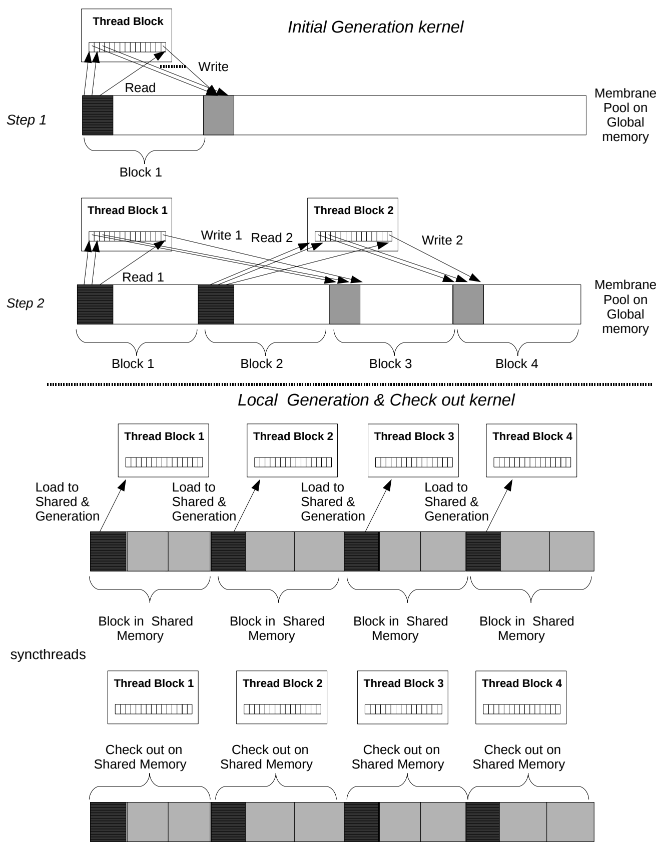

Figure 7: P system simulation on a single GPU.

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 Figure 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 at this moment for CUDA enabled devices with CUDA Compute Capabilities (C.C.C.) 1.3 [10]. Nevertheless, those guidelines are mainly focused on arithmetic intensive applications on a single GPU. 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, 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 we have exactly four CPUs in our system which can individually host each of the required processes. This way, our three implementations lack of 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

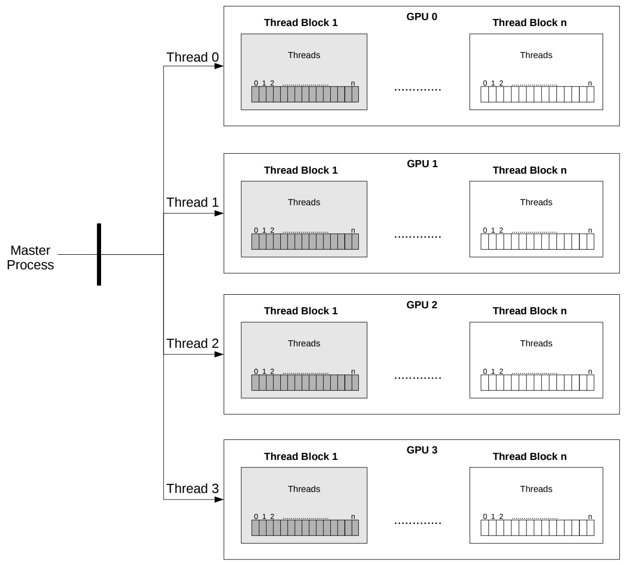

Figure 8: Data policy on a set of four GPUs.

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 [10], which makes hard to measure time in a reliable manner, particularly on multi-GPU environments.

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

At a starting point, the simulation barely exploits GPU resources because the computation begins with a single CUDA thread block (which represents the membrane generated by the *Parallel Preprocessing* step). However, the number of CUDA thread blocks grows exponentially in the *Generation* stage along with the number of membranes, 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. Hardware features are summarized in Tables 1 and 2.

