Communication Complexity of dP Automata Accepting $\{(ac)^s(bd)^s \mid s \geq 0\}$

Kelvin Cui Buño
Algorithms and Complexity Laboratory
University of the Philippines Diliman
Quezon City, Philippines
kcbuno@up.edu.ph

Richelle Ann B. Juayong
Algorithms and Complexity Laboratory
University of the Philippines Diliman
Quezon City, Philippines
rbjuayong@up.edu.ph

Henry N. Adorna
Algorithms and Complexity Laboratory
University of the Philippines Diliman
Quezon City, Philippines
ha@dcs.upd.edu.ph

ABSTRACT

Păun and Pérez-Jiménez introduced a variant of P systems that computes in a distributed way called distributed P systems (dP systems)[9]. They also introduced a distributed computing P automata[3][4] called dP automata. [9] gives a dP automata with two components recognizing the language $L = \{(ac)^s(bd)^s | s \geq 0\}$. This dP automata uses two component P systems that communicate bidirectionally, using a single intercomponent communication rule, and runs in $2s + 2$ steps. In this paper, we have shown that the language L can be recognized by an n-component dP automata that runs in $\frac{4s}{n} + C$ steps, for some constant C.

1. INTRODUCTION

Membrane computing has two main classes of P systems, namely cell-like P systems and tissue-like P Systems. Cell-like P systems is composed of hierarchical structure of membrane. Membranes of the cell-like P systems each has multiset of objects within their regions and computes in a maximally parallel way. Tissue-like P systems emulates how cells interact within a tissue system. The cells in tissue-like P systems can be represented as a graph, all of the same level. The cells communicate through synapses using symport/antiport rules. Symport rules communicate objects in a unidirectional way. Antiport rules communicate objects in a bidirectional manner.

In [5], Csuhaj-Varjú et. al. introduced a system similar to tissue-like P systems called P Colonies. In P Colonies, its components referred to as agents act and evolve in a shared environment. No direct communication happens in between these agents, agents are not connected through links like in tissue-like P systems. Agents communicate through the changing environment which is initially uniform. Agents change the environment using rules similar to antiport rules.

Early proposal of communication complexity of P systems were made in [1]. ECP with energy is a cell-like P system using both evolution rules and symport/antiport rules. It is defined in a way that using symport/antiport rules can only happen when enough quanta of energy is present within the membranes. The quanta of energy is represented by multiset of objects, usually over the alphabet $\{e\}$. Using symport/antiport rules would decrease the multiplicity of e's within the membrane regions. Communication complexity of ECP is the measure of the maximum quanta of energy used, the communication effort of using symport/antiport rules. Although there is a communication analysis of the system, it is not an analysis for an explicitly distributed system.

An explicit form of distributed systems is wherein given a problem, the problems is broken down, or partitioned into subproblems and each component of the system will take in a subproblem computing in a parallel manner, and then constructing a solution to the initial problem. Although, ideally, these components should compute independently, it is sometimes unavoidable that the components need partial results from other components, thus a need for communication. If components become too dependent on the partial results from other components, the resulting computation time may increase which can be seen as loss in performance.

In [9], a distributed version of cell-like P systems was introduced, called dP Systems. Similar to tissue-like P systems, the dP Systems has its components connected through synapses, but each component in itself is a complete P system. Components of dP systems can be defined to interact with the environment, taking in arbitrarily many copies of objects needed by the system for computation. Additionaly, using the environment, the idea of partitioned input can be made, components only interact with their partition.

The idea of dP systems is to (1) take in a problem or a string, (2) partition it according to the number of components the system has, (3) components compute in a maximally parallel manner with minimal communication through the synapses, and (4) the output of the system is determined by all of the components. To say that the system uses minimal communication, three communication complexity measures were defined for dP Systems. $ComN$ measures the number of steps in which a communication happens throughout the compu-

tation. $ComR$ measures the number of rules used during the computation. And $ComW$ measures the number of objects communicated throughout the computation. A minimal communication is defined to be a communication complexity measure upper bounded by a constant value. Another measure made is the analysis of the parallel efficiency, given by the speed up ratio between the dP system, and a usual P system solving the same problem. From the parallel efficiency, we can determine if using dP system is better or worse than a usual P system solving the same problem. Note that a usual P system can be thought of as a single component dP system.

In this paper, we look into a particular language $L = \{(ac)^s (bd)^s \mid s \geq 0\}$ given in [9]. Păun and Pérez-Jiménez have already given some results for its communication complexity and parallelizability using a two component dP Automaton. We will build a three component dP Automaton and compare its communication complexity and parallelizability with that of the two component construct. We then generalize it to n-components and show that the language is efficiently parallelizable for all n-component dP Automaton, $n \geq 2$.

Certain languages such as $L = \{(ac)^s (bd)^s \mid s \geq 0\}$ can seemingly be recognized by distributed parallel manner without loss of performance even if the number of processors is increased indefinitely. As we know, increasing the number of processors for parallel computation may decrease parallel efficiency compared to the sequential computation. But for languages such as L, increasing the number of components of the dP Automata solving it also increases the speed-up ratio, meaning higher parallel efficiency. We look into L as a candidate member for such languages exhibiting this kind of property.

We discuss briefly what each section of the paper contains. The Definition section provides the terminology and notations used for this paper. This section also provides the definition of the dP system we will use and we state the description of the language we would focus on. Section 3 defines the dP systems that uses P automata components called dP Automata. Results regarding the studied language from [9] will be discussed in this section. Section 4 provides a construction of dP automata using 3 P automaton components to recognize the studied language. The section also gives results on the communication measure and the parallel efficiency of the constructed system. The n-dP Automata section will extend the results from section 3 and section 4 to an n-component P automaton model. The last section will provide a summary and some open problems from the results.

2. DEFINITION

The reader is assumed familiar with the basics of membrane computing and of formal language theory. Here we give the formal definitions of dP Systems and dP Automata, as well as the methods of measuring the communication costs of these systems.

Let Σ be an alphabet. Let x and y be strings over Σ. $|x|$ denotes the length of the string x. If $|x| = 1$, x is referred to as a character. xy denotes the concatenation of strings x and y.

We denote the set of finite multiset over the alphabet V as V^o, and the set of their sequences as $(V^o)^*$. We denote $u \in V^o$ by the corresponding string $a_1^{u(a_1)} a_2^{u(a_2)} ... a_t^{u(a_t)} \in V^*$.

