ELSEVIER

Theoretical Computer Science 215 (1999) 69–87

Theoretical Computer Science

An optimal parallel algorithm to convert a regular expression into its Glushkov automaton

Djelloul Ziadi*, Jean-Marc Champarnaud

Faculté des Sciences, Laboratoire d'Informatique de Rouen, Université de Rouen, B.P. 118, Place Emile Blondel, 76821 Mont-Saint-Aignan, Cédex, France

Received June 1996
Communicated by M. Nivat

Abstract

The aim of this paper is to describe a CREW-PRAM optimal algorithm which converts a regular expression of size s into its Glushkov automaton in $O(\log s)$ time using $O(s^2/\log s)$ processors. This algorithm makes use of the star-normal form of an expression as defined by Brüggemann-Klein (1993) and is based on the sequential algorithm due to Ziadi et al. (1997) which computes an original representation of Glushkov automaton in $O(s)$ time.

Keywords: PRAM algorithms; Regular expressions; Finite automata

1. Introduction

PRAM model is a general framework used for describing and analyzing parallel algorithms. It consists of n processors working synchronously and exchanging data through a shared memory unit. We shall suppose here that concurrent reads are allowed, but concurrent writes are not (CREW-PRAM). A PRAM algorithm is said to be efficient if it works in a polylogarithmic time using a polynomial number of processors. It is said to be optimal if its sequential time (the product of its parallel time by the number of processors) is equal to the computation time of the fastest known sequential algorithm solving the problem. Our aim is to produce optimal PRAM algorithms in automata domain [13]; this paper describes such an algorithm, for converting a regular expression into an automaton.

There exist a lot of different sequential algorithms to convert a regular expression into an automaton. Watson's taxonomy [12] is an excellent reference for such a topic. Up to now, Thompson's approach [11], which yields a non-deterministic automaton with ε-transitions, is the only one to have been parallelized. Rytter algorithms [9] are

* Corresponding author. E-mail: ziadi@dir.univ-rouen.fr.

0304-3975/99/$ – see front matter
PII: S0304-3975(97)00131-X

based on adaptations of Thompson's method due to Hopcroft and Ullman [6], and to Sedgewick [10]. They work on a CREW-PRAM model and are optimal; they convert a regular expression of size s in $O(\log s)$ time using $O(s/\log s)$ processors.

Building an automaton from a regular expression is currently performed in order to test whether a word belongs to a given language or not. If the automaton has ε-transitions, their elimination is in $O(s^2)$ sequential time. This elimination is based on the computing of a transitive closure, for which there is no optimal PRAM algorithm. It is one of the reasons why we concentrate our efforts on computing a result without ε-transitions.

This paper describes a CREW-PRAM parallelization of Glushkov approach [5, 7] which yields a non-deterministic automaton without ε-transitions. Our parallel algorithm is based on a new sequential algorithm described by Ziadi et al. in [8, 14]. This sequential algorithm (named ZPC algorithm) first transforms a regular expression of size s into an original representation of its Glushkov automaton in $O(s)$ time. Our parallelization is based on the following results:

(1) an optimal algorithm which builds the ZPC representation in $O(\log s)$ time using $O(s/\log s)$ processors, as far as the expression is in star-normal form (notion due to Brüggemann-Klein [2]),
(2) an efficient algorithm which computes the star-normal form of a regular expression in $O(\log s)$ time using $O(s)$ processors,
(3) an optimal algorithm which converts the ZPC representation into a table of transitions in $O(\log s)$ time using $O(s^2/\log s)$ processors.

Combining 1 and 2 we get an efficient algorithm to compute the ZPC representation. Combining 1–3 we get an optimal algorithm to convert a regular expression into its Glushkov automaton.

Section 2 presents terminology and writing conventions, and recalls Glushkov construction. Section 3 briefly describes ZPC sequential algorithm and introduces the notion of star-normal form. Section 4 presents an optimal algorithm which computes ZPC representation for a regular expression in star-normal form. Section 5 describes an efficient algorithm which constructs the star-normal form of a regular expression. Section 6 provides an optimal algorithm which converts the ZPC representation into a table of transitions in $O(\log s)$ time using $O(s^2/\log s)$ processors.

2. Definitions and writing conventions

In this section, we first introduce the terminology and the notations used in this paper. With a few exceptions, these notations can be found in [8].

