Enzymatic Numerical P Systems Using Elementary Arithmetic Operations

Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, and Claudio Zandron

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
{leporati,mauri,porreca,zandron}@disco.unimib.it

Abstract. We prove that all-parallel enzymatic numerical P systems whose production functions can be expressed as a combination of sums, differences, products and integer divisions characterise **PSPACE** when working in polynomial time. We also show that, when only sums and differences are available, exactly the problems in **P** can be solved in polynomial time. These results are proved by showing how EN P systems and random access machines, running in polynomial time and using the same basic operations, can simulate each other efficiently.

1 Introduction

Numerical P systems have been introduced in [8] as a model of membrane systems inspired both from the structure of living cells and from economics. Each region of a numerical P system contains some numerical variables, that evolve from initial values by means of *programs*. Each program consists of a *production function* and a *repartition protocol*; the production function computes an output value from the values of some variables occurring in the same region in which the function is located, while the repartition protocol distributes this output value among the variables in the same region as well as in the neighbouring (parent and children) ones.

In [8], and also in Chapter 23.6 of [9], some results concerning the computational power of numerical P systems are reported. In particular, it is proved that nondeterministic numerical P systems with polynomial production functions characterize the recursively enumerable sets of natural numbers, while deterministic numerical P systems, with polynomial production functions having non-negative coefficients, compute strictly more than semilinear sets of natural numbers.

Enzymatic Numerical P systems (EN P systems, for short) have been introduced in [10] as an extension of numerical P systems in which some variables, named the *enzymes*, control the application of the rules, similarly to what happens in P systems with promoters and inhibitors [2]. As shown in [11, 3] and references therein, the most promising application of EN P systems seems to be the simulation of control mechanisms of mobile and autonomous robots.

The computational power of EN P systems has also been thoroughly investigated. In [6] a short review of previously known universality results is presented, together with an improvement on some of them: linear production functions involving only one variable suffice to obtain universality in the one-parallel and all-parallel modes.

In this paper we deal with computational complexity issues, and show how the choice of arithmetical operations allowed in the production functions influences the efficiency of computation of all-parallel EN P systems, exactly as it happens for random access machines [5]. Indeed, we prove that these two computation devices can simulate each other efficiently in some relevant cases. As a consequence, we show the limitations of linear production functions, and how these are overcome by allowing multiplication and integer division, leading to polynomial time solutions to **PSPACE**-complete problems.

The paper is organised as follows. In Section 2 we recall the definitions of EN P systems and random access machines, together with the relevant results from the literature. In Section 3 we show, as a technical result, how indirect addressing can be eliminated when RAMs operate in polynomial time, thus simplifying the simulation by means of all-parallel EN P systems that is presented in Section 4. The converse simulation is illustrated in Section 5, leading to our main result about the computational complexity of all-parallel EN P systems. Finally, conclusions and open problems are described in Section 6.

2 Definitions and Previous Results

An *enzymatic numerical P system* (EN P system, for short) is a construct of the form:

$$\Pi = \big(m, H, \mu, (\mathit{Var}_1, \mathit{Pr}_1, \mathit{Var}_1(0)), \ldots, (\mathit{Var}_m, \mathit{Pr}_m, \mathit{Var}_m(0))\big)$$

where $m \geq 1$ is the degree of the system (the number of membranes), H is an alphabet of labels, μ is a tree-like membrane structure with m membranes injectively labeled with elements of H, Var_i and Pr_i are respectively the set of variables and the set of programs that reside in region i, and $\mathit{Var}_i(0)$ is the vector of initial values for the variables of Var_i. All sets Var_i and Pr_i are finite. In the original definition of EN P systems [10] the values assumed by the variables may be real, rational or integer numbers; in what follows we will allow instead only integer numbers. The variables from Var_i are written in the form $x_{j,i}$, for j running from 1 to $|\mathit{Var}_i|$, the cardinality of Var_i; the value assumed by $x_{j,i}$ at time $t \in \mathbb{N}$ is denoted by $x_{j,i}(t)$. Similarly, the programs from Pr_i are written in the form $P_{l,i}$, for l running from 1 to $|\mathit{Pr}_i|$.

The *programs* allow the system to evolve the values of variables during computations. Each program is composed of two parts: a *production function* and a *repartition protocol*. The former can be any function using variables from the region that contains the program. Using the production function, the system computes a *production value*, from the values of its variables at that time. This value is distributed to variables from the region where the program resides, and to

variables in its upper (parent) and lower (children) compartments, as specified by the repartition protocol. Formally, for a given region i, let $v_1, \ldots, v_{n_i}$ be all these variables; let $x_{1,i}, \ldots, x_{k_i,i}$ be some variables from Var_i, let $F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})$ be the production function of a given program $P_{l,i} \in Pr_i$, and let $c_{l,1}, \ldots, c_{l,n_i}$ be natural numbers. The program $P_{l,i}$ is written in the following form:

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}) \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i} \quad (1)$$

where the arrow separates the production function from the repartition protocol. Let $C_{l,i} = \sum_{s=1}^{n_i} c_{l,s}$ be the sum of all the coefficients that occur in the repartition protocol. If the system applies program $P_{l,i}$ at time $t \geq 0$, it computes the value

$$q = \frac{F_{l,i}(x_{1,i}(t), \ldots, x_{k_i,i}(t))}{C_{l,i}}$$

that represents the "unitary portion" to be distributed to variables $v_1, \ldots, v_{n_i}$ proportionally with coefficients $c_{l,1}, \ldots, c_{l,n_i}$. So each of the variables v_s, for $1 \leq s \leq n_i$, will receive the amount $q \cdot c_{l,s}$. An important observation is that variables $x_{1,i}, \ldots, x_{k_i,i}$ involved in the production function are reset to zero after computing the production value, while the other variables from Var_i retain their value. The quantities assigned to each variable from the repartition protocol are added to the current value of these variables, starting with 0 for the variables which were reset by a production function. As pointed out in [12], a delicate problem concerns the issue whether the production value is divisible by the total sum of coefficients $C_{l,i}$. As it is done in [12], in this paper we assume that this is the case, and we deal only with such systems; see [8] for other possible approaches.

Besides programs (1), EN P systems may also have programs of the form

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})|_{e_{j,i}} \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i}$$

where $e_{j,i}$ is a variable from Var_i different from $x_{1,i}, \ldots, x_{k_i,i}$ and $v_1, \ldots, v_{n_i}$. Such a program can be applied at time t only if $e_{j,i}(t) > \min(x_{1,i}(t), \ldots, x_{k_i,i}(t))$. Stated otherwise, variable $e_{j,i}$ operates like an *enzyme*, that enables the execution of the program but, as happens with catalysts, it is neither consumed nor modified by the execution of the program. However, in EN P systems enzymes can evolve by means of other programs, that is, enzymes can receive "contributions" from other programs and regions.

A *configuration* of Π at time $t \in \mathbb{N}$ is given by the values of all the variables of Π at that time; in a compact notation, we can write it as the sequence $(Var_1(t), \ldots, Var_m(t))$, where m is the degree of Π. The *initial configuration* can thus be described as the sequence $(Var_1(0), \ldots, Var_m(0))$. The system Π evolves from an initial configuration to other configurations by means of *computation steps*, in which one or more programs of Π (depending upon the *mode* of computation) are executed. In [12], at each computation step the programs to be executed are chosen in the so called *sequential* mode: one program is nondeterministically chosen in each region, among the programs that can be executed at

that time. Another possibility is to select the programs in the so called *all-parallel* mode: in each region, all the programs that can be executed are selected, with each variable participating in all programs where it appears. Note that in this case EN P systems become *deterministic*, since nondeterministic choices between programs never occur. A variant of parallelism, analogous to the maximal one which is often used in membrane computing, is the so called *one-parallel* mode: in each region, all the programs which can be executed can be selected, but the actual selection is made in such a way that each variable participates in only one of the chosen programs. We say that the system reaches a *final configuration* if and when it happens that no applicable set of programs produces a change in the current configuration.

EN P systems may be used as (polynomial) time-bounded recognising devices as follows. Notice that we use two variables (instead of just one of them), named *accept* and *reject*, to signal the end of computations. This is done because some programs of the system may be applied forever, causing the system to never halt even if the configuration does not change any more. By using two variables, the event of reaching a final configuration is made visible and distinguishable from the outside.

Definition 1. *Let $L \subseteq \{0,1\}^\star$ be a language, and let Π be a deterministic EN P system with two distinguished variables accept and reject. We say that Π decides L in polynomial time iff, for all $x \in \{0,1\}^\star$, when the integer having binary representation $1x$ is initially given to a specified input variable*[1] *the P system Π reaches a final configuration such that*

- *if $x \in L$, then accept $= 1$ and reject $= 0$*
- *if $x \notin L$, then accept $= 0$ and reject $= 1$*

within a number of steps bounded by $O(|x|^k)$ for some $k \in \mathbb{N}$.

As proved in [6], every all-parallel and one-parallel EN P system can be "flattened" into an equivalent (both in terms of output and number of computation steps) system having only one membrane. For simplicity, in the following sections we shall always deal with flattened EN P systems.

The proofs in this paper will be based on random access machines [7, 5]. We define the specific variant we will employ:

Definition 2 (RAM). *A random access machine consists of an infinite number of* registers *$(r_i : i \in \mathbb{N})$ having values in $\mathbb{N}$, initially set to zero, and a finite sequence of* instructions *injectively labelled by elements $\ell \in \mathbb{N}$. The instructions are of the following types:*

- *assignment of a constant $k \in \mathbb{N}$: "ℓ: $r_i := k$" (r_i is assigned a constant value)*
- *copying a register: "ℓ: $r_i := r_j$" (r_i is assigned the content of a fixed register)*
- *indirect addressing: "ℓ: $r_i := r_{r_j}$" (r_i is assigned the content of a register whose number is given by a fixed register)*

[1] The "1" is prefixed to the input string x in order to keep the leading zeroes.