The shared memory platform is a HP Integrity Superdome SX2000 endowed with 64 CPUs, Intel Itanium 2 dual-core Montvale (16 Kbytes L1, 256 Kbytes L2, 18 Mbytes L3). Total DRAM memory available is 1.5 Tbytes and interconnection network is a 4x DDR Infiniband.

The distributed memory system is a 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 Mbytes L2 cache). DRAM memory capacity for the whole system is 1072 Gbytes. Interconnection network is also a 4x DDR Infiniband.

Finally, our GPU-based platform include a four-socket,

Table 1: CUDA and hardware features for the Tesla C1060 GPU used within our GPU-based platform.

Feature	Limitation
Multiprocessors (SM)	30
Streaming processors / SM	8
Total number of streaming processors	240
32-bit registers / SM	16384
Shared memory / SM	16 KB
Threads / SM	1024
Threads / Block	512
Threads / Warp	32
Device (video) memory available	4 GB

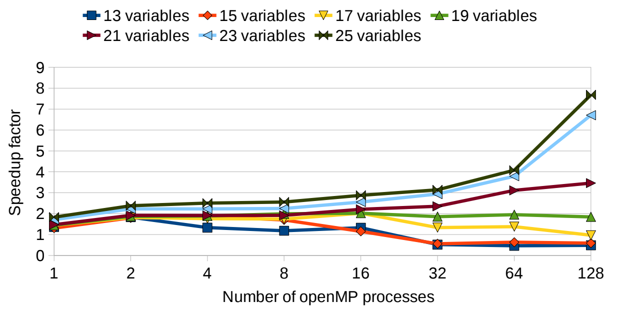

Figure 9: Speed up factor achieved by the blocking algorithm when varying the number of variables.

quad-core Intel Xeon E5530 (Nehalem with a 8 Mbytes L2 cache), which acts as a host machine for our four Nvidia Tesla C1060 GPUs 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 [14]. 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 this parameter from $n = 13$ variables (2^{13} membranes) to $n = 25$ 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 Equation 1.

$$Size = 2^n(membranes) \times l(objects) \times 4(uint)\ bytes \quad (1)$$

5.1 The shared memory platform

A performance comparison between the blocking and non-blocking algorithm for 64 membranes per block is shown in Figure 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

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

	Shared memory	Distributed memory	GPU-based
Hardware platform	Hewlett-Packard Integrity Superdome SX2000	Hewlett-Packard Blade System	4 Intel Xeon E5530 CPU (plus 4 Tesla GPUs described 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)	1536 GB	1072 GB	16 GB. (+ 16 GB. video memory)
Programming model	OpenMP (+ Linux 64 bits)	MPI (+ Linux 64 bits)	CUDA (+ Linux 64 bits)
Compiler	icc Intel 11.1	HP MPI 02.03.01	nvcc Nvidia 2.3

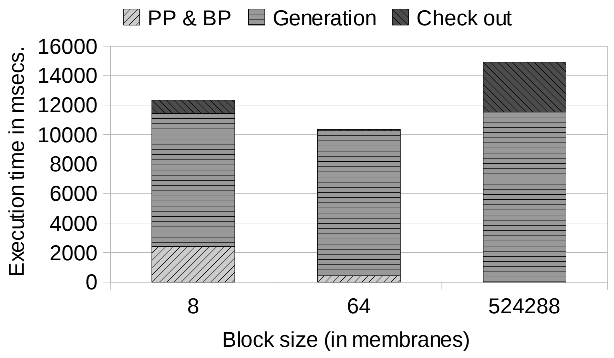

Figure 10: Breakdown for the total execution time using 8 processes for a SAT problem composed of $n = 23$ variables and $l = 200$ literals.

