Computing and Informatics, Vol. 27, 2008, 551–569

AN IMPLEMENTATION OF MEMBRANE COMPUTING USING RECONFIGURABLE HARDWARE

Van Nguyen, David Kearney, Gianpaolo Gioiosa

School of Computer and Information Science
University of South Australia
Mawson Lakes Boulevard, Mawson Lakes, South Australia 5095
e-mail: {Van.Nguyen, David.Kearney, Gianpaolo.Gioiosa}@unisa.edu.au

Revised manuscript received 3 December 2007

Abstract. Because of their inherent large-scale parallelism, membrane computing models can be fully exploited only through the use of a parallel computing platform. We have fully implemented such a computing platform based on reconfigurable hardware that is intended to support the efficient execution of membrane computing models. This computing platform is the first of its type to implement parallelism at both the system and region levels. In this paper, we describe how our computing platform implements the core features of membrane computing models in hardware, and present a theoretical performance analysis of the algorithm it executes in hardware. The performance analysis suggests that the computing platform can significantly outperform sequential implementations of membrane computing as well as Petreska and Teuscher's hardware implementation, the only other complete hardware implementation of membrane computing in existence.

Keywords: Membrane computing, parallel implementations of membrane computing, hardware implementations of membrane computing, reconfigurable hardware

1 INTRODUCTION

Membrane computing [6] investigates models of computation inspired by structural and functional properties of biological cells. Because of their inherent large-scale parallelism, membrane computing models can be fully exploited only through the use of a parallel computing platform. We have fully implemented such a computing platform based on reconfigurable hardware that is intended to support the efficient

execution of membrane computing models. This computing platform is the first of its type to implement parallelism at both the system and region levels. In this paper, we describe how our computing platform implements the core features of membrane computing models in hardware, and present a theoretical performance analysis of the algorithm it executes in hardware. In a companion paper [5], we show how the software elements of the computing platform tailor the hardware elements according to the specific properties of the P system to be executed, and present an empirical analysis which demonstrates that the computing platform achieves very good performance while making economical use of hardware resources.

2 BACKGROUND

2.1 Membrane Computing Models

Membrane computing models are models of computation inspired by structural and functional properties of biological cells, especially properties that arise because of the presence and activity of biological membranes. We call membrane computing models *P system models* and their instances *P systems*.

So far our research has focused on one P system model. This model, which we call the *core P system model*, includes all the fundamental features of a P system model plus two common additional features (catalysts and reaction rule priorities). A P system that instantiates the core P system model is defined as a construct

$$\Pi = (V, T, C, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m)),$$

where

- V is an alphabet that contains labels for all the *types of objects* in the system;
- $T \subseteq V$ is the *output alphabet*, which contains labels for all the types of objects that are relevant to the determination of the system output;
- $C \subseteq V - T$ is the alphabet that contains labels for all the *types of catalysts*, which are the types of objects whose multiplicities cannot change through the application of a reaction rule;
- μ is a hierarchical membrane structure consisting of m membranes, with the membranes (and hence the regions defined by the membranes) injectively labelled by the elements of a given set H of m labels (in this paper, $H = \{1, 2, \ldots, m\}$);
- each $w_i, 1 \leq i \leq m$, is a string over V that represents the *multiset of objects* contained in region i of μ in the initial configuration of the system;
- each $R_i, 1 \leq i \leq m$, is a finite *set of reaction rules* over V associated with the region i of μ;
- a *reaction rule* is a pair (r, p), written in the form $r \rightarrow p$, where r is a string over V representing a multiset of reactant objects and p is a string over $\{a_{\text{here}}, a_{\text{out}},$

$a_{\text{in}} \mid a \in V\}$ representing a multiset of product objects, each of which either (a) stays in the region to which the rule is associated (the subscript 'here' is usually omitted), (b) travels 'out' into the region that immediately contains the region to which the rule is associated, or (c) travels 'in' to one of the regions that is immediately contained by the region to which the rule is associated; and

- each ρ_i is a partial-order relation over R_i which defines the *relative priorities of the reaction rules* in R_i.

Several P system models have been developed that extend in various ways the core P system model. Examples of additional features found in these extended models include: structured (i.e., non-atomic) objects, membrane creation and dissolution, special inter-region communication rules (e.g., symport and antiport rules), membrane permeability, and electronic charge for objects and membranes [6].

2.2 Existing Implementations of Membrane Computing

Because of their inherent parallelism, it is not possible to truly implement P systems on sequential computing platforms. Nevertheless, software packages exist which enable to simulate in a sequential manner the execution of P systems (e.g., see [4]).

To fully benefit from the membrane computing paradigm, developers of P system models require a computing platform that is able to exploit the large-scale parallelism of the models. Such a computing platform might include multiple software-programmed microprocessors. For example, Ciobanu and Guo [1] have developed a software-based parallel implementation of a P system model that is designed to be executed on a cluster of computers. Alternatively, a hardware-based computing platform might be used. For example, Petreska and Teuscher [7] have developed a full implementation on reconfigurable hardware of a particular class of P systems, while other researchers have designed digital circuits for particular aspects of P systems (e.g., see [2] and [3]).

Petreska and Teuscher's hardware implementation has demonstrated the feasibility of implementing some of the fundamental features of P systems in hardware. Nevertheless, it has a serious limitation: it does not implement parallelism at the region level (i.e., the reaction rules in a region are applied sequentially). This is a major limitation for two reasons. First, having two levels of parallelism (at both the system and region levels) is a key feature of the membrane computing paradigm. Second, without implementing both levels of parallelism, it is not possible to exploit the performance advantages of the membrane computing paradigm. Indeed, as the ratio of reaction rules to regions increases, the performance of Petreska and Teuscher's implementation tends towards that of a sequential implementation. Achieving parallelism at the region level requires the implementation of a scheme for the resolution of hardware resource conflicts that arise because different reaction rules may consume or produce the same types of objects in the same region at the same time. It is difficult to efficiently implement such a scheme, and this is probably why Petreska and Teuscher did not attempt to do so.

