The Minimal Parallelism Is Still Universal

Gabriel Ciobanu
Research Institute "e-Austria" Timişoara, Romania
Bd. V. Pârvan 4, 300223 Timişoara, Romania, and
Romanian Academy, Institute of Computer Science, Iaşi
Bd. Copou 8, 700505 Iaşi, Romania
E-mail: gabriel@iit.tuiasi.ro

Gheorghe Păun
Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania, and
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: george.paun@imar.ro, gpaun@us.es

Abstract. A current research topic in membrane computing is to find more (biologically) realistic P systems, and one target in this respect is to relax the condition of using the rules in a maximally parallel way. We contribute in this paper by considering a *minimal* parallelism: if a rule from a set of rules within a membrane or a region may be used, then at least one rule from that membrane or region could be used. Restrictive as it might look, this minimal parallelism still leads to universality, at least for the case of symport/antiport rules. The result is obtained both for generating and accepting P systems, in the latter case also for systems working deterministically. The problem remains open for other classes of P systems.

1 Introduction

Membrane computing and P systems were introduced in [10]. The monograph [11] provides a comprehensive introduction, and various applications are presented in [2]. The web page http://psystems.disco.unimib.it presents recent developments. A debated topic was how "realist" various classes of P systems, hence various ingredients used in them, are from a biological point of view. Starting from the observation that there is an obvious parallelism in the cell biochemistry [1], and relying on the assumption that "if we wait enough, then all reactions which may take place will take place", a feature of the P systems is given by the maximal parallel way of using the rules (in each step, in each region of a system, we can use a maximal multiset of rules). This condition provides a useful tool in proving various properties, because it decreases the non-determinism of the systems evolution, and it supports techniques for simulating the powerful feature of *appearance checking*, a standard method used to get computational completeness/universality of regulated rewriting in formal language theory [5].

For various reasons ranging from looking for more realistic models to just more mathematical challenge, the maximal parallelism was questioned, either simply criticized, or replaced with presumably less restrictive assumptions. For instance, sequential systems were considered in a series of papers, where only one rule is used in each step of a computation. An intermediate case is that of non-synchronized P systems, where any number of rules is used in each step. More details and references can be found in [6]. Various classes of systems with computations similar to the cooperation modes of grammar systems [3] are presented in [4]: in each step, in each region, for a given k, we can use exactly k, at least k, or at most k rules. In general, sequential systems and non-synchronized systems are weaker than systems based on maximal parallelism, but the universality is obtained again as soon as further controls are considered, such as priorities, bi-stable catalysts, etc.

In this paper we propose a rather natural condition which, as far as we know, was not yet considered: from each set of rules where at least a rule can be used, then at least one is actually used (may be more, without any restriction). We say that this is a *minimal parallelism*, because this mode of using the rules ensures that all compartments of the system evolve in parallel, by using at least one rule whenever such a rule is applicable. We have two main cases given by the fact that the rules can be associated with a region of a P system (in the case of rules associated with regions, as in P systems with symbol-objects processed by rewriting-like rules), or they can be associated with a membrane (in the case of rules associated with membranes, as in symport/antiport P systems). The minimal parallelism refers to the fact that from each such set at least one rule is applied. Of course, for systems with only one membrane the minimal parallelism is nothing else than non-synchronization, hence the non-trivial case is that of multi-membrane systems.

In what follows, we consider only the case of P systems with symport/antiport rules [9]. Somewhat surprisingly, the universality is obtained with a small number of membranes (three in the generative case, and two in the accepting case), and with rather simple symport and antiport rules (of weight two). The universality is obtained both for systems working in the generative mode in which one collects the results of all halting computations, defined as the multiplicity of objects from an output membrane in the halting configuration, as well as in the accepting mode where a number is introduced in the system, in the form of the multiplicity of a specified object in a specified membrane, and the number is accepted if the computation halts. Moreover, in the accepting case the system can be deterministic.

The minimal parallelism for other classes of P systems, especially for catalytic systems, remains to be investigated. It is also of interest to consider the minimal parallel mode for systems designed to solve hard problems in a feasible time, where the maximal parallelism seems to have an important role (see [12]).