2.1. Regular expressions and languages

Let Σ be a non-empty finite set of symbols, called the alphabet. Σ^* represents the set of all words over Σ. The empty word is denoted by ε. The symbols in Σ are

represented by the first lower-case letters such as $a, b, c, \ldots$. Union $(+)$, product $(\cdot)$, and Kleene star $(*)$ are the classical regular operations over the subsets of Σ^*. Upper-case letters such as E, F and G represent regular expressions. In order to specify the position of the symbols in the expression, the symbols are subscripted following the order of reading. For example, starting from $E=(a+\varepsilon)ba$ we obtain the subscripted expression $\overline{E}=(a_1+\varepsilon)b_2a_3$. Subscripts are called positions and are represented by the last lower-case letters of the alphabet, such as x, y, z. The set of positions of a regular expression E is denoted by $Pos(E)$. χ is the application which maps each position in $Pos(E)$ to the symbol of Σ which appears at this position in E. We denote by σ ($\sigma=\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, where $|Pos(E)|=n$) the subscripted alphabet. We denote by $L(E)$ the language generated by the regular expression E. We denote by $T(E)$ the syntax tree associated with E. If v is a node in $T(E)$, $symbol(v)$ is the operator or the operand associated with v. We write v_l (resp. v_r) the left (resp. right) child of v. If v has a single child (it is the case if v is labeled '$*$'), this child is written v_s. By E_v we denote the subexpression rooted at v. The size $|E|$ of a regular expression E is the length of its postfix form. Under the assumption that a node of $T(E)$ and its child cannot be both labeled by Kleene star, it is easy to prove that $|E|=O(|Pos(E)|)$. Therefore, we will express our complexity results as functions of $|E|$.

2.2. Glushkov automaton

In order to construct a non-deterministic finite automaton recognizing $L(E)$, Glushkov [5] has introduced four functions:

- $Null(E)$ is equal to $\{\varepsilon\}$ if $\varepsilon \in L(E)$ and $\emptyset$ otherwise.
- $First(E)$ is the set of initial positions of words of the language $L(E)$.
- $Last(E)$ is the set of final positions of words of the language $L(E)$.
- $Follow(E,x)$ is the set of positions which follow immediately the position x in the expression E.

Example 1. For the expression $E=(a+b)^*ab$, we have:

- $\overline{E}=(a_1+b_2)^*a_3b_4$,
- $Null(E)=\emptyset$,
- $First(E)=\{1,2,3\}$,
- $Last(E)=\{4\}$,
- $Follow(E,1)=Follow(E,2)=\{1,2,3\}$,
- $Follow(E,3)=\{4\}$, $Follow(E,4)=\emptyset$.

These four functions can be defined over σ in the following way:

- $Null(E)$ = **if** $\varepsilon \in L(\overline{E})$ **then** $\{\varepsilon\}$ **else** $\emptyset$
- $First(E)=\{x \in Pos(E) \mid \exists u \in \sigma^* : \lambda_x u \in L(\overline{E})\}$
- $Last(E)=\{x \in Pos(E) \mid \exists u \in \sigma^* : u\lambda_x \in L(\overline{E})\}$
- $Follow(E,x)=\{y \in Pos(E) \mid \exists v \in \sigma^*,\ \exists w \in \sigma^* : v\lambda_x\lambda_y w \in L(\overline{E})\}$.

We shall use the following notation: for each set X, we note $\mathcal{I}_X$ the function which is equal to $\{\varepsilon\}$ for all $x \in X$ and $\emptyset$ otherwise.

Proposition 1. *Null(E) can be inductively computed as follows*:

$$
\begin{aligned}
Null(\{\varepsilon\}) &= \{\varepsilon\}\\
Null(\emptyset) &= \emptyset\\
Null(a) &= \emptyset\\
Null(F+G) &= Null(F)\cup Null(G)\\
Null(F\cdot G) &= Null(F)\cap Null(G)\\
Null(F^*) &= \{\varepsilon\}
\end{aligned}
$$

Proposition 2. *First can be inductively computed as follows*:

$$
\begin{aligned}
First(\varepsilon) &= \emptyset\\
First(\emptyset) &= \emptyset\\
First(x) &= \{x\}\\
First(F+G) &= First(F)\cup First(G)\\
First(F\cdot G) &= First(F)\cup Null(F)\cdot First(G)\\
First(F^*) &= First(F)
\end{aligned}
$$

Proposition 3. *Last can be inductively computed as follows*:

$$
\begin{aligned}
Last(\varepsilon) &= \emptyset\\
Last(\emptyset) &= \emptyset\\
Last(x) &= \{x\}\\
Last(F+G) &= Last(F)\cup Last(G)\\
Last(F\cdot G) &= Last(G)\cup Null(G)\cdot Last(F)\\
Last(F^*) &= Last(F)
\end{aligned}
$$