2.3 Reconfigurable Hardware

Because of the performance compromise associated with the use of software-programmed microprocessors, alternative computing methods based on the direct use of hardware are often required. One such method is to use an application-specific integrated circuit (ASIC). ASICs are specially designed for a specific application. Therefore, they can achieve a higher performance than software-programmed microprocessors when executing the algorithm for which they were designed. However, with this higher performance comes a reduction in the flexibility of the computing method – as the implemented algorithm is fabricated on a silicon chip, it cannot be altered without creating another chip. Another method of overcoming the performance compromise associated with the use of software-programmed microprocessors is to use reconfigurable hardware. Unlike ASICs, reconfigurable hardware can be modified. Therefore, by using reconfigurable hardware, it is possible to improve the performance of the software-based method while retaining much of its flexibility. A field-programmable gate array (FPGA) is a type of reconfigurable hardware device. An FPGA consists of a matrix of logic blocks which are connected by means of a network of wires. The logic blocks at the periphery of the device can also perform I/O operations. The functionality of the logic blocks can be modified by loading configuration data from a host computer (a standard PC that is connected to the FPGA by a PCI bus). In this way, any custom digital circuit can be mapped onto the FPGA, thereby enabling the FPGA to execute a variety of applications.

3 DESCRIPTION OF THE HARDWARE IMPLEMENTATION

The research described in this paper contributes the first computing platform based on reconfigurable hardware to implement parallelism at both the system and region levels. We call the computing platform Reconfig-P. To the best of our knowledge, other than Petreska and Teuscher's implementation, Reconfig-P is the only complete hardware-based computing platform for membrane computing applications in existence.

Reconfig-P is able to execute P systems that instantiate the core P system model. It consists of a source code generator (written in Java) and an FPGA. The source code generator analyses the specification of the input P system, and then generates Handel-C source code that implements a customised hardware representation for the P system. Handel-C is a high-level hardware specification language. Having a syntax similar to that of the C programming language, Handel-C allows hardware representations to be specified at a very abstract level (without the structure of the hardware circuit being described in any way), and therefore eases the process of creating customised hardware representations.

In the following section, we describe how Reconfig-P implements the core features of P system models in hardware. In [5], we describe how the source code generator produces Handel-C source code for these features given the specific properties of the input P system.

3.1 Hardware Implementation of Core P System Features

P systems can differ significantly with respect to size, structure and information content. Reconfig-P takes advantage of this fact by configuring the FPGA according to the specific requirements of the P system to be executed.

Although P systems can differ significantly, there are certain core features common to all P systems. These include (a) regions and their containment relationships, (b) mutisets of objects, (c) application of reaction rules, and (d) synchronisation of the application of reaction rules. This section describes how these core features are implemented in hardware in Reconfig-P.

3.1.1 Regions and Their Containment Relationships

As the evolution of a P system is essentially a matter of the modification of the contents of regions according to certain rules, regions do not need to be explicitly represented in hardware. Instead, a region is represented in hardware implicitly via its contents. The only inter-region containment relationships that it is important to represent are those between regions between which it is possible for objects to traverse through the application of a reaction rule. These containment relationships are represented implicitly by ensuring that each reaction rule with an 'in' or 'out' target directive has, for each region to/from which it sends/receives objects, access to the multiset of objects in that region.

3.1.2 Multisets of Objects

Because the multiplicity values of objects in a region can be accessed by multiple reaction rules simultaneously, the hardware elements that store them should support concurrent accesses. Therefore a multiset is implemented as an array of registers (see Figure 1). Because it is infeasible to predict which types of objects may become available in which regions during the evolution of a P system, the array of registers that represents the multiset of objects in a region contains one register for every type of object in the alphabet of the P system. A common bitwidth is used for all object types (the default width is 8 bits).

Using registers can be expensive if a large amount of data needs to be stored. However, because in the hardware design each register corresponds to the multiplicity of a *type* of object in a region (rather than an individual object), for most P systems only a relatively small amount of data needs to be stored.

3.1.3 Reaction Rules

A reaction rule is implemented as a processing unit. This processing unit is represented in Handel-C as a potentially infinite while loop that contains code that specifies the processing associated with the application of the reaction rule. If a reaction rule operates on the multiplicity value for a particular object type in a particular region, then the section of the code for its corresponding processing unit that

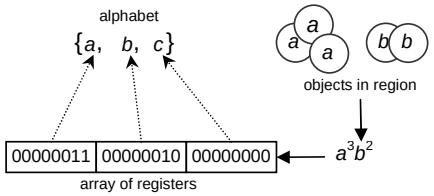

Fig. 1. A multiset of objects is implemented as an array of registers

accomplishes this operation contains a reference to the array element representing that multiplicity value.

In a transition of a P system, all the reaction rules in the system complete one instance of execution, which consists of two phases. In the first phase, called the *preparation phase*, objects are assigned to the reaction rules that require them as reactants or catalysts. That is, for each reaction rule in each region, the number of instances of the reaction rule that can be applied is determined. In the second phase, called the *updating phase*, each applicable reaction rule updates one or more multisets of objects according to its definition and the number of instances of the reaction rule that can be applied.

The rest of this section describes the processing performed by the processing units for reaction rules during the preparation and updating phases.