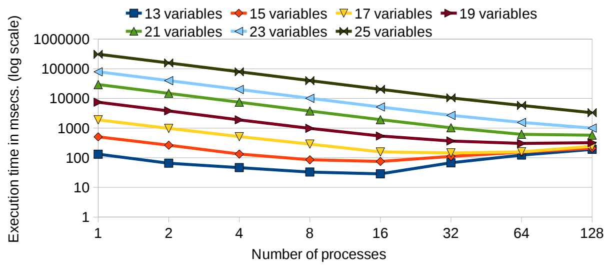

Figure 11: OpenMP code performance varying the number of variables for the block-based version.

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.

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

Finally, Figure 11 shows the execution time (in a log scale) 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 in order to study the scalability of the system.

$$t_{total} = t_{prepro} + t_{cpu} + t_{overhead} \qquad (2)$$

The total execution time is given by the equation 2. 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 needed by each process to prepare the blocking execution (that is, *Block Preprocessing*). 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.

Figure 11 shows that the scalability of the system grows with the problem size, as processing time (see equation 2) predominates over remaining parameters as long as the problem size increases. This scalability gets reduced on smaller benchmarks.

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

5.2 The distributed memory platform

In this case, the maximum speed up obtained by the best configuration for the blocking technique algorithm reaches up to 2x versus the non-blocking alternative, with this peak

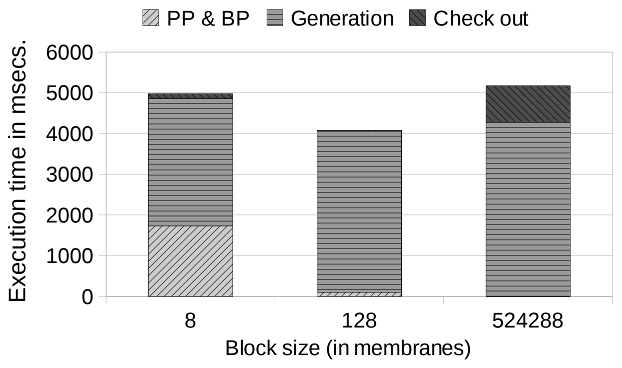

Figure 12: Breakdown of the total execution time using 8 MPI cores with $n = 23$ variables and $l = 200$ literals.

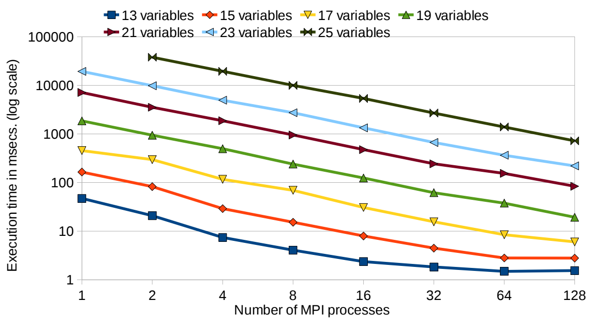

Figure 13: MPI code performance varying the number of variables.

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, Figure 12 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, Figure 12 shows only the largest, shortest, and best performance block size configurations. The optimal case here corresponds to 2^7 membranes per block (100 Kbytes per block).

Figure 13 shows the execution time (in a log scale) for the MPI code, taking the best configuration blocking technique and varying the number of variables of the SAT problem and the number of MPI processes. The total execution time can also be given by the equation 2. 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.

Figure 13 reveals that the system scalability improves again with the problem size, but it scales much better than in the OpenMP case. 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 Gbytes and the maximum memory per node is 16 Gbytes).

Note that this version does not exploit the inter-task parallelism either: Each membrane is sent to a node and simulations are executed sequentially on that node.

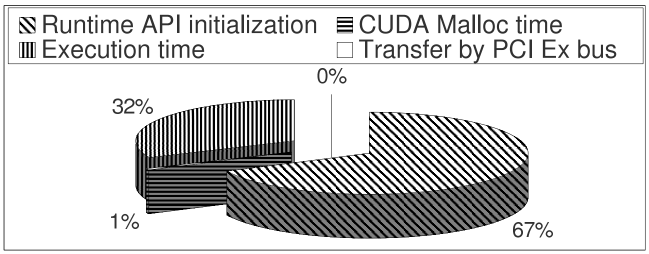

Figure 14: Breakdown of the total execution time in a single GPU with $n = 22$ variables.