– *arithmetic operations, with* $\bullet \in \{+, -, \times, \div\}$*: "*$\ell$: $r_i \coloneqq r_j \bullet r_k$*"*
– *conditional jump, with* $\ell_1, \ell_2 \in \mathbb{N}$*: "*$\ell$: if $r_i \neq 0$ then ℓ_1 else ℓ_2*"*
– *halt and accept: "*ℓ: accept*"*
– *halt and reject: "*ℓ: reject*".*

The labels of the instructions will sometimes be left implicit.

We assume, without loss of generality, that it is never the case that a register or a label are mentioned multiple times in the same instruction (e.g., in "$\ell : r_i \coloneqq r_j \bullet r_k$" we assume $i \neq j$, $j \neq k$, and $i \neq k$).

Since RAMs operate on natural numbers, we only allow *non-negative* subtraction, i.e., $x - y = 0$ when $y > x$.

Definition 3. *Let* $L \subseteq \{0,1\}^\star$ *be a language, and let* M *be a* RAM*. We say that* M decides L *in polynomial time iff, for all* $x \in \{0,1\}^\star$*, when the integer having binary representation* $1x$ *is loaded into a specified input register, the machine* M *behaves as follows:*

– *if* $x \in L$*, then* M *reaches an "*accept*" instruction*
– *if* $x \notin L$*, then* M *reaches a "*reject*" instruction*

within a number of steps bounded by $O(|x|^k)$ *for some* $k \in \mathbb{N}$*.*

In the rest of this paper we will denote the class of random access machines using the set of basic operations $X \subseteq \{+, -, \times, \div\}$ by RAM(X), and the class of all-parallel EN P systems whose production functions can be expressed in terms of X by ENP(X). In particular, we are interested in all-parallel EN P systems having linear production functions, ENP($+, -$), and those with production functions consisting of polynomials augmented by integer division, ENP($+, -, \times, \div$).

We shall also employ the following notation for complexity classes:

Definition 4. *Let* D *be one of the classes of computing devices described above. Then, by* **P**-D *we denote the class of decision problems solvable in polynomial time by devices of type* D*.*

The computational power of polynomial-time RAMs is strictly dependent on the set of basic operations that can be computed in a single time step. When only addition and subtraction are available, then polynomial-time RAMs are equivalent to polynomial-time Turing machines [4].

Proposition 1. **P**-RAM$(+, -) =$ **P**. □

On the other hand, multiplication and division considerably increase the efficiency of polynomial-time RAMs [1]:

Proposition 2. **P**-RAM$(+, -, \times, \div) =$ **PSPACE**. □

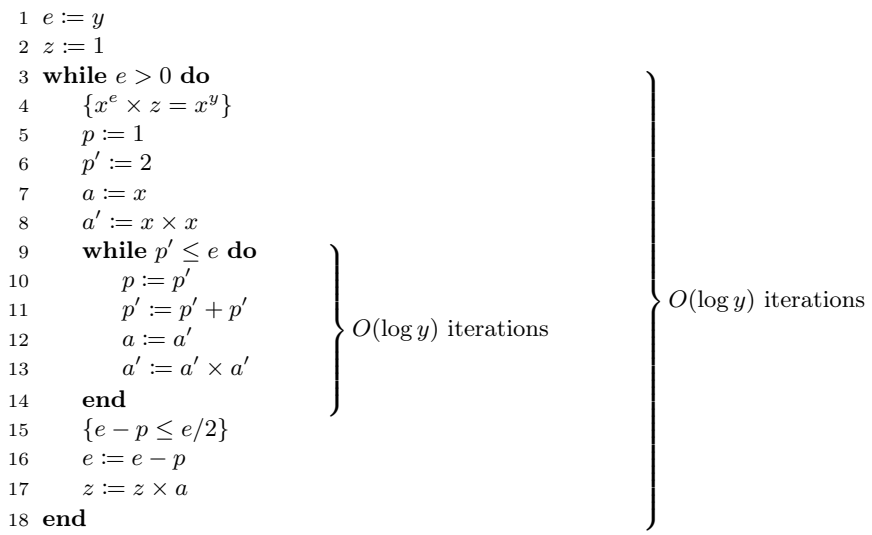

Fig. 1. Polynomial-time exponentiation algorithm by repeated squaring.

3 Avoiding Indirect Addressing

In this section we recall how indirect addressing may be eliminated from random access machines by encoding any number of registers as a single large integer. The resulting machine only needs a constant number of registers and, when the original machine runs in polynomial time, the slowdown is only polynomial.

In order to eliminate indirect addressing we employ multiplication, integer division and exponentiation. The first two operations, which are built-in on a RAM$(+, -, \times, \div)$, can be computed in quadratic time by a RAM$(+, -)$ using repeated doubling.

Proposition 3. *The product $x \times y$ and the quotient $x \div y$ can be computed in $O\big((\log y)^2\big)$ time and $O\big((\log x)^2\big)\big)$ time respectively by a* RAM$(+, -)$ *using a constant number of auxiliary registers.* □

Exponentiation can be also computed in polynomial time, using a repeated squaring algorithm, both by a RAM$(+, -)$ and a RAM$(+, -, \times, \div)$.