Preparation Phase. In the preparation phase, each reaction rule attempts to obtain as many of each of its required types of object as possible so as to maximise the number of instances of the reaction rule that can be applied in the updating phase. Therefore, implementing the preparation phase involves calculating for each reaction rule r the value max-instances$_r$, which is the maximum number of instances of r that can be applied in the current transition of the P system given (a) the current state of the multiset of objects in its region and (b) the relative priorities and requirements of the other reaction rules in its region. The processing unit corresponding to r performs the calculation.

To calculate max-instances$_r$, the processing unit for a reaction rule r first calculates for each of its required object types (using integer division) the ratio of the number of available objects of that type in the region of r to the number of objects of that type needed to apply one instance of r. This is done in one clock cycle. It then calculates max-instances$_r$, which is equal to the minimum ratio calculated in the previous step. The operation of determining the minimum ratio can be represented as a binary tree in which each node corresponds to the execution of a binary MIN operation and executing the MIN operation at the root node gives the value of the minimum ratio. This tree has $\log_2 n$ levels, where n is the sum of the number of reactants and the number of catalysts in the definition of r. The processing unit for r evaluates max-instances$_r$ by first executing in parallel all the MIN operations at the bottom (leaf) level of the tree, then executing in parallel all the MIN operations at

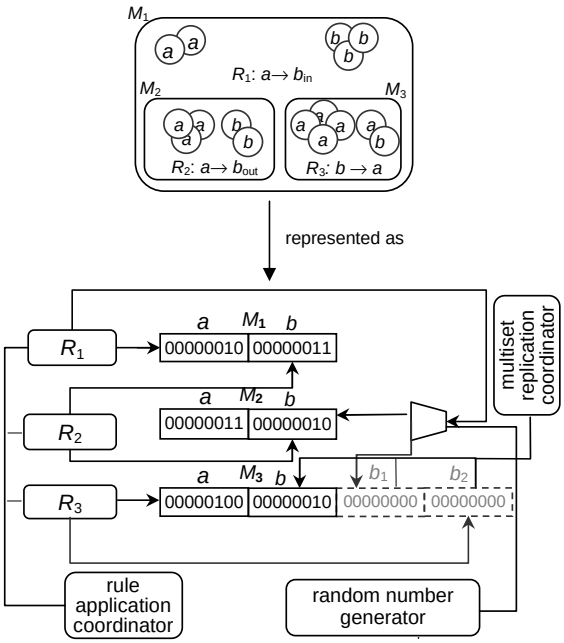

Fig. 2. An illustration of how the structural aspects of a P system are represented as high-level hardware components. (The multiset replication coordinator and extra array elements for object type b in Region 3 are included only if the space-oriented conflict resolution strategy is used.)

the next level up, and so on, until finally it executes the MIN operation at the root node to obtain the value of max-instances$_r$. Therefore, calculating max-instances$_r$ takes $\log_2 n$ clock cycles.

If two reaction rules attempt to obtain objects of the same type, then their corresponding processing units execute the relevant operation one after the other according to their relative priorities. (It is assumed that reaction rules that attempt to obtain objects of the same type have been assigned relative priorities.) Otherwise, the processing units for different reaction rules execute in parallel. Therefore the number of clock cycles taken to complete the preparation phase for the entire P system is the maximum number of clock cycles taken by an individual rule processing unit, out of all the rule processing units in the P system, to complete its own preparation phase.

Updating Phase. At the start of the updating phase, the processing unit for a reaction rule r inspects the value of max-instances$_r$ to determine whether r is applicable in the current transition. If max-instances$_r = 0$, r is inapplicable; otherwise r is applicable. As it takes zero clock cycles to evaluate a conditional expression in

Handel-C, determining the applicability of r takes zero clock cycles. The applicability status of r is recorded in the `isApplicableFlag` of r (see Figure 3). Once the applicability status of each reaction rule has been determined and recorded, the processing unit that coordinates the execution of reaction rules (see Section 3.1.4) is able to determine whether the P system should halt or continue the updating phase. Assume that the P system should continue the updating phase. If r is inapplicable, the processing unit for r simply waits for the next transition. If r is applicable, it moves on to the next step of its updating phase.

In the next step of the updating phase, every instance of every applicable reaction rule is applied. This is implemented by having the processing unit for each applicable reaction rule r bring about the combined effect of the execution of the instances of r. That is, the processing unit decreases/increases certain multiplicity values in certain multiset arrays according to the type, amount and source/destination of the objects consumed/produced by the instances of the reaction rule. For example, in Figure 2, in the next transition of the P system represented at the top of the figure, the processing unit R_2 would decrease by 2 the value stored in the register corresponding to object type a in the multiset array for Region 2, and increase by 2 the value stored in the register corresponding to object type b in the multiset array for Region 1.

If a reaction rule includes 'in' target directives, the definition of a P system calls for nondeterministic targeting of objects if there are multiple child regions. Such nondeterministic targeting can be approximated through the use of pseudorandom numbers. Therefore, the hardware design associates a *random number generator* to each processing unit for a reaction rule that might produce objects in multiple child regions of its own region. When such a processing unit needs to select a destination child region, it invokes its random number generator to obtain a number which identifies the child region to be selected. For example, the processing unit R_1 in Figure 2 invokes its random number generator to determine whether to produce b objects in Region 2 or in Region 3.

Processing units for reaction rules that do not manipulate any multiplicity values in common execute in parallel during the updating phase. This is not necessarily the case for processing units for reaction rules that do manipulate at least one multiplicity value in common, since without further measures being taken, the parallel execution of such processing units would lead to situations where multiple processing units write to the same register at the same time. Section 3.2 describes two alternative conflict resolution strategies that Reconfig-P makes available for the prevention of such situations, and shows the extent to which each strategy allows conflicting rule processing units to execute in parallel during the updating phase. The number of clock cycles taken to complete the updating phase depends on the conflict resolution strategy that is adopted.