A P automaton, Π, defined by [3][4] is a symport/antiport P-system that accepts strings. The system takes in from the environment a sequence of objects referred to as the input sequence. Let $v_1, ..., v_n$ be an input sequence of Π. Let g be a homomorphic mapping from $V^o \rightarrow \Sigma^*$. We refer to the homomorphic image of the input sequence as the string of Π or Π string and is denoted by $g(v_1)...g(v_n)$. A Π string $g(v_1)...g(v_n)$ is an accepted string if and only if the input sequence $v_1, ..., v_n$ leads Π to a halting computation.

We define a dP Scheme as follows:

DEFINITION 1. *[9] A dP scheme of degree $n \geq 1$ is a construct*

$$\Delta = (\, O, \Pi_1, ..., \Pi_n, R),$$

where:

- *O is an alphabet of objects;*
- *$\Pi_1, ..., \Pi_n$ are cell-like P systems with O as the alphabet of objects and the skin membranes are labeled with $s_1, ..., s_n$ respectively;*
- *R is a finite set of rules of the form $(s_i, u/v, s_j)$, where $1 \leq i, j \leq n, i \neq j$, and $u, v \in O^*$, with $uv \neq \lambda$; $|uv|$ is called the weight of the rule $(s_i, u/v, s_j)$.*

The systems $\Pi_1, ..., \Pi_n$ are called the components of Δ, and R contains the rules called inter-components communication rules. Each component can take some input and compute and the system accepts if all components end in a halting configuration. Each components can also communicate symbols with other components as defined by the rules in R.

DEFINITION 2. *[9] Let Δ be a dP scheme and $\delta : w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_h$ be a halting computation in Δ, where w_0 is initial configuration of Δ. Then for each $i = 0, 1, ..., h-1$:*

- *$ComN(w_i \Rightarrow w_{i+1}) = 1$ if a communication rule is used in this transition, and 0 otherwise;*
- *$ComR(w_i \Rightarrow w_{i+1}) =$ the number of communication rules used in this transition,*
- *$ComW(w_i \Rightarrow w_{i+1}) =$ the total weight of the communication rules used in this transition.*

We can also use these parameters to measure computations, results of computations, systems, and languages. We denote the set of strings accepted by Δ as $L(\Delta)$. For $ComX \in \{ComN, ComR, ComW\}$, we define

$ComX(\delta) = \sum_{i=0}^{h-1} ComX(w_i \Rightarrow w_{i+1})$, *for* δ *which is a halting computation.*

$ComX(w, \Delta) = min\{ComX(\delta) | \delta$ *is a computation of* Δ *that accepts the string* $w\}$,

$ComX(\Delta) = max\{ComX(w, \Delta) | w \in L(\Delta)\}$,

$ComX(L) = min\{ComX(\Delta) | L = L(\Delta)\}$.

Parallelizability is a measure of how a language can be efficiently computed in a distributed and parallel paradigm. [9] gives two levels of parallelizability.

DEFINITION 3. *[9] A language* $L \subseteq V^*$ *is said to be (n,m)-weakly ComX parallelizable, for some* $n \geq 2$, $m \geq 1$, *and* $X \in \{N, R, W\}$, *if there is dP automaton* Δ *with n components and there is a finite subset* F_Δ *of* L *such that each string* $x \in L - F_\Delta$ *can be written as* $x = x_1...x_n$, *with* $||x_i| - |x_j|| \leq 1$ *for all* $1 \leq i, j \leq n$, *each component* Π_i *of* Δ *takes as input the string* x_i, $1 \leq i \leq n$, *and the string* x *is accepted by* Δ *by a halting computation* δ *such that* $ComX(\delta) \leq m$. *A language* L *is said to be weakly ComX parallelizable if it is* (n, m)*-weakly ComX parallelizable for some* $n \geq 2, m \geq 1$.

DEFINITION 4. *[9] A language* $L \subseteq V^*$ *is said to be* (n, m, k)*-efficiently ComX parallelizable, for some* $n \geq 2$, $m \geq 1$, $k \geq 2$, *and* $X \in \{N, R, W\}$, *if it is (n,m)-weakly parallelizable, and there is a dP automaton* Δ *that*

$$lim_{x \in L, |x| \to \infty} \frac{time_\Pi(x)}{time_\Delta(x)} \geq k$$

for all P automata Π *such that* $L = L(\Pi)$.

We say that a language L *is efficiently parallelizable if and only if* L *is* (n, m, k)*-efficiently ComX parallelizable for any* $X \in \{N, R, W\}$, *for some* $n \geq 2, m \geq 1, k \geq 2$.

We now define the language to be considered for this study.

$$L = \{(ac)^s (bd)^s | s \geq 0\}$$

Note that L is a member of Context-Free languages. Sample members of L are: ε, *acbd*, *acacbdbd*, and *acacacacbdbdbdbd*.

3. COMMUNICATION COMPLEXITY OF 2 DP AUTOMATA

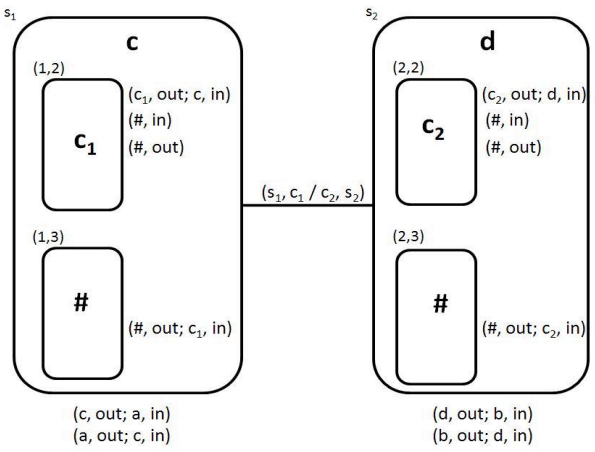

Figure 1: Recognizing $L = \{(ac)^s (bd)^s | s \geq 0\}$ **using dP Automaton with two-way communication from [9]**

By using P automata as components for the dP Scheme, we obtain a distributed version of P automata referred to as dP Automata. For this case, we consider extended P automata, where a distinguished alphabet of objects, T, is used whose elements are taken into account when building the accepted string. Other symbols coming into the system from the environment not an element of T is not considered.