5.3 The set of four GPUs

In this case, the tiling technique obtains up to 1.75x speed up factor versus the non-tiling counterpart.

Figure 14 shows the breakdown of the total execution time for a single GPU executing the benchmark with $n = 22$ variables and using a tiling version. It shows that the 67% of total execution time is spent by the runtime API initialization on average, and only 32% corresponds to the actual execution time. Data transfers are not that important here, and lose the leadership shown on previous platforms.

The runtime API initialization penalty is not usually considered when timing GPU applications because it is not stable between different executions nor related to the actual GPU computation. But in our case it represents two thirds of the total execution time, so we decided to include it within GPU times even though it goes against its performance over the other two architectures.

First of all, we evaluate the impact of the data block size. Figure 15 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 Kbytes for Tesla C1060). 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 15 reflects this fact.

Likewise, memory accesses in the *Generation* and *Check-Out* 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 from allocating more than one block per SM. As a result, the overall improvement is just 14% over the four membranes per block configuration.

Figure 16 shows the performance for the tiling version of the GPU simulator with eight membranes per block, and varying the problem size. The number of GPUs is also increased to study the scalability for the system.

In a multi GPU environment, Figure 16 shows a linear speed up along with the number of GPUs. This is expected as the computational workload is evenly distributed on GPUs. Furthermore, there is more room on each GPU memory space, so higher workloads may be executed. Figure 16 shows that a single GPU cannot execute the benchmark with n=23 variables, whose memory requirements are 6400

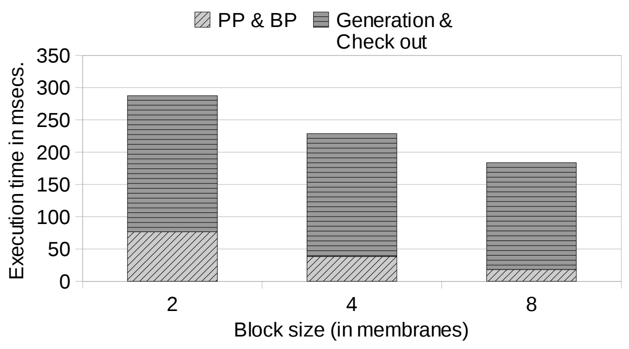

Figure 15: Breakdown for the execution time on a single GPU with $n = 23$ variables and $l = 200$ literals.

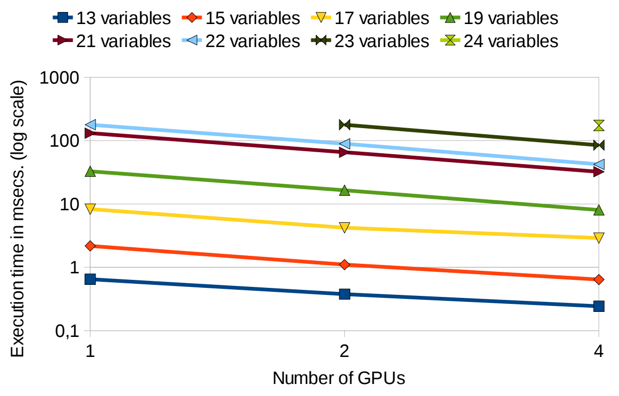

Figure 16: CUDA performance when varying the number of variables (on y axis) and GPUs (x axis).

Mbytes. Similarly, two GPUs cannot execute the benchmark with n=24, with its size reaching 12800 Mbytes. At this point, we recall that a P systems simulation creates an exponential workspace to obtain polynomial time solutions for NP-complete problems. So, the benchmark composed of n=25 variables consumes 25.6 Gbytes, which again becomes unfeasible on four GPUs.

Times in Figure 16 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. *Parallel Preprocessing* time spent to arrange the execution on multiple GPUs is also ignored, though this time is negligible as the simulation creates just four membranes on a four GPUs configuration.