3.1.4 Synchronisation of Reaction Rules

Figure 3 illustrates the synchronisation of rule processing units involved in the execution of a transition of a P system.

The synchronisation of rule processing units is controlled by three sentinels – `preparationSentinel`, `applicableSentinel` and `updatingSentinel` – and corresponding flags associated with each rule processing unit – `preparationCompleteFlag`, `updatingCompleteFlag` and `isApplicableFlag`. The sentinels and flags are implemented as 1-bit registers. The flags of a given type are stored in a single array.

The flags `preparationCompleteFlag`, `isApplicableFlag` and `updatingCompleteFlag` for a rule processing units are used to indicate whether the rule processing unit has completed its preparation phase, is applicable, and has completed its updating phase, respectively. The value of each sentinel is the result of performing the AND or OR function to the values of all its corresponding flags. The value of `preparationSentinel` indicates whether all rule processing units in the P system have completed their respective preparation phases. The value of `applicableSentinel` indicates whether at least one rule processing unit is applicable (i.e., whether the P system should continue execution). And the value of `updatingSentinel` indicates whether all applicable rule processing units in the P system have completed their respective updating phases (and hence whether the P system is ready to proceed to the next transition).

The management of synchronisation is the responsibility of the *rule application coordinator*, a processing unit that executes in parallel with the rule processing units (see Figure 2). The rule application coordinator monitors the conditions relevant to synchronisation at each clock cycle.

For P systems with a large number of reaction rules it might be advantageous to decompose each assignment statement that implements the updating of a sentinel value into multiple assignment statements of reduced logical depth. Therefore Reconfig-P incorporates a *logic depth reduction* feature. It decomposes an assignment statement with n operands into multiple assignment statements, each of which has at most $x \leq n$ operands. If as many of these assignment statements as possible contain x operands, then the original assignment statement is replaced by $\lceil \log_x n \rceil$ assignment statements. The user sets the value of x in order to obtain the best results.

3.2 Conflict Resolution in the Updating Phase

As mentioned in Section 3.1.3, a conflict occurs in the updating phase when multiple rule processing units write to the same register at the same time. This occurs if the rule processing units consume or produce the same type of object in the same region in the same transition. As mentioned in Section 2.2, Petreska and Teuscher's hardware implementation avoids the conflict problem by totally sacrificing the parallelism that gives rise to the problem. This is an undesirable strategy, because it hinders performance significantly.

Reconfig-P implements two alternative conflict resolution strategies: the *time-oriented strategy* and the *space-oriented strategy*. The time-oriented strategy consumes time, whereas the space-oriented strategy consumes space. Therefore, the

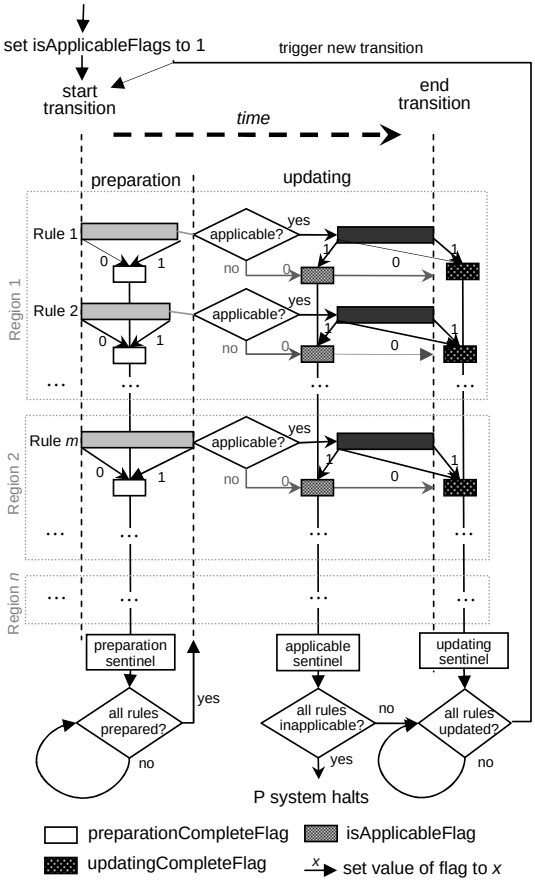

Fig. 3. An illustration of the synchronisation performed by Reconfig-P to accomplish a transition of a P system

best strategy to use depends on whether it is more important to optimise space or time usage. The user selects the strategy to be used.

Both strategies involve determining in software before run-time all of the potential conflicts that might occur between reaction rules, and then generating the hardware circuit for the P system in such a way that all rule processing units can execute independently without any possibility of writing to the same register at the same time. The task of determining the resource conflicts has a time complexity of $\Theta(n_r n_o)$, where n_r is the number of reaction rules in the P system, and n_o is the number of object types in the alphabet of the P system. Therefore it has a negligible impact on performance. Note that, since the circuit for the P system need be generated only once, the task is performed only once.

In both strategies, potential conflicts are determined through the construction of a *conflict matrix*. Each row of a conflict matrix for a P system is a quadruple (p, q, r, s), where p is an object type in the alphabet of the P system, q is a region in the P system, r is the set of reaction rules whose application results in the consumption and/or production of objects of type p in q, and s – called the *conflict degree* of (p, q) – is the size of r. There is a row for every pair (p, q).

We now describe how the updating phase occurs when (a) the time-oriented strategy is used, and (b) the space-oriented strategy is used.

3.2.1 Time-Oriented Conflict Resolution

In the time-oriented conflict resolution strategy, if two rule processing units need to update the multiplicity value for the same type of object in the same region, then they do so one after the other (the order in which they do so is not important and so is chosen arbitrarily).