Proposition 4. *The exponential x^y can be computed in $O\big((\log y)^2\big)$ time by a* RAM$(+, -, \times)$ *and in $O\big((y \log y \log x)^2\big)$ time by a* RAM$(+, -)$ *using a constant number of auxiliary registers.*

Proof. The algorithm of Fig. 1 computes $z \coloneqq x^y$ by repeated squaring.

The outermost loop maintains the invariant $x^e \times z = x^y$, and the innermost loop computes the largest power 2^i less than or equal to e, which is then subtracted from e, thus reducing the value of this register by half or more (hence,

eventually, to 0); the product of the values x^{2^i} is accumulated into z. In other words, the algorithm computes the value x^y as

$$\begin{aligned} x^y &= x^{y_m 2^m} \times x^{y_{m-1} 2^{m-1}} \times \cdots \times x^{y_1 2^1} \times x^{y_0 2^0} \\ &= x^{y_m 2^m + y_{m-1} 2^{m-1} + \cdots + y_1 2^1 + y_0 2^0} \end{aligned}$$

where $y_m y_{m-1} \cdots y_1 y_0$ is the binary expansion of y.

Each line of the algorithm is performed by a RAM$(+, -, \times)$ in constant time, for a total of $O\big((\log y)^2\big)$ time. On a RAM$(+, -)$, the product of line 8 is computed in $O\big((\log x)^2\big)$ time, and the products of lines 13 and 17 in $O\big((\log x^y)^2\big) = O\big((y \log x)^2\big)$ time, since a reaches the value x^y in the worst case (i.e., when y is a power of 2). The total time is thus $O\big((y \log y \log x)^2\big)$. □

An arbitrary random access machine never uses more registers than time steps; however, in principle, the largest register index employed can be exponential on a RAM$(+, -)$, or even doubly exponential on a RAM$(+, -, \times, \div)$. The following proposition [5] obviates the problem.

Proposition 5. *Let M be a* RAM *with addition, subtraction and possibly multiplication and division, working in time $t(n)$. Then there exists a* RAM *with the same basic operations working in time $O\big(t(n)^2\big)$, having the same output as M, and using only its first $O\big(t(n)\big)$ registers.* □

The three Propositions 3, 4, and 5 allow us to simulate indirect addressing from polynomial-time RAMs with a polynomial slowdown.

Proposition 6. *Let M_1 be a* RAM$(+, -)$ *(respectively, a* RAM$(+, -, \times, \div)$*) working in polynomial time $O(n^k)$. Then, there exists a* RAM$(+, -)$ *(resp., a* RAM$(+, -, \times, \div)$*) M_2 working in $O\big(n^{8k} (\log n)^2\big)$ time (resp., $O\big(n^{2k} (\log n)^2\big)$) and computing the same result as M_1 without using indirect addressing.*

Proof. Since M_1 works in polynomial time, by Proposition 5 there exists another RAM M_1' with the same output as M_1, working in polynomial time $t = c_1 n^{2k} + c_0$ and using at most the first $m = d_1 n^k + d_0$ registers (for some $c_0, c_1, d_0, d_1 \in \mathbb{N}$).

The machine M_2 simulates M_1' as follows. All the registers $(r_0, \ldots, r_{m-1})$ of M_1' are stored in a single register r of M_2 as a base-b number:

$$r = b^{m-1} r_{m-1} + b^{m-2} r_{m-2} + \cdots + b^1 r_1 + b^0 r_0.$$

The base b is one more than the largest number that can ever be stored in a register by M_1', which can be computed as follows:

- If M_1' is a RAM$(+, -)$, the most expensive instruction (in terms of magnitude of the values of the registers) is "$x \coloneqq x + x$", where x is the input register. After t steps, the value of any register is thus bounded by $2^t x$, and we choose $b = 2^t x + 1$.

– If M_1' is a RAM$(+, -, \times, \div)$, then the most expensive instruction is squaring, i.e., "$x \coloneqq x \times x$", leading to an upper bound of x^{2^t} after t steps. In this case, we choose $b = x^{2^t} + 1$.

Notice that r has an upper bound of b^{m+1}.

The machine M_2 first computes the length $n = O(\log x)$ of the input (contained in the register x) as follows:

```
1 y ≔ x
2 n ≔ 0
3 while y ≠ 0 do
4     y ≔ y ÷ 2
5     n ≔ n + 1
6 end
```

This requires $O(\log x)$ steps on a RAM$(+, -, \times, \div)$, and $O\big((\log x)^3\big)$ steps on a RAM$(+, -)$, due to the cost of the division of line 4.

M_2 then computes the number of steps t of M_1' to be simulated:

```
7 t ≔ c_1 n^{2k} + c_0
```

Line 7 can be executed in $O(1)$ time by a RAM$(+, -, \times, \div)$, since k, c_0, and c_1 are constants; on a RAM$(+, -)$ the time is $O\big((\log n)^2\big) = O\big((\log \log x)^2\big)$. Notice that evaluating such complex expressions only requires a constant number of auxiliary registers.