2 Prerequisites

We suppose that the reader is familiar with the basic elements of Turing computability [7], and of membrane computing [11]. We introduce here, in a rather informal way, only the necessary notions and notation.

An *alphabet* is a finite and non-empty set of abstract symbols. For an alphabet A we denote by A^* the set of all strings of symbols from A, including the empty string λ. A *multiset* over an alphabet A is a mapping from A to $\mathbf{N}$, the set of natural numbers; we represent a multiset by a string from A^*, where the number of occurrences of a symbol $a \in A$ in a string w represents the multiplicity of a in the multiset represented by w (hence all strings obtained by permuting symbols in the string w represent the same multiset). The

family of Turing computable sets of natural numbers is denoted by NRE (with RE coming from "recursively enumerable").

In our universality proof presented in the next section, we use a characterization of NRE by means of *register machines*. Such a device consists of a given number of registers, each of which can hold an arbitrarily large natural number, and a set of labelled instructions which specify how the numbers stored in registers can change, and which instruction should follow after any used instruction. There are three types of instructions:

- $l_1 : (\texttt{ADD}(r), l_2, l_3)$ add 1 to register r, and then go to one of the instructions labelled by l_2 and l_3, non-deterministically chosen,
- $l_1 : (\texttt{SUB}(r), l_2, l_3)$ if register r is non-empty (non-zero), then subtract 1 from it and go to the instruction labelled by l_2, otherwise go to the instruction labelled by l_3,
- $l_h : \texttt{HALT}$ the halt instruction.

A *register machine* is a construct $M = (n, H, l_0, l_h, R)$, where n is the number of registers, H is the set of instruction labels, l_0 is the start label, l_h is the halt label (assigned to HALT), and R is the set of instructions. Each label of H labels only one instruction from R, thus precisely identifying it. A register machine M generates a set $N(M)$ of numbers in the following way: we have initially all registers empty (hence storing the number zero), start with the instruction labelled by l_0, and proceed to apply instructions as indicated by the labels and by the contents of registers. If we reach the halt instruction, then the number stored at that time in register 1 is said to be computed by M, and hence it is introduced in $N(M)$. Since we have a non-deterministic choice in the continuation of the computation in the case of ADD instructions, $N(M)$ can be an infinite set. It is known (see [8]) that in this way we can compute all the sets of numbers which are Turing computable, even using a small number of registers (however this detail is not of interest here).

Without loss of generality, we may assume that in all instructions $l_1 : (\texttt{ADD}(r), l_2, l_3)$ we have $l_2 \neq l_1$, $l_3 \neq l_1$, and the unique instruction labelled by l_0 is an ADD instruction. Moreover, we may assume that the registers of the machine except (possibly) register 1 are empty in the halting configuration.

A register machine can also work in an accepting mode. The number to be accepted is introduced in register 1, with all other registers empty. We start computing with the instruction labelled by l_0; if the computation halts, then the number is accepted (the contents of the registers in the halting configuration does not matter). We still denote by $N(M)$ the set of numbers accepted by a register machine M. In the accepting case, we can also consider deterministic register machines, namely with all instructions $l_1 : (\texttt{ADD}, l_2, l_3)$ having $l_2 = l_3$. It is known that deterministic accepting register machines characterize NRE.

A *P system with symport/antiport rules* is a construct of the form

$$\Pi = (O, \mu, w_1, \ldots, w_n, E, R_1, \ldots, R_n, i_0),$$

where:

1. O is the alphabet of objects,
2. μ is the membrane structure (of degree $n \geq 1$, with the membranes labelled in a one-to-one manner with $1, 2, \ldots, n$),

3. $w_1, \ldots, w_n$ are strings over O representing the multisets of objects present in the n compartments of μ in the initial configuration of the system,

4. $E \subseteq O$ is the set of objects supposed to appear in the environment in arbitrarily many copies,

5. $R_1, \ldots, R_n$ are the (finite) sets of rules associated with the m membranes of μ,

6. $i_0 \in H$ is the label of a membrane of μ, which indicates the *output* region of the system.