DEFINITION 5. *An extended dP automaton is a construct*

$$\Delta = (\ O, E, T, \Pi_1, ..., \Pi_n, R),$$

where $(O, \Pi_1, ..., \Pi_n, R)$ *is a dP scheme,* $E, T \subseteq O$, E *is the set of objects that have arbitrarily many copies in the environment,* T *is the set of symbols that are only considered when building the accepted string,*
$\Pi_i = (O, \mu_i, w_{i,1}, ..., w_{i,k_i}, E, T, R_{i,1}, ..., R_{i,k_i})$ *is a* k_i*-membrane P automaton and the skin membrane labeled with* $(i, 1) = s_i$ *for all* $i = 1, ..., n$.

A halting computation with respect to Δ *accepts the string* $x = x_1 x_2 ... x_n$ *over* O *if the components* $\Pi_1, ..., \Pi_n$, *starting from the initial configuration, using the symport/antiport rules as well as the inter-component communication rules, in a non-deterministically maximally parallel way, bring from the environment the substrings* $x_1, ..., x_n$, *respectively, and eventually halts.*

We only consider a balanced partition of the problem, meaning given a string $x = x_1 x_2 ... x_n$, $||x_i| - |x_j|| \leq 1$. [9] also gives a constraint of having $ComX$ upper bounded by a constant.

Here we present the 2-component dP automaton that was presented in [9]

Figure 1 shows the graphical illustration of a two-way dP Automaton having $O = \{a, b, c_1, c_2, \#\}$ and $E = \{a, b, c, d\}$ and recognizing L as presented in [9]. The details of its computation is as follows:

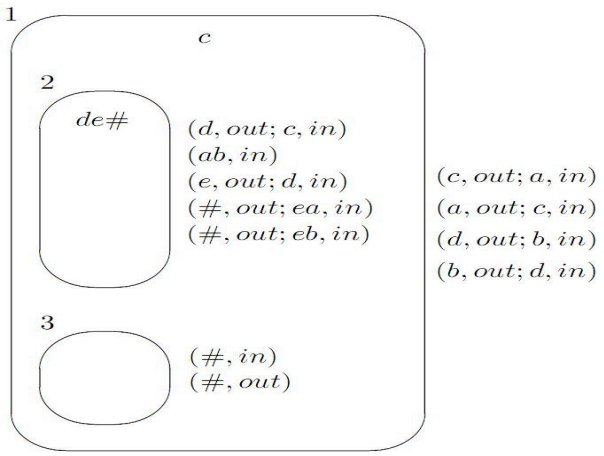

Figure 2: Recognizing $L = \{(ac)^s(bd)^s | s \geq 0\}$ using a P Automaton from [9]

The strings $(ac)^s$ and $(bd)^{s'}, s, s' \geq 0$ are generated by continuously applying rules $(c, out; a, in)$, $(a, out; c, in)$ in Π_1 and $(d, out; b, in)$, $(b, out; d, in)$ in Π_2, respectively. The generation stage stops when rules $(c_1, out; c, in)$ in membrane $(1, 2)$ and $(c_2, out; d, in)$ in $(2, 2)$ is applied. At this point, the computation will only be successful if the inter-component communicating rule $(s_1, c_1/c_2, out)$ is applicable in the next step. This indicates that the generation stage for both processors terminates at the same time, so that $s = s'$. If this does not occur, the trap object # will be moved out to the skin membrane and both processors shall oscillate forever because of the rules $(\#, in)$ and $(\#, out)$ in both membranes $(1, 2)$ and $(2, 2)$.

Figure 2 is a P automaton solving L. By Theorem 1 of [9], L is efficiently $ComX$ parallelizable, for $X \in \{N, R, W\}$. We strengthen this result by showing that L is efficiently parallelizable using higher number of components. A 3-component dP Automaton solving L is created, and then generalized to n-component dP Automaton, for $n \geq 2$.

4. COMMUNICATION COMPLEXITY OF 3 DP AUTOMATA

A 3 dP Automata is simply a dP Automata consisting of 3 P automaton components. That is,

Definition 6.

$$\Delta_3 = (O, E, T, \Pi_1, \Pi_2, \Pi_3, R),$$

where $(O, \Pi_1, \Pi_2, \Pi_3, R)$ is a dP scheme, $E \subseteq O$ is the set of all objects having arbitrarily many copy in the environment, $T \subseteq O$ is the output alphabet, and for $i = 1, 2, 3$, Π_i is a P automaton.

We also consider a balanced partition of input, that is for $x = x_1x_2x_3$, $||x_i| - |x_j|| \leq 1$, for $i \neq j$, $i, j = 1, 2, 3$. We also restrict the components to be connected similar to a linear array. That is a component can only have at most two links to other components, a left link and a right link. When components are sorted by numerical label, component i is connected only to component $(i-1)$ and component $(i+1)$. The first and last component are not linked to together.

We now show that L is parallelizable using 3 dP Automaton. We prove the following theorem for L,

Theorem 1. *Let $L = \{(ac)^s(bd)^s | s \geq 0\}$. The language L is efficiently parallelizable for 3-component dP automaton.*

We construct a 3 dP Automata accepting L and show the time and communication complexity of this dP automaton to prove the theorem.

We define our 3-component dP automaton, Δ_3 as follows:

$\Delta_3 = ($

$O = \{a, b, c, d, s, 1_{ac}, 1'_{ac}, 0_{ac}, 1_{bd}, 1'_{bd}, 0_{bd}, 0, x_a, x_b, t, f_1, f_2, f_3, \#\}$

$E = \{a, b, c, d, x_a, x_b\}$

$T = \{a, b, c, d\}$

$R = \{(s_1, f_1/f_2, s_2), (s_1, 0_{ac}/\lambda, s_2), (s_1, 1_{ac}/\lambda, s_2), (s_2, f_2/f_3, s_3), (s_2, \lambda/0_{bd}, s_3), (s_2, \lambda/1'_{bd}, s_3)\}$

$\Pi_1 = ([_{s_1}[_{(1,2)}]_{(1,2)}[_{(1,3)}]_{(1,3)}]_{s_1},$

$s_1 = (\{s\}, \{(c, out; a, in), (a, out; c, in)\}),$

$(1, 2) = (\{c, t, f_1, 0_{ac}, 1_{ac}, 0\}, \{(c, out; s, in), (0, out; s, in), (0_{ac}tf_1, out; c, in), (1_{ac}tf_1, out; a, in), (tf_1, out; 0, in), (\#, in), (\#, out)\})$