Table 1 illustrates the time-oriented strategy. In the table, 'u(p, q)' denotes the operation of updating the multiplicity value of object type p in region q.

The correct interleaving of the various conflicting operations of the rule processing units is determined by means of an analysis of the conflict matrix for the P system before run-time. That is, the Handel-C source code that is generated for the P system specifies the interleaving directly. This is achieved by inserting the appropriate number of single-clock-cycle `delay` statements in the appropriate places in the source code for the rule processing units. For example, the code in the processing unit for r_1^3 that updates the multiplicity value of object type a in Region 2 is preceded by two `delay` statements, whereas the corresponding code for object type b in Region 3 is not preceded by any `delay` statements. For the general case, take a quadruple (p, q, r, s) from the conflict matrix for a P system. Assume that the reaction rules $r_1, r_2, \ldots, r_n \in r$ are ordered (for the purpose of conflict resolution) according to the natural ordering of their subscripts. Then the number of `delay` statements to be inserted immediately before the code in the processing unit for the reaction rule $r_i \in r$ that updates the multiplicity value of object type p in region q is equal to $i - 1$.

As Table 1 illustrates, the number of clock cycles taken to update the multiplicity value for object type p in region q of a P system is equivalent to the conflict degree of (p, q), which is recorded in the conflict matrix for the P system.

Let k be the highest conflict degree in the conflict matrix for a P system. Then the updating phase for the P system takes k clock cycles to complete when the time-oriented conflict resolution strategy is used.

3.2.2 Space-Oriented Conflict Resolution

In the space-oriented conflict resolution strategy, if n reaction rules need to update the multiplicity value for the same type of object in the same region, then n copies are made of the register that stores that multiplicity value. The processing units for

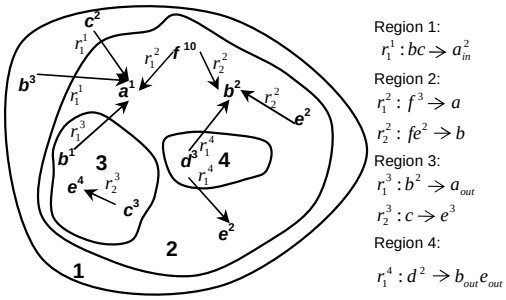

Fig. 4. An example P system configuration. An arrow labelled by reaction rule r from object type p_1 in region q_1 to object type p_2 in region q_2 means that p_1 is a reactant of r (taken from q_1) and p_2 is a product of r (produced in q_2)

Clock cycle	r_1^1	r_1^2	r_2^2	r_1^3	r_2^3	r_1^4
1	u$(a, 2)$, u$(b, 1)$, u$(c, 1)$ end	u$(f, 2)$	u$(b, 2)$, u$(e, 2)$	u$(b, 3)$	u$(c, 3)$, u$(e, 3)$ end	u$(d, 4)$, u$(e, 2)$
2		u$(a, 2)$ end	u$(f, 2)$ end			u$(b, 2)$ end
3				$u(a, 2)$ end		

Table 1. How the processing units for the reaction rules in the P system in Figure 4 execute during the updating phase of the current transition if the time-oriented conflict resolution strategy is used

the conflicting reaction rules are assigned one copy register each, and in the updating phase write to their respective copy registers (see Figure 2 for an example). Once all of the rule processing units have completed writing to their registers, processing units called *multiset replication coordinators* (each of which is associated with one object type in one region and runs in parallel with the other processing units in Reconfig-P) read the values that have been stored in the copy registers, and set the original registers in the relevant multiset arrays accordingly (again see Figure 2). This step takes one clock cycle to complete. However, for P systems with a large number of object copies, it may be beneficial to perform logic depth reduction (see Section 3.1.4). Table 2 illustrates the space-oriented strategy.

4 THEORETICAL PERFORMANCE ANALYSIS

In this section, we analyse the time complexity of the parallel algorithm executed in hardware by Reconfig-P (in both the time-oriented and space-oriented modes), and

Clock cycle	r_1^1	r_1^2	r_2^2	r_1^3	r_2^3	r_1^4
1	u$(a,2)$, u$(b,1)$, u$(c,1)$ end	u$(f,2)$, u$(a,2)$ end	u$(b,2)$, u$(e,2)$	u$(f,2)$ end	u$(b,3)$, u$(e,3)$ end	u$(d,4)$, u$(e,2)$, u$(b,2)$ end
2	Multiset replication coordinators update original registers in relevant multiset arrays					

Table 2. How the processing units for the reaction rules in the P system in Figure 4 execute during the updating phase of the current transition if the space-oriented conflict resolution strategy is used

compare its time complexity with the time complexity of the sequential algorithm used in sequential implementations of membrane computing. The results of the analysis demonstrate the performance advantages of the parallelism implemented by Reconfig-P.

In the analysis, operations that have a negligible time complexity are not considered. More specifically, the simplifying assumption is made that the total execution time for an algorithm is the sum of the execution times for the preparation phase and the updating phase and any associated synchronisation operations. All execution times are measured in clock cycles.