The rules of any R_i can be of two types: *symport* rules of form $(x, in), (x, out)$, and *antiport* rules of form $(u, out; v, in)$, where x, u, v are strings over O. The length of x, respectively the maximum length of u, v, is called the *weight* of the corresponding (symport or antiport) rule.

In what follows, the rules are used in the *non-deterministic minimal parallel* manner: in each step, from each set R_i $(1 \leq i \leq n)$ we can use at least one rule (without specifying how many) provided that this is possible. The rules to be used, as well as the objects to which they are applied, are non-deterministically chosen. More specifically, we assign non-deterministically objects to rules, starting with one rule from each set R_i. If for a set R_i we cannot enable a rule, then this set remains idle. After having one rule enabled in each set R_i, for which this is possible, if we still have objects which can evolve, then we evolve them by any number of rules (possibly none).

We emphasize an important aspect: the competition for objects. It is possible that rules from two sets R_i, R_j associated with adjacent membranes i, j (that is, membranes which have access to a common region, either horizontally or vertically) use the same objects, so that only rules from one of these sets can be used. Such conflicts are resolved in the following way: if possible, objects are assigned non-deterministically to a rule from one set, say R_i; then, if possible, other objects are assigned non-deterministically to a rule from R_j, and thus fulfilling the condition of minimal parallelism. After that, further rules from R_i or R_j can be used for the remaining objects, non-deterministically assigning objects to rules. If no rule from the other set (R_j, respectively R_i) can be used after assigning objects to a rule from R_i or R_j, then this is a correct choice of rules. This depends on the first assignment of objects to rules making applicable rules from each set.

An example can illuminate this aspect: let us consider the configuration

$$[\ [\ ]_i\ aab\ [\ ]_j\ ]_k,$$

with $R_i = \{(b, in)\}$, and $R_j = \{(a, in),\ (b, in)\}$. The objects aab are available to rules from both sets. All the following five assignments are correct: b goes to membrane i by means of the rule $(b, in) \in R_i$, and one or two copies of a is/are moved to membrane j by means of the rule $(a, in) \in R_j$; b goes to membrane j by means of the rule $(b, in) \in R_j$, no rule from R_i can be used, and zero, one, or two copies of a are also moved to membrane j by means of the rule $(a, in) \in R_j$. Moving only b to membrane i and not using at least once the rule $(a, in) \in R_j$ is not correct.

As usual in membrane computing, we consider successful only computations which halt, and with a halting computation we associate a result, in the form of the number of objects present in the halting configuration in region i_0 (note that this region is not necessarily an elementary one). External output or a squeezing through a terminal set of objects are also possible, but we do not consider these cases here. The set of numbers generated by a system Π in the way described above is denoted by $N_{gen}(\Pi)$. The family of all sets $N_{gen}(\Pi)$ generated by systems with at most $n \geq 1$ membranes, using symport rules of weight at most $p \geq 0$,

and antiport rules of weight at most $q \geq 0$ are denoted by $N^{gen}_{mp}OP_n(sym_p, anti_q)$, with the subscript "$mp$" indicating the "minimal parallelism" used in computations.

It is also possible to work in the accepting mode, starting with a number introduced in an input region, and accepting it if the computation stops.

We can also use a P system as above in the accepting mode: For a specified object a, we add a^m to the multiset of region i_0, and then we start working; if the computation stops, then the number m is accepted. We denote by $N_{acc}(\Pi)$ the set of numbers accepted by a system Π, and by $N^{acc}_{mp}OP_n(sym_p, anti_q)$ the family of all sets of numbers $N_{acc}(\Pi)$ accepted by systems with at most n membranes, using symport rules of weight at most p and antiport rules of weight at most q.

In the accepting case we can consider a deterministic system where at most one continuation is possible in each step. In our context, this means that zero or exactly one rule can be used from each set R_i, $1 \leq i \leq n$. We add a letter D in front of $N^{acc}_{mp}OP_n(sym_p, anti_q)$ for denoting the corresponding families by $DN^{acc}_{mp}OP_n(sym_p, anti_q)$.