The base b described above is then computed. For a RAM$(+, -)$ the calculation is

```
8 b ≔ 2^t x + 1
```

which executes in $O\big((t \log t)^2 + (\log x)^2\big) = O\big((n^{2k} \log n)^2\big)$ time.

For a RAM$(+, -, \times, \div)$ the calculation is

```
8 b ≔ x^{2^t} + 1
```

which executes in $O(t^2) = O(n^{4k})$ time.

The last phase of the initialisation of M_2 sets up register r, which initially contains only x in its 0-th position:

```
9 r ≔ x
```

Every time a register of M_1', say r_i (with i a constant), has to be read, its value can be extracted from the register of r of M_2 and stored in an auxiliary register, say y, as follows:

$$y \coloneqq (r \div b^i) \bmod b$$

where $a \bmod b = a - (a \div b \times b)$. This requires

$$\begin{aligned} O\big((\log b)^2 + (\log r)^2\big) &= O\big((\log b)^2 + (\log b^{m+1})^2\big) = O\big((\log b^{m+1})^2\big) \\ &= O\big((m \log b)^2\big) = O\big(((d_1 n^k + d_0) \log(2^t x + 1))^2\big) \\ &= O\big((n^k (t + \log x))^2\big) = O(n^{6k}) \end{aligned}$$

time on a RAM$(+, -)$, and $O(1)$ time on a RAM$(+, -, \times, \div)$.

If indirect access is needed, that is, we read r_i where $i < m$ is *not* a constant, then the computation time becomes

$$\begin{aligned} O\big((i \log i \log b)^2 + (\log r)^2\big) &= O\big((m \log m \log b)^2 + (m \log b)^2\big) \\ &= O\big((m \log m \log b)^2\big) = O\big((n^k \log n \log b)^2\big) \\ &= O\big((n^k \log n \cdot (t + \log x))^2\big) \\ &= O\big((n^k \log n \cdot n^{2k})^2\big) = O\big((n^{3k} \log n)^2\big) \\ &= O\big(n^{6k} (\log n)^2\big) \end{aligned}$$

on a RAM$(+, -)$, and

$$O\big((\log i)^2\big) = O\big((\log m)^2\big) = O\big((\log n)^2\big)$$

on a RAM$(+, -, \times, \div)$. Hence, reading a register of M_1' (and, in particular, indirect addressing) can be simulated in polynomial time both on a RAM$(+, -)$ and on a RAM$(+, -, \times, \div)$.

The operation of writing the value of a register y of M_2 into a simulated register r_i of M_1' is similar:

1 $z \coloneqq (r \div b^i) \bmod b$
2 $r \coloneqq r - (z \times b^i) + (y \times b^i)$

and has the same asymptotical time complexity as above (keeping in mind that i is a constant in this case).

We can now finally describe how the instructions of M_1' are simulated by M_2.

- Assignment of a constant "$r_i \coloneqq c$"
  $z \coloneqq (r \div b^i) \bmod b$
  $r \coloneqq r - (z \times b^i) + (c \times b^i)$
- Copying the value of a register "$r_i \coloneqq r_j$"
  $y \coloneqq (r \div b^j) \bmod b$
  $z \coloneqq (r \div b^i) \bmod b$
  $r \coloneqq r - (z \times b^i) + (y \times b^i)$
- Copying the value of a register through indirect addressing "$r_i \coloneqq r_{r_j}$"
  $y \coloneqq (r \div b^j) \bmod b$
  $y' \coloneqq (r \div b^y) \bmod b$
  $z \coloneqq (r \div b^i) \bmod b$
  $r \coloneqq r - (z \times b^i) + (y' \times b^i)$

– Arithmetical operations "$r_i \coloneqq r_j \bullet r_k$" with $\bullet \in \{+, -\}$ (for a RAM$(+, -)$) or $\bullet \in \{+, -, \times, \div\}$ (for a RAM$(+, -, \times, \div)$)

  $y_1 \coloneqq (r \div b^j) \bmod b$
  $y_2 \coloneqq (r \div b^k) \bmod b$
  $y \coloneqq y_1 \bullet y_2$
  $z \coloneqq (r \div b^i) \bmod b$
  $r \coloneqq r - (z \times b^i) + (y \times b^i)$

– Conditional jump "if $r_i \neq 0$ then ℓ_1 else ℓ_2"

  $y \coloneqq (r \div b^i) \bmod b$
  if $y \neq 0$ then ℓ'_1 else ℓ'_2

  where ℓ'_1 (resp., ℓ'_2) is the label of the first of the instructions of M_2 simulating the instruction ℓ_1 (resp., ℓ_2) of M'_1.

The discussion above implies that simulating each instruction of M'_1 requires at most $O\big(n^{6k}(\log n)^2\big)$ for a RAM$(+, -)$, and $O\big((\log n)^2\big)$ for a RAM$(+, -, \times, \div)$. Hence, the total number of steps to complete the simulation is $O\big(n^{8k}(\log n)^2\big)$ and $O\big(n^{2k}(\log n)^2\big)$ respectively. □

4 Simulating RAMs without Indirect Addressing

We now prove that each RAM, whose instructions satisfy the mild constraints we have imposed in the definition, and do not use indirect addressing, can be simulated by an appropriate EN P system working in the all-parallel mode. The simulation is efficient, in the sense that each RAM instruction is simulated in just one step.