4.1 Definitions

In our analysis, $M = \{m_1, m_2, \ldots, m_n\}$ denotes the set of membranes in the P system. $V = \{o_1, o_2, \ldots, o_v\}$ denotes the alphabet of the P system. $R_{m_x} = \{r_{1,m_x}, r_{2,m_x}, \ldots, r_{k_{m_x},m_x}\}$ denotes the set of reaction rules in the region defined by membrane m_x. r_{y,m_x} denotes the y^{th} reaction rule in the region defined by membrane m_x. The superscript 'a' in r^a_{y,m_x} indicates that r_{y,m_x} is applicable. $n^{\text{R}}(r_{y,m_x}), n^{\text{C}}(r_{y,m_x})$ and $n^{\text{P}}(r_{y,m_x})$ denote the number of reactant, catalyst and product object types in reaction rule r_{y,m_x}, respectively. $M^{\text{UPDATE}}_{m_x} : V \longrightarrow R_{m_x}$ denotes the function that maps each object type in the region defined by membrane m_x to the set of reaction rules that might update its multiplicity. $Max^n_{i=1}x_i$ denotes the function that returns the maximum value in the set $\{x_1, x_2, \ldots, x_n\}$. Finally, e denotes the time taken to execute one transition of a P system. This time is composed of the separate times taken to execute the preparation phase (p), the updating phase (u) and (in the parallel algorithm) synchronisation operations (s). Synchronisation operations include updates of the sentinels `preparationSentinel` and `updatingSentinel`, as well as operations related to the coordination of multiset replication.

4.2 Analysis of the Sequential Algorithm

In the sequential algorithm used in sequential implementations of membrane computing, all reaction rules in all regions are executed one after the other. Let $p^{\text{SEQ}}(r)$ be the number of clock cycles for a reaction rule r to execute its preparation phase and $u^{\text{SEQ}}(r)$ the number of clock cycles for it to execute its updating phase. Then the number of clock cycles e^{SEQ} taken to execute one transition of the P system is given by

$$e^{\text{SEQ}} = \sum_{i=1}^{n} \sum_{j=1}^{k_{m_i}} = \begin{cases} p^{\text{SEQ}}(r_{j,m_i}), & \text{if } r_{j,m_i} \text{ is not applicable,} \\ p^{\text{SEQ}}(r_{j,m_i}) + u^{\text{SEQ}}(r_{j,m_i}), & \text{if } r_{j,m_i} \text{ is applicable.} \end{cases}$$

The preparation phase for a reaction rule involves a series of calculations of the minimum of a pair of ratios of multiplicity values for reactants and catalysts (see Section 3.1.3). The updating phase for a reaction rule involves updating the multiplicity value for each of the reactant and product object types in the definition of the rule (again see Section 3.1.3). To facilitate a comparison with the parallel algorithm executed by Reconfig-P, we make two assumptions regarding the sequential algorithm. First, we assume that it takes one clock cycle to calculate the minimum of a pair of ratios of multiplicity values. Second, we assume that it takes one clock cycle to update the multiplicity value of an object type. Therefore we have

$$p^{\text{SEQ}}(r_{y,m_x}) = n^{\text{R}}(r_{y,m_x}) + n^{\text{C}}(r_{y,m_x}) - 1$$

and

$$u^{\text{SEQ}}(r^a_{y,m_x}) = n^{\text{R}}(r^a_{y,m_x}) + n^{\text{P}}(r^a_{y,m_x}).$$

4.3 Analysis of the Parallel Algorithm

The number of clock cycles e^{PAR} taken to execute a transition of the P system when the parallel algorithm executed by Reconfig-P is used is the sum of (a) the number of clock cycles taken by the longest preparation phase execution in the whole P system, (b) the number of clock cycles taken by the longest updating phase execution in the whole P system, and (c) the number of clock cycles taken by synchronisation operations:

$$e^{\text{PAR}} = Max_{i=1}^{n} Max_{j=1}^{k_{m_i}} p^{\text{PAR}}(r_{j,m_i}) + Max_{i=1}^{n} Max_{j=1}^{k_{m_i}} u^{\text{PAR}}(r^a_{j,m_i}) + s^{\text{PAR}}.$$

Preparation Phase. As stated in Section 3.1.3, the preparation phase for a reaction rule involves $\log_2 n$ steps of applying in parallel distinct MIN calculations, where n is the number of reactant/catalyst object types in the rule. If a reaction rule conflicts with other reaction rules, it has to wait for all conflicting reaction rules with higher priority to complete execution of their respective preparation phases before

it can begin executing its preparation phase. Therefore, the number of clock cycles taken by a reaction rule to complete its preparation phase is given by

$$p^{\text{PAR}}(r_{y,m_x}) = \begin{cases} \sum_{s=1}^{y} \log_2(n^{\text{R}}(r_{s,m_x}) + n^{\text{C}}(r_{s,m_x})), \\ \qquad \text{if } r_{s,m_x} \text{ has an assigned priority,} \\ \log_2(n^{\text{R}}(r_{y,m_x}) + n^{\text{C}}(r_{y,m_x})), \\ \qquad \text{if } r_{y,m_x} \text{ does not have an assigned priority.} \end{cases}$$

(It is assumed that reaction rules with assigned priorities are labelled in the order that reflects their priority ordering.)

Updating Phase. In the updating phase, reaction rules update multiplicity values of object types in parallel.

Let $M_{m_x}^{\text{UPDATE}} : V \longrightarrow R_{m_x}$ represent the conflicts that exist in the updating phase in the region m_x. $M_{m_x}^{\text{UPDATE}}$ maps an object type $o \in V$ in m_x to the set of reaction rules that might access the multiplicity value for that object type during the updating phase (each of these reaction rules is associated with either m_x, the parent region of m_x, or a child region of m_x). The size of the set which is the value of the function is called the *conflict degree* for the object type o in m_x.