3 Universality

Our expectation of introducing the minimal parallelism was to obtain a decidable class of P systems. This turns not to be true: the minimal parallelism still contains the possibility to enforce the checking for zero from SUB instructions of the register machines (this is the same with appearance checking from regulated rewriting). Therefore we get again an universal system.

Theorem 3.1 *$N^{gen}_{mp}OP_n(sym_p, anti_q) = NRE$ for all $m \geq 3,\ p \geq 2,\ q \geq 2$.*

Proof. It is sufficient to prove the inclusion $NRE \subseteq N^{gen}_{mp}OP_2(sym_2, anti_2)$: the inclusions $N^{gen}_{mp}OP_n(sym_p, anti_q) \subseteq N^{gen}_{mp}OP_{n'}(sym_{p'}, anti_{q'}) \subseteq NRE$, for all $1 \leq n \leq n'$, $0 \leq p \leq p'$, and $0 \leq q \leq q'$ are obvious.

Let us consider a register machine $M = (n, H, l_0, l_h, R)$. We consider an object $g(l)$ for each label $l \in H$, and denote by $g(H)$ the set of these objects. We define $H' = \{l' \mid l \in H\}$, $H'' = \{l'' \mid l \in H\}$, and $H''' = \{l''' \mid l \in H\}$, where l', l'' and l''' are new symbols associated with $l \in H$. For an arbitrary set of objects Q, let us denote by $w(Q)$ a string which contains exactly once each object from Q; thus $w(Q)$ represents the multiset which consists of exactly one occurrence of each object of Q. In general, for any alphabet Q we denote $Q' = \{b' \mid b \in Q\}$, $Q'' = \{b'' \mid b \in Q\}$, and $Q''' = \{b''' \mid b \in Q\}$ (where b', b'', b''' are new symbols associated with $b \in Q$).

We construct the P system (of degree 3)

$$\Pi = (O, \mu, w_1, w_2, w_3, E, R_1, R_2, R_3, 3)$$

with

$$\begin{aligned}
O &= \{l, l', l'', l''', g(l) \mid l \in H\} \cup \{a_r \mid 1 \leq r \leq n\} \cup \{c, d, t\},\\
\mu &= [\ [\ [\ ]_3\ ]_2\ ]_1,\\
w_1 &= w(H - \{l_0\})w(H')w(H'')ctt,\\
w_2 &= w(g(H)),\\
w_3 &= \lambda,\\
E &= \{a_r \mid 1 \leq r \leq n\} \cup H''' \cup g(H) \cup \{c, d, l_0\},
\end{aligned}$$

and with the following sets of rules (we present the rules in groups associated with various tasks during the computation, also indicating the steps when they are used).

1. **Starting the computation:**

R_1	R_2	R_3
$(c, out; ca_r, in), 1 \leq r \leq n$	–	–
$(c, out; dl_0, in)$	–	–
–	(dl_0, in)	–

As long as c is present in region 1, we can bring copies of objects $a_r, 1 \leq r \leq n$, in the system, by using rule $(c, out; ca_r, in) \in R_1$. The number of copies of a_r $(1 \leq r \leq n)$ from region 2 represent the number stored in register r of the register machine. At any moment we can use a rule $(c, out; dl_0, in)$ which brings the initial label of M inside (note that l_0 does not appear in w_1), together with the additional object d. From now on, no further object a_r $(1 \leq r \leq n)$ can be brought in the system. In the next step, the objects dl_0 can enter region 2 by means of rule $(dl_0, in) \in R_2$, and this initiates the simulation of a computation in our register machine M.

2. We also have the following **additional rules** which are used in interaction with rules from the other groups, including the previous group:

R_1	R_2	R_3
(l_0, out)	$(g(l), out; dt, in), l \in H$	–
–	$(t, out; t, in)$	–

The rôle of these rules is to prevent "wrong" steps in Π. For instance, as soon as the instruction with label l_0 is simulated (see details below), label l_0 should be sent out of the system in order to avoid another computation in M, and this is done by means of rule $(l_0, out) \in R_1$. This rule should be used as soon as l_0 is available in region 1, and no other rule from R_1 may be applied – we will see immediately that this is ensured by the rules simulating the instructions of M. However, l_0 should not be eliminated prematurely, i.e., before simulating the instruction with label l_0. In such a case, a rule $(g(l), out; dt, in)$ for some $l \in H$ should be used once and only once, because we have only one available copy of d. In this way, a copy of t enters membrane 2, and the computation can never stop because of the rule $(t, out; t, in) \in R_2$. This trap-rule is used also in order to control the correct simulation of instructions of M.