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 on 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.

5.4 Overall comparison

Figure 17 summarizes the performance for all our implementations. For the smallest benchmark, GPU performance gets severely affected by initialization overheads, but this is quickly amortized as we increase the problem size. The situation reverses for larger benchmarks, reaching its peak for $n = 23$ variables, where the problem size only fits into two or four GPUs, and that is the reason why the time on a single GPU is missing. With the last run for $n = 25$ variables requiring 25.6 Gbytes, we were unable to execute it on GPUs even considering together the video memory of our four GPUs.

Considering the largest problem size and amount of parallelism we were able to expose on the three parallel platforms for a fair comparison ($n = 23$ variables and four processors), execution times were 20049.70 msec. using OpenMP on the shared memory multiprocessor, 4954.03 msecs. using MPI on the distributed memory multiprocessor and 565.56 msecs. using CUDA in our set of four GPUs. Consequently, the speed-up we attain with our set of GPUs reaches 8.75x versus the distributed memory system and 35.44x versus the shared memory platform for a much cheaper high-performance alternative.

6. 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 base on shared memory, distributed memory and a set of GPUs. We have also used three different programming models: OpenMP, MPI and CUDA, respectively.

Our data placement analysis reveals that blocking increases the bandwidth in all targeted systems by taking advantage of data locality, but performance varies depending of the memory architecture and the way to manage it. We also dedicate some efforts to reduce the cost of preprocessing steps required for applying this technique on each platform.

The blocking technique improves the parallel efficiency of the shared memory architecture, 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.

The distributed memory system exhibits good scalability with the number of processors, which can be partially explained by the low number of communications required by our simulations.

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

For the future, the newest generation of many-core GPU architectures, Nvidia Fermi, enhances the GPU with memory resources to develop general purpose applications and more sophisticated models of P systems. Moreover, the combination of cloud computing and heterogeneous systems can be an alternative for increasing the memory size without sacrificing performance at all.

Alternative models of P systems which could be used to computationally replicate biological systems within the framework of population and systems biology (i.e., probabilistic/stochastic models) are well positioned to be successfully simulated on multi- and many-core systems due to its arithmetic intensity and large number of iterations required to adjust the model. A high-performance implementation of those simulation models looks promising on GPUs and we have provided some guidelines to succeed by using CUDA.