$(1, 3) = (\{\#\}, \{(t, in), (f_2, in), (t\#, out; f_1, in)\})),$

$\Pi_2 = ([_{s_2}[_{(2,2)}]_{(2,2)}[_{(2,3)}]_{(2,3)}[_{(2,4)}]_{(2,4)}]_{s_1},$

$s_2 = (\{s\}, \{(c, out; ax_a, in), (c, out; bx_b, in), (a, out; c, in), (d, out; bx_b, in), (b, out; d, in)\}),$

$(2, 2) = (\{a, c, t, f_2, f_2, 0_{ac}, 1'_{ac}, 0_{bd}, 1_{bd}, 0, x_a\}, \{(c0_{ac}, out; s, in), (a1'_{ac}x_a, out; s, in), (0, out; s, in), (0_{bd}tf_2f_2, out; d, in), (1_{bd}tf_2f_2, out; b, in), (tf_2f_2, out; 0, in), (\#, in), (\#, out)\}),$

$(2,3) = (\{\}, \{(x_a x_b, in), (1_{ac}1'_{ac}1_{bd}1'_{bd}), (0_{ac}0_{ac}0_{bd}0_{bd})\})$,

(2,4) = ({#}, {(t, in), ($f_1 f_3$, in), (t#, out; f_2, in), ($f_1 f_3$#, out; x_a, in), ($f_1 f_3$#, out; x_b, in), ($f_1 f_3$#, out; 0_{ac}, in), ($f_1 f_3$#, out; 0_{bd}, in), ($f_1 f_3$#, out; 1_{ac}, in),($f_1 f_3$#, out; $1'_{ac}$, in), ($f_1 f_3$#, out; 1_{bd}, in), ($f_1 f_3$#, out; $1'_{bd}$, in) })),

$\Pi_3 = ([_{s_3}[_{(3,2)}]_{(3,2)}[_{(3,3)}]_{(3,3)}]_{s_3}$,

$s_3 = (\{s\}, \{(d, out; b, in), (b, out; d, in)\})$,

$(3,2) = (\{b, d, t, f_3, 0_{bd}, 1'_{bd}, 0\}, \{(d0_{bd}$, out; s, in), ($b1'_{bd}$, out; s, in), (0, out; s, in), (tf_3, out; d, in), (tf_3, out; 0, in), (tf_3, out; 0, in), (#, in), (#, out)})

$(3,3) = (\{\#\}, \{(t, in), (f_2, in), (t\#, out; f_3, in)\}))$,

Each component's skin membrane would have the object s. s indicates the start of the string generation stage when it enters membrane $(i, 2)$, for $i = 1, 2, 3$. For Π_1, since it will always start the string by an 'a', then s can only be exchanged by c. For Π_2, it can start the string with 'a', or 'c' , so s enters $(2, 3)$ and either c, or a will enter s_2 respectively. For Π_3, it can start with 'b', or 'd'.

Part of the computation of Δ_3 is to alternate the a's and $c's$, and b's and d's to form $(ac)^s(bd)^s$. We refer to this part of the computation as the string input stage.

The partitioning forces the Π_2 substring to always have a balanced number of ac's and bd's. This means that $num(a)+num(c) = num(b)+num(d)$. $num(x)$ indicates the number of x characters in the string.

The objects 0_{ac}, 1_{ac}, $1'_{ac}$, 0_{bd}, 1_{bd}, $1'_{bd}$ indicates a property of the substring of the components. 0_{ac} indicates that $num(a) = num(c)$. 1_{ac} means $num(a) > num(c)$. $1'_{ac}$ is $num(a) < num(c)$. The same is similar for the other symbols, comparing $num(b)$ and $num(d)$. 0_{bd} indicates that $num(b) = num(d)$. 1_{bd} means $num(b) > num(d)$. $1'_{bd}$ is $num(b) < num(d)$. For brevity, we refer to this symbols as form-count symbols. We pass these objects towards Π_2 using the intercomponent communication rules in R.

For Δ_3 to recognize L, all the following conditions must be met:

1. Form-count objects does not only indicate count but also implicitly tells the start and end character of the input string of a component. Therefore, there exists a finite combination of form-count objects that must be followed. There are only two possible combinations that would result to valid form and equal number of pairs of ac's and bd's. This is either $1_{ac}1'_{ac}1_{bd}1'_{bd}$ or $0_{ac}0_{ac}0_{bd}0_{bd}$. These objects would enter $(2, 3)$ only if it follows one of the two combinations. If it fails to enter $(2, 3)$, when the objects f_1 and f_3 enter $(2, 4)$, indicating the end of the string input stage, then it would exchange one of these symbols with the trap symbol # to cause a non-halting computation.
2. There must be an equal number of ac's and bd's in Π_2. The rules in s_2 takes in x_a's and x_b's from the environment. It should be that the number of x_a's is equal to the number of x_b's. Otherwise, when the string input stage ends, $(2, 4)$, would consume an x_a or x_b and release the trap symbol.

We have now constructed our 3-component dP automaton Δ_3 recognizing L. The following lemma is a result from taking the time and communication complexity of Δ_3.

LEMMA 1. *The language $L = \{(ac)^s(bd)^s | s \geq 0\}$ is $(3, m, k)$-efficiently ComX parallelizable, for $(m, X) \in \{(1, N), (4, R), (6, W)\}$, and for k, $2 \leq k \leq 3$.*

The running time of Δ_3 for any accepting computation is $\frac{4s}{3} + 3$. 1 step to prepare the string input stage, 1 step to communicate, 1 step to check the combinations, and $\frac{4s}{3}$ steps for the string input stage.

The communication measure is as follows: $ComN(\Delta_3) = 1$, $ComR(\Delta_3) = 4$, and $ComW(\Delta_3) = 6$. The communication step happens only once at the end of the string input stage. Components Π_1 and Π_3 will use the communication rule to send f_1 and f_3 to component Π_2. At the same time, Π_1 and Π_3 will use exactly one communication rule to communicate a form-count symbol, so the total number of communcication rules used is 4. The objects communicated are f_1, f_2, f_2, f_3 and two form-count objects for a total of 6.

We measure Δ_3's running time against a single component P automaton recognizing the same language. By the definition, it is efficiently parallelizable if:

$$lim_{x \in L, |x| \to \infty} \frac{time_\Pi(x)}{time_\Delta(x)} \geq k$$

where $k \geq 2$ is some constant. The P automaton that recognizes L in [9] runs in $4s + 2$ steps. Plugging in the values and taking the limit, we get that $lim_{s \to \infty} \frac{time_\Pi(x)}{time_{\Delta_3}(x)} = 3$. Therefore our language $L = \{(ac)^s(bd)^s | s \geq 0\}$ is (3,m,k)-efficiently ComX parallelizable, $k \leq 3$.