3. **Simulating an instruction** $l_1 : (\texttt{ADD}(r), l_2, l_3)$:

Step	R_1	R_2	R_3
1	–	$(l_1, out; l_k a_r, in), k = 2, 3$, or $(l_1, out; t, in)$	–

When l_1 is in region 2, the number of copies of a_r is incremented by bringing a_r from the skin region, together with any of labels l_2 and l_3. Since we have provided in w_2 one copy of any label different from l_0, labels l_2 and l_3 are present in region 1. We have assumed that each ADD instruction of M has $l_2 \neq l_1$ and $l_3 \neq l_1$, hence we do not need two copies of any $l \in H$ in Π. If we have not enough copies of a_r because the initial phase was concluded prematurely, and we have brought some copies of a_r in the system, but all of them were already moved in membrane 2, then we cannot use a rule

$(l_1, out; l_k a_r, in), k = 2, 3$. Because of the minimal parallelism, the rule $(l_1, out; t, in)$ should be used (no other rule from R_2 is applicable). Therefore the computation will never end, because of the trap-rule $(t, out; t, in) \in R_2$.

Note that no rule from R_1 can be used, excepting (l_0, out), whenever l_0 is present in the skin region (which means that the initial instruction was already simulated).

4. **Simulating an instruction** $l_1 : (\texttt{SUB}(r), l_2, l_3)$:

Step	R_1	R_2	R_3
1	–	$(g(l_1)l_1, out; l_1', in)$	–
2	$(g(l_1), out; g(l_1)l_1''', in)$	$(l_1' a_r, out; l_1'', in)$	–
3	–	$(g(l_1)l_1''', in)$	–
4	–	$(l_1'''l_1', out; l_3, in)$ or $(l_1'''l_1'', out; l_2, in)$	–
1	(l_1''', out)	.....	–
–	–	$(d, out; l_1'''t, in)$	–

With l_1 present in region 2, we use a rule $(g(l_1)l_1, out; l_1', in) \in R_2$ by which the witness object $g(l_1)$ goes to the skin region, and the object l_1' is brought into region 2. During the next step, this latter object checks whether there is any copy of a_r present in region 2. In a positive case, rule $(l_1' a_r, out; l_1'', in) \in R_2$ is used in parallel with $(g(l_1), out; g(l_1)l_1''', in) \in R_1$. This last rule brings into the system a "checker" l_1'''.

In the next step we have two possibilities.

If we use a rule $(g(l_1)l_1''', in) \in R_2$, then both $g(l_1)$ and l_1''' are introduced in region 2. Depending on what l_1''' finds here, one can use either $(l_1'''l_1', out; l_3, in)$ if the subtraction was not possible (because no a_r was present), or $(l_1'''l_1'', out; l_2, in)$ if the subtraction was possible. Thus, only the correct label selected between l_2 and l_3 is brought into region 2.

If in step 3, instead of a rule $(g(l_1)l_1''', in) \in R_2$, we use again $(g(l_1), out; g(l_1)l_1''', in) \in R_1$, then the existing object l_1''' either exits the system by means of the rule $(l_1''', out) \in R_1$, and hence we repeat the configuration, or we use a rule $(d, out; tl_1''', in) \in R_2$ and the computation never stops (no other rule of R_2 can be used). Similarly, if in step 3 we use the rule $(l_1''', out) \in R_1$, then at the same time the object $g(l_1)$ from region 1 either remains unchanged, or uses the rule $(g(l_1), out; g(l_1)l_1''', in) \in R_1$. In the former case, in the next step we have to use the rule $(g(l_1), out; g(l_1)l_1''', in) \in R_1$ (no other rule of R_1 can be applied). In both cases, the configuration is repeated. Of course, if we directly use the rule $(d, out; tl_1''', in) \in R_2$, then the computation never stops.