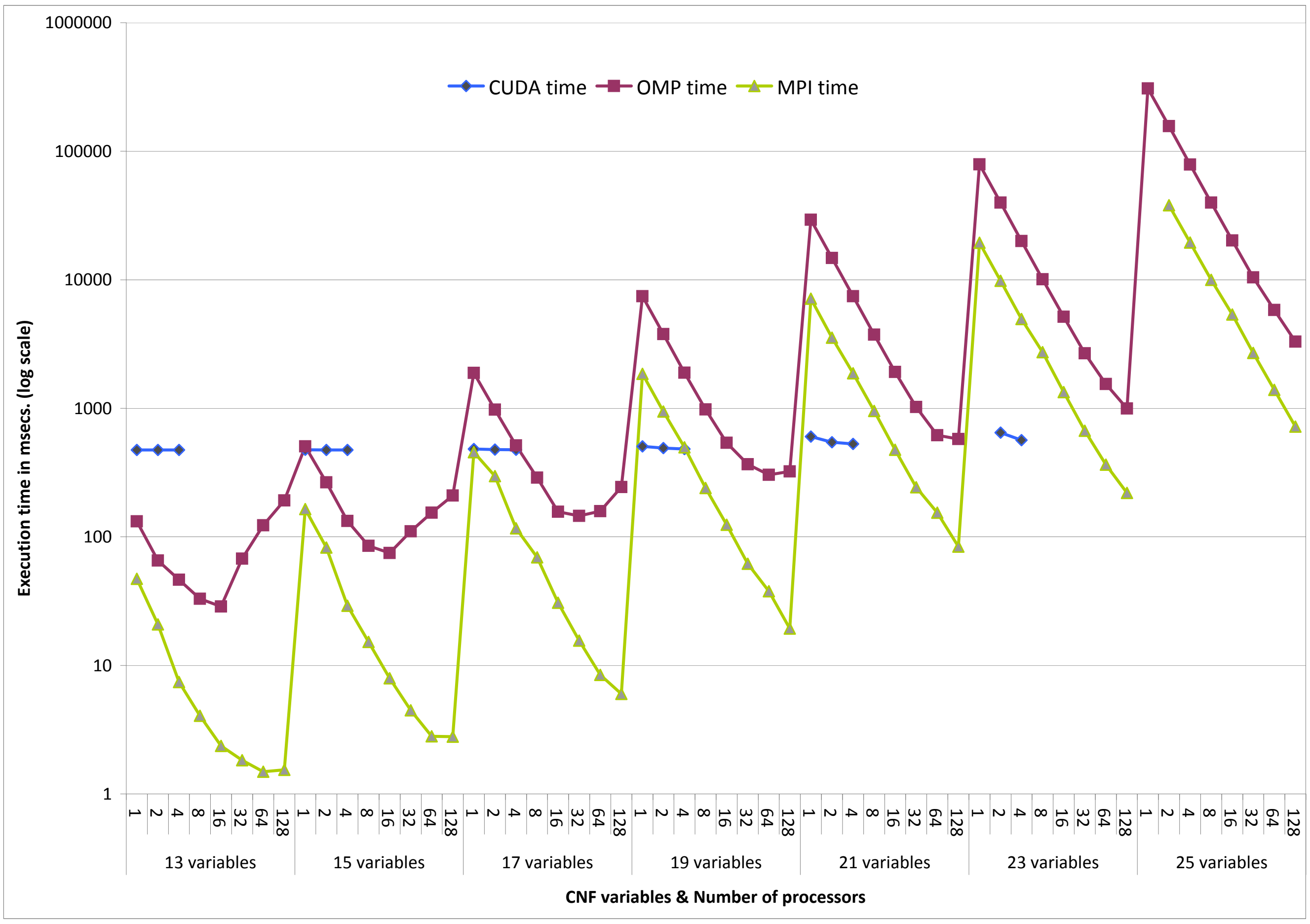

Figure 17: Execution time for the three different programming models and architectures: CUDA on GPUs, OpenMP on a shared memory system and MPI on a distributed memory platform.

7. REFERENCES

[1] Message Passing Interface (MPI). http://www.mcs.anl.gov/mpi.

[2] The OpenMP Specification. http://www.openmp.org.

[3] S. Alonso, L. Fernández, F. Arroyo and J. Gil. A circuit implementing massive parallelism in transition P systems. *International Journal Information Technologies and Knowledge*, 2(1):35–42, 2008.

[4] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. M. del Amor, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79(6):317–325, 2010.

[5] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. M. del Amor, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[7] D. Díaz, C. Graciani, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado and M. J. Pérez-Jiménez. Software for p systems. In *Gh.Paun, G.Rozenberg, A.Salomaa, editors*, The Oxford Handbook of Membrane Computing, pages 437–454. Oxford Univ. Press, 2009.

[8] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez and A. Riscos-Núñez. An overview of p-lingua 2.0. *Lecture Notes in Computer Science*, 5957:264–288, 2010.

[9] V. Nguyen, D. Kearney and G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. *J. Logic and Algebraic Programming*, 79(6):383–396, 2010.

[10] NVIDIA. *CUDA Programming Guide 2.0.* 2008.

[11] G. Paun. Membrane computing. An introduction. *Springer-Verlag*, pages 9–419, 2002.

[12] G. Paun, T. Centre and C. Science. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[13] M. J. Pérez-Jiménez, Á. Romero-Jiménez and F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *J. Natural Computing*, 2(3):265–285, 2003.

[14] M. Qasem. WinSAT website: (http://users.ecs.soton.ac.uk/mqq06r/winsat).