Theorem 1. *Let M be a* RAM *that does not use indirect addressing. Then, for each instruction of M there exists a set of programs for an all-parallel EN P system Π that simulates it in one computation step.*

Proof. We proceed by examining all possible cases. In what follows, z is a variable whose value is always zero, variables r_i, r_j, r_k represent registers of M (containing non negative integer values), and variables p_ℓ assume values in $\{0, 1\}$ to indicate the next instruction of M to be simulated.

RAM instructions of type "$\ell\colon r_i \coloneqq k$" can be simulated by the following set of all-parallel programs:

$$0r_i + k + z|_{p_\ell} \to 1|r_i$$
$$p_\ell \to 1|p_{\ell+1}$$

When $p_\ell = 0$ the first program is not executed, while the second program zeroes p_ℓ (thus leaving its value unaltered) and gives a contribution of zero to variable $p_{\ell+1}$, thus behaving as a NOP (No OPeration). Hence no interference is produced in the variables involved in the RAM instruction currently simulated. On the other hand, if $p_\ell = 1$ then the first program first zeroes r_i and then

assigns the value k to it, while the second program zeroes p_ℓ and sets $p_{\ell+1}$ to 1, thus pointing to the next instruction of M to be simulated.

Assignment instructions of type "$\ell\colon r_i \coloneqq r_j$", with $j \neq i$, can be simulated using the following programs:

$$\begin{aligned} &0r_i + 2r_j + z|_{p_\ell} \to 1|r_i + 1|r_j \\ &p_\ell \to 1|p_{\ell+1} \end{aligned}$$

As in the previous case, when $p_\ell = 0$ the first program is not active while the second one operates like a NOP. When $p_\ell = 1$, instead, the first program first zeroes both r_i and r_j and then assigns to them the old value of r_j; the second program, as before, passes the control to instruction $\ell+1$. Albeit in our definition of RAMs we have avoided the case when $j = i$, here we just observe that we can also easily deal with it: we simply remove the first program, since in this case it always operates like a NOP.

Arithmetic instructions of type "$\ell\colon r_i \coloneqq r_j \bullet r_k$", with $\bullet \in \{+, -, \times, \div\}$ and $i \neq j$, $j \neq k$, and $i \neq k$, can be simulated as follows:

$$\begin{aligned} &0r_i + r_j \bullet r_k + z|_{p_\ell} \to 1|r_i \\ &r_j + z|_{p_\ell} \to 1|r_j \qquad (2) \\ &r_k + z|_{p_\ell} \to 1|r_k \qquad (3) \\ &p_\ell \to 1|p_{\ell+1} \end{aligned}$$

When $p_\ell = 0$ the first three programs are not executed, while the last program behaves as a NOP. On the other hand, if $p_\ell = 1$ then the first program first zeroes variables r_i, r_j and r_k, and then it assigns to r_i the result of the operation $r_j \bullet r_k$, using the old values of r_j and r_k. Programs (2) and (3) are used to preserve the old values of variables r_j and r_k, whereas the last program passes the control to instruction $\ell + 1$.

Finally, instructions of type "$\ell\colon$ if $r_i \neq 0$ then ℓ_1 else ℓ_2", with $\ell \neq \ell_1$, $\ell \neq \ell_2$, and $\ell_1 \neq \ell_2$, can be simulated by the following programs:

$$\begin{aligned} &p_\ell \to 1|p_{\ell_1} \\ &r_i - 1|_{p_\ell} \to 1|p_{\ell_1} \qquad (4) \\ &r_i + 1|_{p_\ell} \to 1|p_{\ell_2} \qquad (5) \end{aligned}$$

in which we assume $r_i \neq 0$ and correct if this is not the case. Note, in particular, that programs (4) and (5) are active if and only if $p_\ell = 1$ and $r_i = 0$. So, when $p_\ell = 0$ only the first program is executed, behaving as a NOP. When $p_\ell = 1$ and $r_i > 0$, the first program passes the control to instruction ℓ_1 whereas the other two programs are not executed. Finally, when $p_\ell = 1$ and $r_i = 0$ the first program zeroes p_ℓ and (incorrectly) sets p_{ℓ_1} to 1. This time, however, also the other two programs are executed: after resetting once again the value of r_i to 0, program (4) gives a contribution of -1 to p_{ℓ_1}, so that its final value will be zero, whereas program (5) sets p_{ℓ_2} to 1, indicating the next instruction of M to be simulated. □

5 Simulating all-parallel EN P Systems with RAMs

Having proved that all-parallel EN P systems are able to simulate efficiently random access machines using the same arithmetic operations, we now turn our attention to the converse simulation. Without loss of generality, we assume that the all-parallel EN P systems being simulated have a single membrane [6].