Thus, in all these cases, the simulation of SUB instruction is correct (or the system never stops). In the next step, namely step 1 of the simulation of any type of instruction, the object l_1''' exits the system by means of rule $(l_1''', out) \in R_1$. Note that no other rule of R_1 is used in this step, neither for ADD, nor for SUB instructions.

Consequently, both ADD and SUB instructions of M are correctly simulated. The configuration of the system is restored after each simulation, namely all objects of $g(H)$ are in region 1, and no object of H''' is present in the system. The simulation of instructions can be iterated, finally simulating in Π a computation of M. If a computation of M does not stop, then its simulation in Π does not stop as well. When the computation in M stops, then a label l_h is introduced in region 2, and all registers $2, 3, \ldots, n$ are empty. Then we use the following group of rules.

5. **Final rules:**

R_1	R_2	R_3
–	–	$(l_h a_1, in)$
–	–	(l_h, out)

The halting label l_h carries all copies of a_1 from region 2 to region 3, and only when this process is completed the computation can stop.

Consequently, the system Π correctly simulates (all and nothing else than) the computations of M. This means that $N_{gen}(\Pi) = N(M)$. □

Note that in this proof the output region is an elementary one, and that the minimal parallelism is essentially used in ensuring the correct simulation of the computations in the register machine M. Otherwise, it is possible to stop the computation in Π without completing the simulation of a halting computation in M.

The previous proof can be easily adapted for the accepting case: we consider l_0 already in region 1, and introduce additionally a_1^m, for m being the number we want to accept. Omitting the technical details, we provide the following result:

Corollary 3.1 *$N_{mp}^{acc}OP_n(sym_p, anti_q) = NRE$ for all $m \geq 2,\ p \geq 2,\ q \geq 2$.*

Moreover, this last result can be strengthened by considering deterministic P systems, and also by decreasing with one the number of membranes.

Theorem 3.2 *$DN_{mp}^{acc}OP_n(sym_p, anti_q) = NRE$ for all $m \geq 2,\ p \geq 2,\ q \geq 2$.*

Proof. Let us consider a deterministic register machine $M = (n, H, l_0, l_h, R)$ accepting an arbitrary set $N(M) \in NRE$. We also use the notation Q^u for the sets $\{b^u \mid b \in Q\}$, for various markings (priming) u, and $w(Q)$, as specified in the proof of Theorem 3.1. We construct the P system (of degree 2)

$$\Pi = (O, [\ [\ ]_2\ ]_1, w_1, w_2, E, R_1, R_2, 1),$$

with

$$\begin{aligned} O &= \{l, l', l'', l''', l^{iv}, l^v \mid l \in H\} \cup \{a_r \mid 1 \leq r \leq n\} \cup \{c\}, \\ w_1 &= l_0, \\ w_2 &= w(H^{iv}), \\ E &= O, \end{aligned}$$

and with the following sets of rules presented in two groups.

The computation starts by introducing in region 1 a multiset a_1^m encoding the number m. Note that the initial label of M is already present here.

1. **Simulating an instruction** $l_1 : (\texttt{ADD}(r), l_2, l_2)$:

Step	R_1	R_2
1	$(l_1, out; l_2 a_r, in)$	–

Since the environment contains the necessary objects, ADD instruction is simulated whenever we have an object l_1 in region 1.

2. **Simulating an instruction** $l_1 : (\texttt{SUB}(r), l_2, l_3)$:

Step	R_1	R_2
1	$(l_1, out; l_1' l_1'', in)$	–
2	$(l_1' a_r, out; l_1''', in)$	$(l_1^{iv}, out; l_1'', in)$
3	$(l_1^{iv} l_1', out; l_1^{iv} l_3^v, in)$ or $(l_1^{iv} l_1'', out; l_1^{iv} l_2^v, in)$	–
4	$(l_3^v, out; cl_3, in)$, resp. $(l_2^v, out; cl_2, in)$	–
1	.....	(cl_1^{iv}, in)