Proposition 4. *Follow(E,x) can be inductively computed as follows*:

$$
\begin{aligned}
Follow(\varepsilon,x) &= \emptyset\\
Follow(\emptyset,x) &= \emptyset\\
Follow(a,x) &= \emptyset\\
Follow(F+G,x) &= \mathcal{I}_{Pos(F)}(x)\cdot Follow(F,x)\cup \mathcal{I}_{Pos(G)}(x)\cdot Follow(G,x)\\
Follow(F\cdot G,x) &= \mathcal{I}_{Pos(F)}(x)\cdot Follow(F,x)\cup \mathcal{I}_{Pos(G)}(x)\cdot Follow(G,x)\\
&\quad \cup \mathcal{I}_{Last(F)}(x)\cdot First(G)\\
Follow(F^*,x) &= \mathcal{I}_{Pos(F)}(x)\cdot Follow(F,x)\cup \mathcal{I}_{Last(F)}(x)\cdot First(F)
\end{aligned}
$$

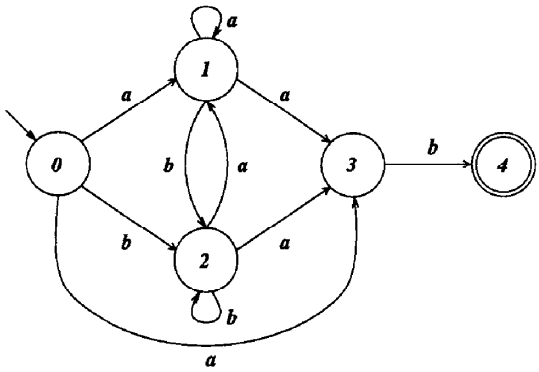


Fig. 1. Glushkov automaton of the expression $E=(a+b)^*\cdot a\cdot b$.

Definition 1. The Glushkov automaton $M_E=(Q_E,\Sigma,\delta_E,s_I,F_E,\chi)$ of the expression E is defined as follows:
- $Q_E=Pos(E)\cup\{s_I\}$
- $\forall a\in\Sigma,\ \delta_E(s_I,a)=\{y\mid y\in First(E) \text{ and } \chi(y)=a\}$
- $\forall a\in\Sigma,\ \forall x\in Pos(E),\ \delta_E(x,a)=\{y\mid y\in Follow(E,x) \text{ and } \chi(y)=a\}$
- $F_E=Last(E)\cup Null(E)\cdot\{s_I\}$

Theorem 1 (Ziadi et al. [14]). *Let E be a regular expression and M_E its Glushkov automaton, then $L(E)=L(M_E)$* (Fig. 1).