By Lemma 1 and Definition 4, we have proven Theorem 1.

5. N DP AUTOMATA

Now, we generalize the construction of an n-component dP automaton recognizing L, $n \geq 3$. We can partition the string $w \in L$ such that half of the components generate the ac pairs and the other half generates the bd pairs. If n is odd, then one component will have to generate both pairs. Because of this, there are some differences in the construction of the dP automaton when n is odd or even.

First we introduce another checking phase we would refer to as 'start-end check'. The start-end check is used to ensure that adjacent component substrings form a correct sequence when concatenations between the substrings is done. Hence, we compare the start and end character of each adjacent component string.

A string $w \in L$ is of the form $(ac)^s(bd)^s$. We let w be partitioned into n substrings of almost equal length, $w = w_1...w_n$, $||w_i| - |w_j|| \leq 1$, for $1 \leq i, j, \leq n$. Let w_i and w_{i+1} be substrings of w, for $1 \leq i \leq n$. Let x be the end character of w_i and y be the start character of w_{i+1}. What we mean by the start character is the first symbol that enters the skin membrane of a component and the end character is the last symbol that enters the skin membrane from the environment. Since we are using extended P automata, $x, y \in T = \{a, b, c, d\}$.

We say that w_i and w_{i+1} passes the start-end check if it satisfies one of the following conditions:

- $x = a$ and $y = c$
- $x = c$ and $y = a$
- $x = c$ and $y = b$
- $x = b$ and $y = d$
- $x = d$ and $y = b$

The start-end check is done between adjacent components. The checking can be done using the intercomponent communication rules. Adjacent components would use symbols to the indicate the start and end characters of their string. Since there are n components, there would be $n-1$ adjacent pairs, Π_1 and Π_2, Π_2 and Π_3, ..., Π_{n-1} and Π_n. We define Π_1 and Π_2 as the 1st adjacent pair and Π_{n-1} and Π_n as the $(n-1)$th adjacent pair.

We denote the start and end symbols as follows: $r_i^{(j)}$, where $r \in \{a, b, c, d\}$, $i = 1...n$, $j = 1...n-1$. r indicates which character a component starts or ends with. i indicates which component it came from. j indicates which adjacent pair $r_i^{(j)}$ belongs to. If $i = j$, then $r_i^{(j)}$ is the end character of the Π_i string. If $j = i-1$, then $r_i^{(j)}$ is the start character of the Π_i string. For brevity, we refer to these objects as 'start-end' objects.

We modify the antiport rules that we used to start and end the string input stage. The string input stage starts when s enters the $(i, 2)$ membrane of the component Π_i, for $1 \leq i \leq n$, exchanging it with a, c, b or d, with some other objects. We include in this rule the objects $r_i^{(j)}$. Taking for example $(c, out; s, in)$, we modify this to $(ca_i^{(j)}, out; s, in)$. We paired c with $a_i^{(j)}$ because when we start the string input stage, if c is the object present at the skin, the first object to enter the component is a, thus the component string starts with an 'a'.

For the antiport rule that ends the string input stage, we do the same. Take for instance the Π_1 rule $(0_{ac}tf_1, out; c, in)$ of the 3-component dP automaton in section 4. We know that Π_1 always starts with 'a', so we know that when c is the last to enter the component, the component substring ended with 'c'. Therefore, we modify the rule to $(0_{ac}c_1^{(1)}tf_1, out; c, in)$.

We form our intercomponent communication rules based on the given conditions above. If $(s_i, r_i^j/r_{i+1}^{'j}, s_{i+1})$ is in R, then the jth pair has valid start and end characters so we can apply this communication rule. We list the following valid pairs of r and r', denoted by (r, r'): (a, c), (c, a), (c, b), (b, d), (d, b). We refer to $(s_i, r_i^j/r_{i+1}^{'j}, s_{i+1})$ as a comparison communication rule. Note that all pairs should use these rules. If not, then it means there is at least one pair that have an invalid start and end causing the trap symbol # to be released triggering an non-halting computation.

We also take into account that for $n > 3$, the start and end characters are not predetermined for other components in between the first component, Π_1, and the last component, Π_n. We modify some antiport rules in $(i, 2)$, for $1 < i < n$, for it to release the correct form-count objects.

We double each start-end object that indicates the start character of the component substring. When we use an rule to place s within the second membrane, we exchange the two start-end objects. One would be used later for the start-end check of the adjacent components, and the other will be used to acquire the correct form-count object. An example for the rule modification is: $(ca_i^{(j)}, out; s, in)$ to $(ca_i^{(j)}a_i^{(j)}, out; s, in)$.

Similar to the previous section, our form-count set would consist of:

$$FC = \{1_{ac}, 1'_{ac}, 0_ac, 1_{bd}, 1'_{bd}, 0_{bd}\}$$

At the end of the string input stage, there should be an object $x \in T$ on the skin membrane and the two start-end objects. For the components Π_i, $i = 1 + 1...n-1$, the membrane $(i, 2)$ would use the following rule to release the correct form-count object:

$$(zx_i^{(j+1)}tf_i, out; xy_i^{(j)}, in)$$

where $j = i-1$, $z \in FC$, $x, y \in T'$, $x_i^{(j+1)}, y_i^{(j)}$ are start-end objects. Note that $y_i^{(j)}$ indicates the start character of the Π_i substring and $x_i^{(j+1)}$ is the end character. T' is a subset of T. For components Π_1 to $\Pi_{\frac{n}{2}}$, $T' = \{a, c\}$. For the other half, $T' = \{b, d\}$. If n is odd, the middle component would have $T' = T$.

Components Π_1 and Π_n uses another rule since Π_1 does

not need to check for its start character and Π_n for its end character with another component. We use the following rules for Π_1 and Π_n respectively for releasing the correct form-count symbol.

- $(zx_1^{(1)}tf_1, out; x, in)$ for Π_1.
- $(ztf_n, out; xy_n^{(n-1)}, in)$

z depends on x and y. If $x \neq y$ then it means that the component started and ended with a different character. Therefore there is an equal number of a's with c's or b's with d's then z is 0_{ac} or 0_{bd}. If $x = y$, then we would have the following: $x = a, 1_{ac}$, $x = b, 1_{bd}$, $x = c, 1'_{ac}$, and $x = d, 1'_{bd}$.