With l_1 present in region 1, we bring into the system (from the inexhaustible environment) the objects l_1', l_1''. l_1' is used to subtract 1 from register r by leaving together with a copy of a_r. This is bringing in the system the object l_1''', and, simultaneously, l_1'' enters region 2, releasing the "checker" l_1^{iv}. In the next step, depending on what l_1^{iv} finds in membrane 1, either $(l_1^{iv} l_1', out; l_1^{iv} l_3^v, in)$ is used whenever subtraction is not possible, or $(l_1^{iv} l_1'', out; l_1^{iv} l_2^v, in)$ is used whenever subtraction is possible. The corresponding l_2^v or l_3^v is brought into the system. In the fourth step, these last objects exit the system and bring inside their corresponding non-marked objects l_2 and l_3, together with an auxiliary object c.

In the next step, object c "helps" object l_1^{iv} (which waits for one step in the skin region) to return to region 2, and so making sure that the multiset of region 2 contains again all objects of H^{iv}. This step is the first one of simulating another ADD or SUB instruction, and it is important to note that in this step no other rule from R_2 can be used.

By iterating such simulations of ADD and SUB instructions, we can simulate any computation of M. Obviously, since any computation of M is deterministic, then the computation in Π is also deterministic (from each set R_i, in each step, at most one rule is applicable). When a halt instruction is reached, hence l_h appears in the system, the computation stops (with a further last step when a rule $(cl_1^{iv}, in) \in R_2$ is used).

We conclude that $N_{acc}(\Pi) = N(M)$. □

Note that the deterministic construction is simpler than that from the proof of Theorem 3.1, as we should not initially bring "resources" into the system, and we have nothing to do in the final stage (in particular, we should not "clean" of any other objects than those we need to count).

4 Final Remarks

The minimal parallelism remains to be considered for other classes of P systems, starting with catalytic systems and systems with active membranes used in addressing hard problems in the membrane computing framework. From a mathematical point of view, it is also of interest to look for improvements of the result in Theorems 3.1 and 3.2 in what concerns the number of membranes in the former theorem and the weights of symport and antiport rules in both theorems – as done in a series of papers for the case of maximal parallelism. It should be determined how many of the known results are also valid for the minimal parallelism, and how many results can be re-proven in the new framework.

A more interesting problem is to find mechanisms of "keeping under control" the power of P systems in order to obtain non-universal classes of systems. Minimal parallelism fails, the sequential use of rules is similarly restrictive as the maximal parallelism, non-synchronization is too loose and weak in order to be used in "programming" the P systems. What else to consider is a topic to investigate, and it would be useful to go back to biology, and get inspiration from the living cell.

Acknowledgements. Thanks are due to Linqiang Pan for a careful reading of a previous version of the paper. The work of the second author was partially supported by National Natural Science Foundation of China (Grant No. 60373089).

References

[1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter: *Molecular Biology of the Cell*, 4th ed. Garland Science, New York, 2002.

[2] G. Ciobanu, Gh. Păun, M.J. Pérez–Jiménez, eds.: *Applications of Membrane Computing*. Springer-Verlag, Berlin, 2005.

[3] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun: *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach, London, 1994.

[4] Z. Dang, O.H. Ibarra: On P systems operating in sequential mode. *Pre-proc. of DCFS Workshop* (L. Ilie, D. Wotschke, eds.), London-Ontario, 2004, 164–177.

[5] J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag, Berlin, 1989.

[6] R. Freund: Asynchronous P systems and P systems working in the sequential mode. In *Membrane Computing, International Workshop, WMC5, Milano, Italy, 2004, Selected and Invited Papers* (G. Mauri, Gh. Păun, M.J. Pérez–Jiménez, G. Rozenberg, A. Salomaa, eds.), LNCS 3365, Springer-Verlag, Berlin, 2005, 36–62.

[7] J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[8] M. Minsky: *Computation. Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.

[9] A. Păun, Gh. Păun: The power of communication: P systems with symport/antiport. *New Generation Computing*, 20, 3 (2002), 295–306.

[10] Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61 (2000), 108–143.

[11] Gh. Păun: *Membrane Computing. An Introduction*. Springer-Verlag, Berlin, 2002.

[12] M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini: Hard problems addressed through P systems. In [2].