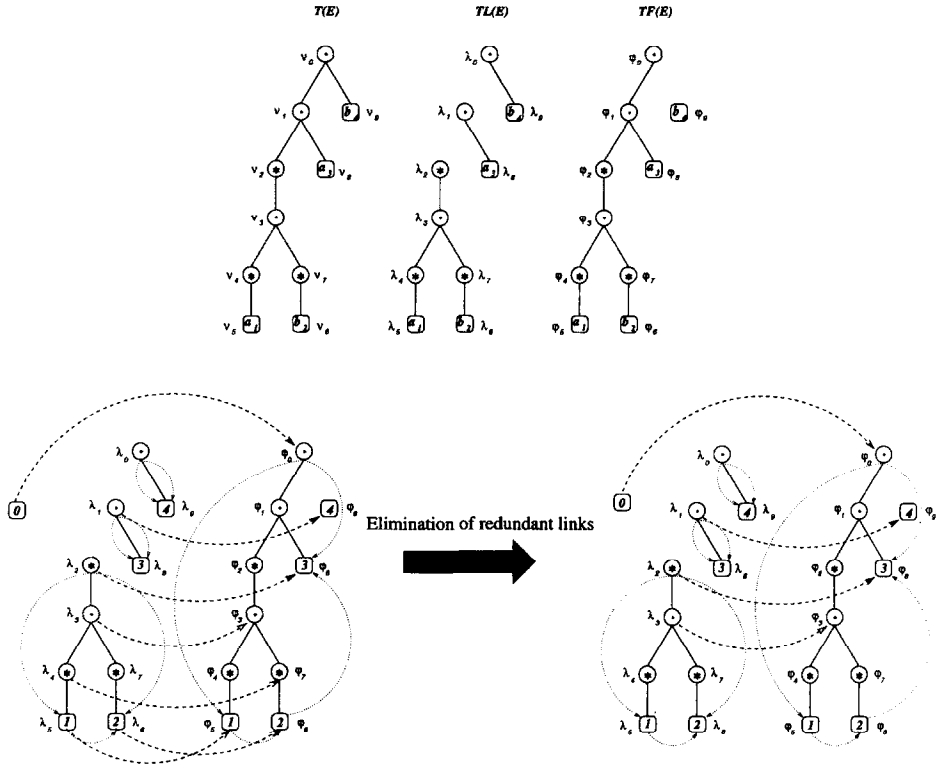
3. ZPC sequential algorithm

Let E be a regular expression of size s. A naive implementation of Glushkov algorithm leads to a $O(s^3)$ time complexity. Brüggemann-Klein [2], Chang and Paige [3], and Ziadi et al. [8, 14] have proposed algorithms with an $O(s^2)$ time complexity. The latter one (named ZPC algorithm) is the basis of our parallelization. We briefly describe it now.

3.1. Computation of First and Last

We consider the syntax tree $T(E)$ and for each node v in $T(E)$, we denote by $Set(v)$ the set of leaves in the subtree whose root is v. This set will be represented by a list. In order to have an $O(1)$ access to $Set(v)$, v points to its leftmost leaf and to its rightmost leaf. The forest $TF(E)$ which maps each node v to $First(E_v)$ is computed in the following way according to the Proposition 2:

1. Initialize $TF(E)$ by $T(E)$.
2. For every node v labeled "$\cdot$", cut the link to its right child v_r if $Null(E_{v_l})=\emptyset$, with respect to the statement $First(F\cdot G)=First(F)\cup Null(F)\cdot First(G)$.
3. For each node, update its pointers to its leftmost and rightmost leaves, and link the rightmost leaf of its left child to the leftmost leaf of its right child.



ν_i	$Last(\nu_i)$	$First(\nu_i)$		Δ_{ν_i}
ν_5	{1}	{1}		= ∅
ν_4	{1}	{1}	$Last(\nu_5) \times First(\nu_5) = \{1\} \times \{1\}$	= {(1,1)}
ν_6	{2}	{2}		= ∅
ν_7	{2}	{2}	$Last(\nu_6) \times First(\nu_6) = \{2\} \times \{2\}$	= {(2,2)}
ν_3	{1,2}	{1,2}	$Last(\nu_4) \times First(\nu_7) = \{1\} \times \{2\}$	= {(1,2)}
ν_2	{1,2}	{1,2}	$Last(\nu_3) \times First(\nu_3) = \{1,2\} \times \{1,2\}$	= {(1,1),(1,2),(2,1),(2,2)}
ν_8	{3}	{3}		= ∅
ν_1	{3}	{1,2,3}	$Last(\nu_2) \times First(\nu_8) = \{1,2\} \times \{3\}$	= {(1,3),(2,3)}
ν_9	{3}	{3}		= ∅
ν_0	{4}	{1,2,3}	$Last(\nu_1) \times First(\nu_9) = \{3\} \times \{4\}$	= {(3,4)}

Fig. 2. ZPC algorithm for the expression $E = (a_1^* b_2^*)^* \cdot a_3 \cdot b_4$.

The forest $TL(E)$ which maps each node v to $Last(E_v)$ is computed in a dual way with respect to the statement $Last(F \cdot G) = Last(G) \cup Null(G) \cdot Last(F)$ of Proposition 3 (see Fig. 2).

3.2. Computation of $\delta(M_E)$

Let Δ_v be the set of edges induced by the node v. Δ_v is defined as follows:

$$\Delta_v = \begin{cases} Last(E_{v_l}) \times First(E_{v_r}) & \text{if } v \text{ is labeled } \cdot, \\ Last(E_{v_s}) \times First(E_{v_s}) & \text{if } v \text{ is labeled } *, \\ \emptyset & \text{otherwise.} \end{cases}$$

On the data structure we use, cartesian products involved by Δ_v calculus are implemented by a pointer from v_l in $TL(E)$ to v_r in $TF(E)$ if v is labeled by "$\cdot$", or from v_s in $TL(E)$ to v_s in $TF(E)$ if v is labeled by "$*$" (see Fig. 2). These pointers are called "follow links".

Proposition 5 (Ziadi et al. [14]). $\delta(M_E) = (\bigcup_{v \in T(E)} \Delta_v) \cup (\{s_I\} \times First(E))$.

$\delta(M_E)$ is not necessarily a disjoint union. ZPC algorithm eliminates redundant Δ_v sets in order to get a disjoint union. The parallel algorithm we describe in the next section makes use of the fact that the star-normal form of the expression E also yields a disjoint union.