5.1 Algorithm for n is even

We now discuss how the dP automaton will run. We start with even numbered component dP automaton. We separate the case because we handle odd numbered components differently because of the existence of the middle component.

Let $w \in L$, $w = w_1w_2...w_n$. We partition w so that Π_1 to $\Pi_{\frac{n}{2}}$ would generate the pairs of ac's and $\Pi_{\frac{n}{2}+1}$ to Π_n would generate the pairs of bd's, and $||w_i| - |w_j|| \leq 1$, for $1 \leq i, j \leq n$.

We connect the components similar to a linear array. The intercomponent communication rules would only have the following form, $(s_i, x/y, s_{i+1})$ for $1 \leq i \leq n-1$, $x, y \in O^*$. With these, we can use our start-end check algorithm.

We choose $\Pi_{\frac{n}{2}}$ as the 'central' component of the ac components and $\Pi_{\frac{n}{2}+1}$ as the 'central' component of the bd components. The central components are responsible for checking if there is an equal number of pairs of ac's and pairs of bd's.

Each non-central component would pass their form-count objects towards the central components. For Π_i, $1 \leq i \leq \frac{n}{2}$, the components pass the form-count objects $\{0_{ac}, 1_{ac}, 1'_{ac}\}$ towards $\Pi_{\frac{n}{2}}$. For Π_i, $\frac{n}{2}+1 \leq i \leq n$, they pass the form-count objects $\{0_{bd}, 1_{bd}, 1'_{bd}\}$ towards $\Pi_{\frac{n}{2}+1}$. We do this by adding the following intercomponent communication rules to R:

For Π_i, $1 \leq i \leq \frac{n}{2} - 1$

- $(s_i, 0_{ac}/\lambda, s_{i+1})$
- $(s_i, 1_{ac}/\lambda, s_{i+1})$
- $(s_i, 1'_{ac}/\lambda, s_{i+1})$

For Π_i, $\frac{n}{2}+1 \leq i \leq n-1$

- $(s_i, \lambda/0_{bd}, s_{i+1})$
- $(s_i, \lambda/1_{bd}, s_{i+1})$
- $(s_i, \lambda/1'_{bd}, s_{i+1})$

The central components would exchange their collected form-count objects from each side. If all ac form-count objects are given to $\Pi_{\frac{n}{2}+1}$ and all bd form-count objects are given to $\Pi_{\frac{n}{2}}$, then it is a successful and accepting computation.

Note that the length of w, $4s$, is always even. Therefore, if take $q = 4s \ mod \ n$, for an even n, then the result would always be even. q indicates the number of components that would take one more step to finish the string input stage, meaning these components would have strings of length greater than the others by one symbol.

Since q is even, then we can divide it by two and we can restrict the dP automaton so that half of q components would generate ac's and the other half of q would generate bd's. We use this to form the intercomponent communication rules between the two central components. This the rules simpler but other valid partitioning of w may not be considered.

For the exchange between $\Pi_{\frac{n}{2}}$ and $\Pi_{\frac{n}{2}+1}$, we add the following rules to R:

- $(s_{\frac{n}{2}}, 1_{ac}1'_{ac}/1_{bd}1'_{bd}, s_{\frac{n}{2}+1})$
- $(s_{\frac{n}{2}}, 0_{ac}/0_{bd}, s_{\frac{n}{2}+1})$

The rules are formed so that it satisfies two conditions: (1) $num(a) = num(c)$ and $num(b) = num(d)$. For every 1_{xy}, there should be an $1'_{xy}$. If this is not the case, not only will it fail in these exchange between the central components, it will also not pass the start-end check. (2) $num(ac) = num(bd)$. Because 1_{ac} and $1'_{ac}$ would make an extra pair of ac, we compare these to the extra pairs produced by 1_{bd} and $1'_{bd}$.

When the central components fail to exchange all the form-count objects, we should make it so that the membrane containing the trap symbol # consumes the form-count ojbects and release #. But since the form-count object from the first and last component would be the last to enter the the central component, we can add an object on both ends so that the central component's # membrane would only do this when the form-count objects from both ends arrive.

Let e_1 and e_n be the objects to indicate that the form-count objects from Π_1 and Π_n has already reached the central components. All components would also pass these towards the central component. The central components would then exchange these objects and use it as a trigger to consume any remaining unexchanged form-count objects. We add the following rules to R:

- $(s_i, e_1/\lambda, s_{i+1})$, if Π_i generates ac pairs
- $(s_i, \lambda/e_n, s_{i+1})$, if Π_i generates bd pairs
- $(s_{\frac{n}{2}}, e_1/e_n, s_{\frac{n}{2}+1})$

We add the rules for the # membrane of the central components:

For $\Pi_{\frac{n}{2}}, (\frac{n}{2}, 3)$

- (e_n, in)
- $(z, in; \#e_n, out)$, $z \in \{0_{ac}, 1_{ac}, 1'_{ac}\}$

For $\Pi_{\frac{n}{2}+1}, (\frac{n}{2}+1, 3)$

- (e_1, in)
- $(z, in; \#e_1, out)$, $z \in \{0_{bd}, 1_{bd}, 1'_{bd}\}$

5.2 Algorithm for n is odd

We have constructed the rules for n-components for n is even. For n is odd, the middle component would make the one assumption result into a false accepting computation. This assumption is when we formed our form-count objects. Taking into example the ac components. 1_{ac} means that there is an extra a in a component substring, and $1'_{ac}$ is for an extra c. But since the middle component would have to generate both ac and bd pairs, the length difference of the ac substrings of the non-middle component with the ac substring of the middle component would be more than one, hence comparison of the equality of the number of ac's and bd's would become difficult.

If we let the middle component generate only one set of pairs, either ac's or bd's only, to make sure that each component have an equal number of pairs, all components would have to use a counting object representing the number ac pairs and bd pairs. These would result into a $ComX$ as a linear function of s. So for an odd numbered component, we also force the middle component to generate only even length, similar to the case in the 3-component dP automaton in section 4.

We connect the components to form a linear array. The intercomponent communication rules would only have the following form, $(s_i, x/y, s_{i+1})$ for $1 \leq i \leq n-1$, $x, y \in O^*$.