As mentioned in Section 3.2.1, if the time-oriented conflict resolution strategy is used, the number of clock cycles taken to complete the updating phase is equal to the highest conflict degree in the conflict matrix for the P system. As mentioned in Section 3.2.2, if the space-oriented conflict resolution strategy is used, only one clock cycle is needed to complete the updating phase. Therefore

$$Max_{i=1}^{n} Max_{j=1}^{k_{m_i}} u^{\text{PAR}}(r_{j,m_i}^{a}) = \begin{cases} Max_{i=1}^{n} Max_{j=1}^{v} \left| M_{m_i}^{\text{UPDATE}}(o_j) \right|, \\ \qquad \text{if the time-oriented strategy is used,} \\ 1, \\ \qquad \text{if the space-oriented strategy is used.} \end{cases}$$

No matter which conflict resolution strategy is used, synchronisation operations need to be performed. The first synchronisation operation that consumes clock cycles is the updating of `preparationSentinel` that occurs once all reaction rules in the P system have completed their respective preparation phases. As explained in Section 3.1.4, this involves the execution of $\log_x \lceil \sum_{i=1}^{n} |R_{m_i}| \rceil$ Handel-C assignment statements. The first of these assignment statements makes use of a `signal` (instead of a register), and therefore consumes zero clock cycles. Each of the other assignment statements takes one clock cycle to execute. Therefore, the total number of clock cycles consumed in this step is $\log_x \lceil \sum_{i=1}^{n} |R_{m_i}| \rceil - 1$. The second synchronisation operation that consumes clock cycles is the updating of `updatingSentinel` that occurs once all reaction rules in the P system have completed their respective updating phases. This operation is identical in form to the operation of updating `preparationSentinel`, and so consumes $\log_x \lceil \sum_{i=1}^{n} |R_{m_i}| \rceil - 1$ clock cycles. Finally, when the space-oriented conflict resolution strategy is used, coordination of

multiset replication needs to be performed at the end of the updating phase (see Section 3.2.2). Coordinating multiset replication for all regions takes one clock cycle. Therefore, summing the number of clock cycles taken by each synchronisation operation, we have

$$s^{\mathrm{PAR}} = \begin{cases} 2(\log_x \lceil \sum_{i=1}^{n} |R_{m_i}| \rceil - 1), & \text{if the time-oriented strategy is used,} \\ 2(\log_x \lceil \sum_{i=1}^{n} |R_{m_i}| \rceil - 1) + 1, & \text{if the space-oriented strategy is used.} \end{cases}$$

4.4 Comparison of the Algorithms

Regions	Rules	k	a (%)	Number of clock cycles per transition			
				Sequential	1-level parallelism	2-level parallelism (time-oriented)	2-level parallelism (space-oriented)
Horizontal cascading							
10	50	3	30	470	25	10	9
10	50	3	70	630	28	10	9
50	250	3	30	2 350	27	12	11
50	250	3	70	3 150	30	12	11
100	500	3	30	4 700	27	12	11
100	500	3	70	6 300	30	12	11
Vertical cascading							
10	50	3	30	470	25	10	9
10	50	3	70	630	28	10	9
10	250	15	30	2 350	186	36	23
10	250	15	70	3 150	351	36	23
10	500	30	30	4 700	606	66	38
10	500	30	70	6 300	1 206	66	38

Table 3. An illustration of the time complexity results

Table 3 illustrates the relative theoretical performances of (a) the sequential algorithm, (b) an algorithm that implements parallelism only at the system level (as in Petreska and Teuscher's implementation), (c) the parallel algorithm executed by Reconfig-P when the time-oriented conflict resolution strategy is used, and (d) the parallel algorithm executed by Reconfig-P when the space-oriented conflict resolution strategy is used. Larger and larger P systems are derived from one initial basic P system using either horizontal or vertical cascading. In horizontal cascading, more and more regions are added, but the number of reaction rules per region is held constant. In vertical cascading, the number of reaction rules per region increases, but the number of regions is held constant. The following assumptions, deemed to represent the average case, are made: (a) there are 20 object types; (b) each reaction rule has four reactant object types, four catalyst object types and four product object types; (c) there are conflicts on 20 % of the object types in the preparation

phase; and (d) there are conflicts on 60 % of the object types in the updating phase. Assumption (d) gives rise to a k value for each P system, which is the highest conflict degree in the conflict matrix for the P system. P systems are also assigned an arbitrary a value, which is the percentage of reaction rules that are applicable in a transition on average.