Since the production functions of EN P systems may evaluate to negative numbers, even if the variables themselves are always non-negative, it is convenient to employ RAMs with registers holding values in $\mathbb{Z}$. This poses no restriction, since signed integers may be simulated with a constant-time slowdown by RAMs using non-negative numbers, for instance by storing them with a sign-and-modulus representation.

Proposition 7. *Let Π be an* $\mathrm{ENP}(+,-)$ *(respectively, an* $\mathrm{ENP}(+,-,\times,\div)$*) working in all-parallel mode and polynomial time $t(n) \leq c_1 n^k + c_0$. Then, there exists a* $\mathrm{RAM}(+,-)$ *(respectively, a* $\mathrm{RAM}(+,-,\times,\div)$*) M computing the same output as Π in time $O\big(t(n)^3\big)$ (respectively, $O\big(t(n)\big)$).*

Proof. Let $x_1, \ldots, x_m$ be the variables of Π. The machine M stores the values of these variables in registers that we will denote with the same names, and will have the same values in the initial configuration, including the input variable of Π. Let $p_1, \ldots, p_h$ be the programs of Π.

Before describing the simulation proper, let us compute the maximum value of a variable of Π. If Π is an $\mathrm{ENP}(+,-)$, then the rules have one of the following forms:

$$a_{i_1} x_{i_1} \pm \cdots \pm a_{i_k} x_{i_k} \pm a \to b_1|x_1 + \cdots + b_m|x_m$$
$$a_{i_1} x_{i_1} \pm \cdots \pm a_{i_k} x_{i_k} \pm a|_e \to b_1|x_1 + \cdots + b_m|x_m$$

for some constants $a, a_{i_1}, \ldots, a_{i_k}, b_1, \ldots, b_m \in \mathbb{N}$. The following program, with some constant $a \in \mathbb{N}$, produces the maximum increase in the variable x, which we assume to be the input variable:

$$ax \to 1|x \tag{6}$$

After $t = c_1 n^k + c_0$ computation steps, the value of x reaches its maximum $a^t x$. (Naturally, a program such as (6) is not admissible in a halting EN P system; that program is considered here only in order to provide an upper bound to the value of the variables of Π.)

On the other hand, if Π is an $\mathrm{ENP}(+,-,\times,\div)$, the program that maximises the value of x is

$$x^a \to 1|x$$

for some $a \in \mathbb{N}$. In this case, after t steps the value of x reaches x^{a^t}. These upper bounds to the values of the variables of Π will be used later in order to determine the time required by M in order to simulate the EN P system.

The following is an overview of the simulation of Π:

repeat
save the current values of the variables
compute the variations due to p_1 *(if applicable)*
$\vdots$
compute the variations due to p_h *(if applicable)*
compute the new values of the variables
until *a final configuration is reached*
if Π *accepted* **then**
accept
else
reject
end

At the beginning of each simulated step, the current values of the variables are copied:

$$x'_1 \coloneqq x_1$$
$$\vdots$$
$$x'_m \coloneqq x_m$$

In the variables $\Delta_1, \ldots, \Delta_m$, initially zero, we accumulate the contributions to $x_1, \ldots, x_m$ given by the programs of Π during the current step:

$$\Delta_1 \coloneqq 0$$
$$\vdots$$
$$\Delta_m \coloneqq 0$$

Each program p_i of the form $f(x_{i_1}, \ldots, x_{i_k}) \to a_1|x_1 + \cdots + a_m|x_m$ is simulated as follows:

$$f \coloneqq f(x_{i_1}, \ldots, x_{i_k})$$
$$x'_{i_1} \coloneqq 0$$
$$\vdots$$
$$x'_{i_k} \coloneqq 0$$
$$u \coloneqq f \div (a_1 + \cdots + a_m)$$
$$\Delta_1 \coloneqq \Delta_1 + a_1 u$$
$$\vdots$$
$$\Delta_m \coloneqq \Delta_m + a_m u$$

First, the value of the production function is computed. This requires $O(1)$ time, since by construction Π and M admit the same basic arithmetic operations. Then, the copies of the variables occurring on the left-hand side of the program are zeroed.

The unit u to be distributed according to the repartition protocol is then computed. Here the division is performed in $O(1)$ time if M is a RAM$(+,-,\times,\div)$, but $O\big((\log f)^2\big) = O\big((\log(a^t x))^2\big) = O(t^2) = O(n^{2k})$ if it is a RAM$(+,-)$.

Finally, the contributions to the variables of Π are updated according to the repartition protocol. This only requires $O(1)$ time, as $a_1, \ldots, a_m$ are constants.