Let $w \in L$, $w = w_1 w_2 ... w_n$. We partition w such that Π_i, for $1 \leq i \leq \lceil \frac{n}{2} \rceil - 1$ would generate only ac pairs. $\Pi_{\lceil \frac{n}{2} \rceil}$ would generate ac and bd. Π_i, for $\lceil \frac{n}{2} \rceil + 1 \leq i \leq n$, generates the bd pairs.

We restrict the partition so that the middle component, $\Pi_{\lceil \frac{n}{2} \rceil}$ or simply Π_{mid}, would always generate an even length string. In section 5.1, q indicates the number of components that generates an extra symbol. We make it so that when q is odd, Π_{mid} is included in those components generating an extra symbol. The half of the $q-1$ components would generate ac's and the other half will be bd's. If q is even, Π_{mid} would not be included in those q components.

We have to satisfy two conditions to consider an accepting computations, similar to section 4. One is that there should be that $num(a) + num(c) = num(b) + num(d)$. The second is similar to the even n-component dP automaton that we can pair off the correct combination of form-count objects. This is all done in the middle component. We assign Π_{mid} as the central component, where unlike in an even numbered component dP automaton, we have two. The conditions are enough for an n-component dP automaton to recognize L, n is odd, but other valid partitions of w are not considered as correct computations.

The construct of Π_{mid} is similar to Π_2 in section 4. Π_{mid} has four components. $(mid, 4)$ is the # membrane. $(mid, 3)$ is used for collecting the proper combination of form-count objects.

To ensure that Π_{mid} generates an even length string containing an equal number of ac's and bd's, during the string input stage, whenever an a enters the skin membrane, an object x_a representing the number of ac's enters the skin membrane at the same time. When a b enters the skin membrane, an object x_b representing the number of bd's enters the skin membrane at the same time.

If w_{mid}, the Π_{mid} substring, starts with 'c', we add an extra x_a and it should end with a 'b'. If it started with an 'a', it should end with 'd' but we do not add an extra x_a or x_b. If it starts and ends in other combinations, it would result into an odd length string or an uneven number of x_a's and x_b's.

$(mid, 3)$ uses the antiport rule, $(x_a x_b, in)$ to consume the x_a's and x_b's from the skin membrane. If after the string input stage there still exists at least one x_a or x_b, then $(mid, 4)$ would consume that symbol and release the trap object #, causing a non-halting computation.

The next is the checking of the combination of form-count objects. Similar to section 5.1, we pass on towards the central component all form-count objects. At the same time, the Π_1 and Π_n also passes e_1 and e_n. We add the following rules to R:

For Π_i, $1 \leq i \leq \lceil \frac{n}{2} \rceil - 1$

- $(s_i, 0_{ac}/\lambda, s_{i+1})$
- $(s_i, 1_{ac}/\lambda, s_{i+1})$
- $(s_i, 1'_{ac}/\lambda, s_{i+1})$
- $(s_i, e_1/\lambda, s_{i+1})$

For Π_i, $\lceil \frac{n}{2} \rceil \leq i \leq n-1$

- $(s_i, \lambda/0_{bd}, s_{i+1})$
- $(s_i, \lambda/1_{bd}, s_{i+1})$
- $(s_i, \lambda/1'_{bd}, s_{i+1})$

- $(s_i, \lambda/e_n, s_{i+1})$

$(mid, 3)$ would use the following antiport rules to check correct form-count combinations:

- $(1_{ac}1'_{ac}1_{bd}1'_{bd}, in)$
- $(0_{ac}0_{bd})$

Notice the similarity between these antiport rules and the intercomponent communication rules used by the central components in section 5.1. We also give the same reason for these combinations.

We add the following antiport rules to $(mid, 4)$ so that it can go to a non-halting computation if one of the two conditions are not satisfied.

- $(f_{mid-1}f_{mid+1}, in)$, where Π_{mid-1} and Π_{mid+1} are the components on the left and right of Π_{mid} respectively.
- (e_1e_n, in)
- $(x_a, in; \#f_{mid-1}f_{mid+1})$, and $(x_b, in; \#f_{mid-1}f_{mid+1})$
- $(X, in; \#e_1e_n, out)$, $X \in \{0_{ac}, 1_{ac}, 1'_{ac}, 0_{bd}, 1_{bd}, 1'_{bd}\}$

We now have a construction of an n-component dP automaton. All that is left is to show the time and communication complexity of the system.

LEMMA 2. *Let $L = \{(ac)^s(bd)^s | s \geq 0\}$. The language L is (n, m)-weakly $ComX$ parallelizable for all $n \geq 2$, $(m, ComX) \in \{(O(n^2), ComN), (O(n^2), ComR), (O(n^2), ComW)\}$*

Proof:
Let $w \in L$, $w = w_1w_2...w_n$. Let Δ_n be an n-component dP automaton, $n \geq 2$. Δ_n's components are labeled as $\Pi_1, \Pi_2, ..., \Pi_n$, and let Π_{mid} be the central component, $mid = \lceil \frac{n}{2} \rceil$.

From the construction of Δ_n, for a component Π_i, if $1 \leq i < mid$, then form-count objects produced in Π_i are communicated to component $i + 1$ and propagated towards Π_{mid}. If $mid < i \leq n$, the form-count objects in Π_i are communicated to component $i - 1$ and propagated towards Π_{mid}. Since, the central component is in the middle, a form-count object will pass at most $n/2$ components. Specifically, a form-count object originating from component i will pass through $|i - mid|$ components. n components would produce n form-count objects.

The total number of communication needed for all form-count objects to reach the central component is,

$$2(1 + 2 + ... + \frac{n}{2}) = 2\sum_{i}^{\frac{n}{2}} i$$

.

So, $ComN = 2\sum_i^{\frac{n}{2}} i = \frac{n^2}{4} + \frac{n}{2}$. $ComR$ and $ComW$ is only a factor plus some constants of $ComN$. Using Big-O notation $ComX = O(n^2)$, $ComX \in \{ComN, ComR, ComW\}$. Therefore, L is (n, m)-weakly $ComX$ parallelizable, $m \in O(n^2)$, for all $ComX \in \{ComN, ComR, ComW\}$. Note that n, the number of components is not a factor of the length of the input $4s$ and therefore can be considered constant with respect to s. Q.E.D.