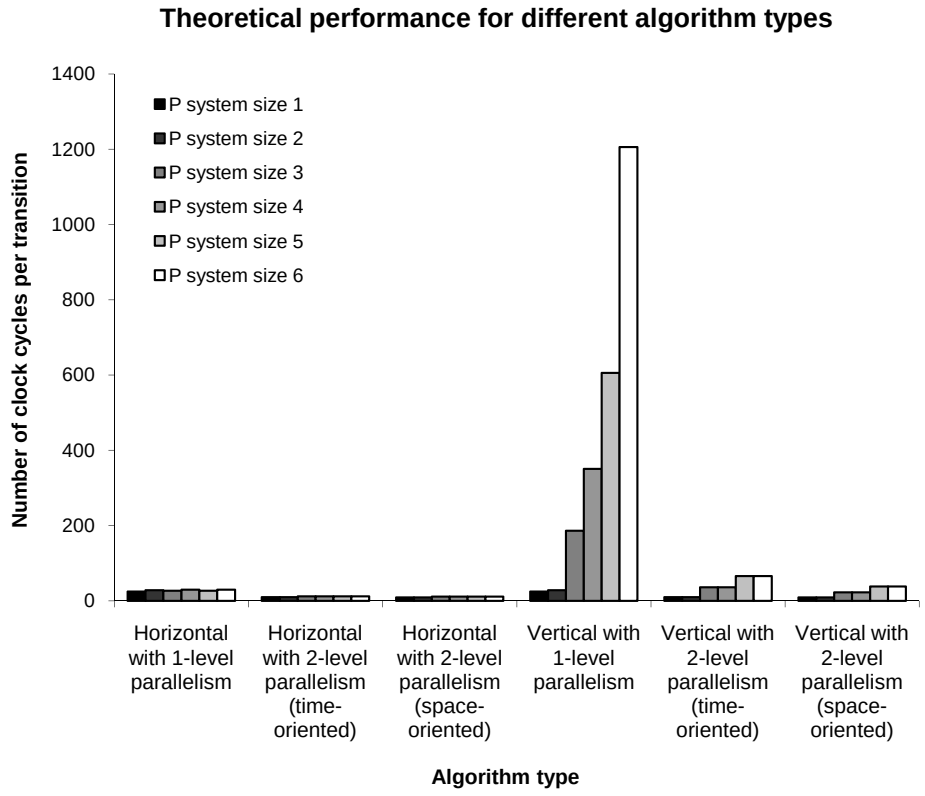

Fig. 5. A graph of the theoretical performance data in Table 3

A graph of the theoretical performance data in Table 3 is shown in Figure 5. The graph clearly demonstrates the superior speed of the algorithm executed by Reconfig-P over both the sequential algorithm and the algorithm with one level of parallelism. When horizontal cascading is applied, the algorithm shows exceptional scalability in both the time-oriented and space-oriented modes. When the k value is small and reaction rules are evenly distributed across regions, the algorithm is more effective in time-oriented mode than in space-oriented mode because it uses less space while achieving similar speeds. When vertical cascading is applied, the algorithm is significantly faster than the algorithm with one level of parallelism. The algorithm is faster in space-oriented mode than in time-oriented mode. The main

reason is that, whereas increasing the k value reduces the degree of parallelism in the updating phase when the time-oriented conflict resolution strategy is used, this is not the case when the space-oriented conflict resolution strategy is used.

5 CONCLUSION

The hardware implementation of membrane computing presented in this paper is the first to achieve parallelism at both the system and region levels. Unlike Petreska and Teuscher's implementation, it tackles the resource conflict problem associated with the updating phase of a P system transition, and is therefore able to achieve parallelism at the region level.

The theoretical performance results presented in Section 4 suggest that the hardware implementation can significantly outperform sequential implementations of membrane computing as well as Petreska and Teuscher's implementation. However, the theoretical performance data are measured in clock cycles per transition, whereas the performance metric of ultimate interest is the amount of real time elapsed per transition. To measure this metric, an empirical performance analysis is required. In [5], we present such an analysis, which shows that the hardware implementation achieves very good performance. Also in [5], we present an empirical analysis of the hardware resource usage of the implementation. The extent to which the implementation consumes hardware resources is important, because it determines the largest P systems that can be executed on a given hardware platform. The results of this analysis show that the implementation makes economical use of hardware resources.

In the next phase of this research, we intend to develop strategies for the implementation in hardware of P system features not covered by the core P system model, and to investigate the use of our hardware implementation for the execution of real-world applications.

REFERENCES

[1] Ciobanu, G.—Guo, W.: P Systems Running on a Cluster of Computers. In: C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg and A. Salomaa (Eds.): Membrane Computing. Lecture Notes in Computer Science, Vol. 2933, 2004, pp. 123–139.

[2] Fernandez, L.—Arroyo, F.—Tejedor, J. A.—Castellanos, J.: Massively Parallel Algorithm for Evolution Rules Application in Transition P Systems. In: Pre-proceedings of the Seventh Workshop on Membrane Computing, WMC7, Leiden, The Netherlands, July 2006, pp. 337–343.

[3] Fernandez, L.—Martinez, V. J.—Arroyo, F.—Mingo, L. F.: A Hardware Circuit for Selecting Active Rules in Transition P Systems. In: Pre-proceedings of the First International Workshop on Theory and Application of P Systems, Timisoara, Romania, September 2005, pp. 45–48.

[4] Nepomuceno-Chamarro, I. A.: A Java Simulator for Basic Transition P Systems. Journal of Universal Computer Science, Vol. 9, 2004, No. 5, pp. 620–629.

[5] Nguyen, V.—Kearney, D.—Gioiosa, G.: Balancing Performance, Flexibility, and Scalability in a Parallel Computing Platform for Membrane Computing Applications. In: G. Eleftherakis et al. (Eds.): Membrane Computing. Lecture Notes in Computer Science, Vol. 4860, 2007, pp. 385-413.

[6] Păun, G.: Membrane Computing: An Introduction. Springer, 2002.

[7] Petreska, B.—Teuscher, C.: A Reconfigurable Hardware Membrane System. In: C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg and A. Salomaa (Eds.): Membrane Computing. Lecture Notes in Computer Science, Vol. 2933, 2004, pp. 269–285.

Van Nguyen received the Bachelor of Information Technology (Software Engineering) with first-class honours from the University of South Australia in 2006. She is currently a Ph. D. candidate in computer science at the University of South Australia. Her research focuses on hardware implementations of membrane computing. She also lectures on web-based applications. She has expertise in the development of web-based applications using Java EE technologies.

David Kearney is an associate professor and director of the Reconfigurable Computing Laboratory at the University of South Australia. He is an author of more than 70 publications, including the most cited publication on operating systems for reconfigurable computing. His research program in reconfigurable computing aims at identifying key algorithms and abstractions that will allow software engineers to easily develop applications, to discover new programming language concepts for the expression of parallel algorithms, and to discover how to best exploit the advantages of the reconfigurable computing paradigm in implementations of popular parallel algorithms from the domains of image processing and bio-inspired computing.

Gianpaolo Gioiosa received the Bachelor of Information Technology (Advanced Computer and Information Science) with first-class honours from the University of South Australia in 2005. He is currently a Ph. D. candidate in computer science at the University of South Australia. His research interests include top-level ontologies and membrane computing.