Programs p_i of the form $f(x_{i_1}, \ldots, x_{i_k})|_e \to a_1|x_1 + \cdots + a_m|x_m$ are simulated analogously, only with an extra test in order to ensure that the value of the enzyme is larger than the minimum of the variables.

if $e > x_{i_1}$ **or** $e > x_{i_1}$ **or** $\cdots$ **or** $e > x_{i_k}$ **then**
  $f \coloneqq f(x_{i_1}, \ldots, x_{i_k})$
  $x'_{i_1} \coloneqq 0$
  $\vdots$
  $x'_{i_k} \coloneqq 0$
  $u \coloneqq f \div (a_1 + \cdots + a_m)$
  $\Delta_1 \coloneqq \Delta_1 + a_1 u$
  $\vdots$
  $\Delta_m \coloneqq \Delta_m + a_m u$
end

The time required is again $O(1)$ if Π is an ENP$(+,-,\times,\div)$ and $O(n^{2k})$ if it is an ENP$(+,-)$.

After all programs have been examined (and applied, when possible), we can check whether a final configuration is reached: this occurs when, for each variable x_i, we have $x_i = x'_i + \Delta_i$, i.e., when the old value x_i equals the (possibly zeroed) value increased by the sum of the contributions it received in the current simulated step. If this is *not* the case, then the values of the variables are updated:

$x_1 \coloneqq x'_1 + \Delta_1$
$\vdots$
$x_m \coloneqq x'_m + \Delta_m$

and the next step of Π is simulated.

When a final configuration is actually reached, the machine M checks the value of the *accept* variable of Π and provides the same result:

if *accept* $= 1$ **then**
  accept
else
  reject
end

The total time required in order to perform the simulation of Π is $O(n^{3k})$ for an ENP$(+,-)$, and $O(n^k)$ for an ENP$(+,-,\times,\div)$. □

We can now state our main result, summarising the computational efficiency of EN P systems using arithmetic operations.

Theorem 2. *The following complexity classes coincide:*

$$\mathbf{P}\text{-ENP}(+,-) = \mathbf{P}\text{-RAM}(+,-) = \mathbf{P}$$
$$\mathbf{P}\text{-ENP}(+,-,\times,\div) = \mathbf{P}\text{-RAM}(+,-,\times,\div) = \mathbf{PSPACE}$$

Furthermore, the inclusion $\mathbf{P}\text{-ENP}(+,-,\times) \subseteq \mathbf{P}\text{-RAM}(+,-,\times)$ *holds.* □

6 Conclusions

We have analysed the computational efficiency of all-parallel EN P systems and their relationships with more traditional computing devices such as RAMs and Turing machines. We have showed some efficient simulations of all-parallel EN P systems by RAMs and vice versa, when the same basic arithmetic operations are used.

Hence we found that, by using only addition and subtraction, EN P systems working in polynomial time and all-parallel mode characterise the complexity class **P**, whereas by also allowing multiplication and integer division we obtain a characterisation of **PSPACE**.

Establishing the precise efficiency of all-parallel EN P systems (as well as random access machines) with addition, subtraction and multiplication is still an open problem. The possibility to extend the results exposed in this paper to EN P systems working in the sequential or in the one-parallel mode, as well as to numerical P systems not using the enzyme control, is also open.

References

1. Bertoni, A., Mauri, G., Sabadini, N.: A characterization of the class of functions computable in polynomial time on random access machines. In: STOC '81 Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. pp. 168–176 (1981), http://dx.doi.org/10.1145/800076.802470
2. Bottoni, P., Martin-Vide, C., Păun, Gh., Rozenberg, G.: Membrane systems with promoters/inhibitors. Acta Informatica 38(10), 695–720 (2002), http://dx.doi.org/10.1007/s00236-002-0090-7
3. Buiu, C., Vasile, C., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences 187, 33–51 (2012), http://dx.doi.org/10.1016/j.ins.2011.10.007
4. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. Journal of Computer and System Sciences 7, 354–375 (1973), http://dx.doi.org/10.1016/S0022-0000(73)80029-7
5. Hartmanis, J., Simon, J.: On the power of multiplication in random access machines. In: IEEE Conference Record of 15th Annual Symposium on Switching and Automata Theory. pp. 13–23 (1974), http://dx.doi.org/10.1109/SWAT.1974.20

6. Leporati, A., Porreca, A.E., Zandron, C., Mauri, G.: Improved universality results for parallel enzymatic numerical P systems. International Journal of Unconventional Computing 9, 385–404 (2013), http://www.oldcitypublishing.com/IJUC/IJUCcontents/IJUCv9n5-6contents.html
7. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
8. Păun, Gh., Păun, R.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae 73(1–2), 213–227 (2006), http://iospress.metapress.com/content/7xyefwrwy7mkg46a/
9. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
10. Pavel, A.B., Arsene, O., Buiu, C.: Enzymatic numerical P systems – A new class of membrane computing systems. In: Li, K., Tang, Z., Li, R., Nagar, A.K., Thamburaj, R. (eds.) Proceedings 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010). pp. 1331–1336 (2010), http://dx.doi.org/10.1109/BICTA.2010.5645071
11. Pavel, A.B., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing 11(3), 387–393 (2012), http://dx.doi.org/10.1007/s11047-011-9286-5
12. Vasile, C.I., Pavel, A.B., Dumitrache, I., Păun, Gh.: On the power of enzymatic numerical P systems. Acta Informatica 49, 395–412 (2012), http://dx.doi.org/10.1007/s00236-012-0166-y