LEMMA 3. *Given $L = \{(ac)^s(bd)^s | s \geq 0\}$, an n-component dP Automaton Δ_n, and a P automaton Π, such that $L(\Delta_n) = L(\Pi) = L$. Then,*

$$lim_{x \in L, |x| \to \infty} \frac{Time_P(x)}{Time_{\Delta_n}(x)} = n$$

Proof:
Notice that from the construction of our n-component in section 5.1 and 5.2, the running time is mainly dependent on the string input stage. Since we partition w in an almost balanced partition, the difference of the string input stage of each component would only be by one computation step. The string input stage would last for $\frac{4s}{n}$ steps.

For the central components to complete the checking of the form-count object combinations, it would take at most $\frac{n}{2}$ steps for e_1 and e_n to arrive at the central components. Therefore, the total running time of Δ_n is $\frac{4s}{n} + \frac{n}{2}$.

Taking the limit of the ratio between the running time of a single component P automaton Π and the running time of Δ_n:

$$lim_{w \in L, |w| \to \infty} \frac{4s + 2}{\frac{4s}{n} + \frac{n}{2}} = n$$

This means that Δ_n is n times faster than Π. Q.E.D.

Using Lemma 2 and Lemma 3, we can prove the following theorem.

THEOREM 2. *Let $L = \{(ac)^s(bd)^s | s \geq 0\}$. The language L is (n, m, k)-efficiently $ComX$ parallelizable for all n-component dP automaton Δ_n, $L(\Delta_n) = L$, $n \geq 2$, $m \in O(n^2)$, $ComX \in \{ComN, ComR, ComW\}$, $k \leq n$.*

By Lemma 2, we know that L is (n, m)-weakly $ComX$ parallelizable for $n \geq 2$, and $m \in O(n^2)$ for all $ComX \in$

$\{ComN, ComR, ComW\}$. By Lemma 3, we know that the speed up ratio is n. Since the condition for k is $k \geq 2$, then any value of $k \leq n$ can satisfy the limit of ratio speed up inequality. Therefore, the language L is (n, m, k)-efficiently $ComX$ parallelizable for all n-component dP automaton Δ_n, $L(\Delta_n) = L$, $n \geq 2$, $m \in O(n^2)$, $ComX \in \{ComN, ComR, ComW\}$, $k \leq n$. Q.E.D.

Using Definition 4, we arrive at the following Corollary,

COROLLARY 1. *L is efficiently parallelizable for all n-component dP automaton, $n \geq 2$.*

6. CONCLUSION

We have created an algorithm to construct an n-dP Automata recognizing $L = \{(ac)^s(bd)^s | s \geq 0\}$. One trivial assumption we have with regards to the input length is that $4s \geq n$, so that all components would be used, else a loss in efficiently. The following table is a comparison of the time and communication complexity of the 2-dP automaton, 3-dP automaton, and n-dP automaton, $n \geq 2$.

dPA model	ComN	ComR	ComW	$\frac{time_\Pi}{time_\Delta}$
2-dPA	1	1	2	2
3-dPA	1	4	6	3
n-dPA	$O(n^2)$	$O(n^2)$	$O(n^2)$	n

Table 1: Results for dP Automata Accepting $L = \{(ac)^s(bd)^s | s \geq 0\}$

Based on these results, there are languages such as L that is efficiently parallelizable for all $n \geq 2$. This implies that there are such problems that the number of components or processors of distributed parallel models can be scaled up indefinitely without suffering performance loss. We define such class of languages as Embarrassingly Parallelizable languages or problems. The definition of an embarrassingly parallel problem is that its input set can be distributed into multiple processors with little to no effort. One future research of interest is to study such class of languages for dP Automata, and dP Systems in general.

Acknowledgement

We would like to thank the following groups/organizations:
ERDT Scholarship Program
Algorithms and Complexity Lab, Department of Computer Science, College of Engineering, University of the Philippines Diliman.

7. REFERENCES

[1] Adorna, Henry. Păun, Gheorghe. Pérez-Jimeńez, Mario J.; On communication complexity in evolution-communication P Systems.; Proc. 8th Brainstorming Week on Membrane Computing, Sevilla, February 2010, 1-22, and Romanian J. Inform. Sci. and Technology, 2010, in press.

[2] Ciobanu, Gabriel. Păun, Gheorghe. Pérez-Jimeńez, Mario J.; Applications of Membrane Computing.; Springer-Verlag Berlin Heidelberg 2006.

[3] Csuhaj-Varjú, E.; P Automata.; In: Mauri, G., Păun, G., Pérez-Jiménez, Rozenberg, G., Salomaa, A. (eds), Membrane Computing: 5th International Workshop, WMC 2004, Milan, Italy, June, 14-16, 2004. Revised Selected and Invited Papers. Lecture Notes in Computer Science 3365, Springer 2005, 19-35.

[4] Csuhaj-Varjú, E.; P Automata: Concept, Results, and New Aspects. Workshop on Membrane Computing 2009: 1-15

[5] Csuhaj-Varjú, Erzsébet. Kelemen, Jozef. Kelemenová, Alica. Păun, Gheorghe. Vaszil, György.; Computing with Cells in Environment: P Colonies.; Technical Report 2005/1, Theoretical Computer Science Research Group, MTA SZTAKI, Budapest, 2005.

[6] Csuhaj-Varjú, Erzsébet. Vaszil, György. Păun, Gheorghe.; Grammar System versus Membrane Computing: The Case of CD Grammar Systems. Fundam. Inform. 76(3): 271-292 (2007).

[7] Csuhaj-Varjú, Erzsébet. Păun, Gheorghe. Vaszil, György.; Grammar System versus Membrane Computing: The Case of PC Grammar Systems. Fundamenta Informaticae - SPECIAL ISSUE ON DEVELOPMENTS IN GRAMMAR SYSTEMS Volume 76 Issue 3, March 2007.

[8] Freund, Rudolf. Kogler, Marian. Păun, Gheorghe. Pérez-Jiménez, Mario.; On the Power of P and dP Automata. Analele Universitatii Din Bucuresti (Seria Matematica-Informatica). 2009. Pag. 5-22.

[9] Păun, G., Pérez-Jiménez, M.J.; Solving Problems in a Distributed Way in Membrane Computing: dP Systems. International Journal of Computers Communications and Control, ISSN 1841-9836, 5(2):238-250, 2010.

[10] Leighton, F.T.; Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes; Morgan Kaufmann Publishers. September 1991.

[11] Hromkovič, J.; Communication Complexity and Parallel Computing; Springer. April 1997.

[12] The P Systems Website: www.ppage.psystems.eu.