3.3. Star-normal form

According to Brüggemann-Klein [2], a regular expression E is said to be in star-normal form (SNF) if for each H such that H^* is a subexpression of E, we have

$$\forall x \in Last(H), \quad Follow(H,x) \cap First(H) = \emptyset.$$

This condition is called *SNF* condition.

Theorem 2 (Brüggemann-Klein [2]). *For each regular expression E, there is a regular expression $E^\bullet$ called the star-normal form of E such that*

1. *$E^\bullet$ satisfies the SNF condition,*
2. *$M_{E^\bullet} = M_E$,*
3. *$M_{E^\bullet}$ is computed in $O(s^2)$ time, where $s = |E|$.*

In order to compute $E^\bullet$, Brüggemann-Klein introduces the expression E° verifying the following conditions:

1. E° satisfies the SNF condition,
2. $M_{E^{\circ *}} = M_{E^*}$.

By recursively substituting each subexpression H^* of E with $H^{\circ *}$, we obtain $E^\bullet$.

Proposition 6 (Brüggemann-Klein [2]). *$E^\bullet$ can be computed by the following inductive rules:*

$$\begin{array}{ll} [E = \varepsilon \text{ or } \emptyset] & E^\bullet = E \\ [E = a] & E^\bullet = E \\ [E = F + G] & E^\bullet = F^\bullet + G^\bullet \\ [E = F \cdot G] & E^\bullet = F^\bullet \cdot G^\bullet \\ [E = F^*] & E^\bullet = F^{\circ \bullet *} \end{array}$$

Proposition 7. *$E^{\circ\bullet}$ can be computed by the following inductive rules*:

$$
\begin{array}{ll}
[E=\varepsilon \text{ or } \emptyset] & E^{\circ\bullet}=\emptyset \\
[E=a] & E^{\circ\bullet}=E \\
[E=F+G] & E^{\circ\bullet}=F^{\circ\bullet}+G^{\circ\bullet} \\
[E=F\cdot G] & E^{\circ\bullet}=\begin{cases} F^{\bullet}\cdot G^{\bullet} & \text{if } Null(F)=\emptyset \text{ and } Null(G)=\emptyset \\ F^{\circ\bullet}\cdot G^{\bullet} & \text{if } Null(F)=\emptyset \text{ and } Null(G)=\{\varepsilon\} \\ F^{\bullet}\cdot G^{\circ\bullet} & \text{if } Null(F)=\{\varepsilon\} \text{ and } Null(G)=\emptyset \\ F^{\circ\bullet}+G^{\circ\bullet} & \text{if } Null(F)=\{\varepsilon\} \text{ and } Null(G)=\{\varepsilon\} \end{cases} \\
[E=F^*] & E^{\circ\bullet}=F^{\circ\bullet}
\end{array}
$$

Example 2. Computation of $E^{\bullet}$ for $E=(a^*b^*)^*ab$:

$$
\begin{aligned}
E^{\bullet} &= ((a^*b^*)^*ab)^{\bullet} \\
&= ((a^*b^*)^*a)^{\bullet}b^{\bullet} \\
&= ((a^*b^*)^*)^{\bullet}a^{\bullet}b \\
&= (a^*b^*)^{\circ\bullet *}ab \\
&= (a^{*\circ\bullet}+b^{*\circ\bullet})^*ab \\
&= (a^{\circ\bullet}+b^{\circ\bullet})^*ab \\
&= (a+b)^*ab
\end{aligned}
$$

The relation between the star-normal form and the elimination of redundant links in ZPC algorithm is resumed by the following proposition:

Proposition 8 (Ziadi et al. [14]). *If E is in star-normal form then*:

$$\delta(M_E)=\left(\biguplus_{v\in T(E)} \Delta_v\right) \uplus (\{s_I\}\times First(E)).$$

3.4. Modified ZPC algorithm

In order to parallelize ZPC algorithm we modify it as follows (Fig. 3):

- 1.a Construct the tree $T(E)$.
- 1.b Compute the star-normal form $E^{\bullet}$ of E.
- 2. For each node v in the tree $T(E^{\bullet})$ compute $Null(E^{\bullet}_v)$.
- 3. Construct the forests $TL(E^{\bullet})$ and $TF(E^{\bullet})$.
- 4. For each node v in $T(E^{\bullet})$ compute the follow links representing Δ_v.

4. Parallelization of modified ZPC algorithm

In this section we suppose that the expression E verifies *SNF* condition. Let s be the size of E. We show that in this case modified ZPC algorithm can be parallelized in $O(\log s)$ time using $O(s/\log s)$ processors, which is an optimal result. Steps 1a, 2, 3 and 4 of modified ZPC algorithm are parallelized in the following way.

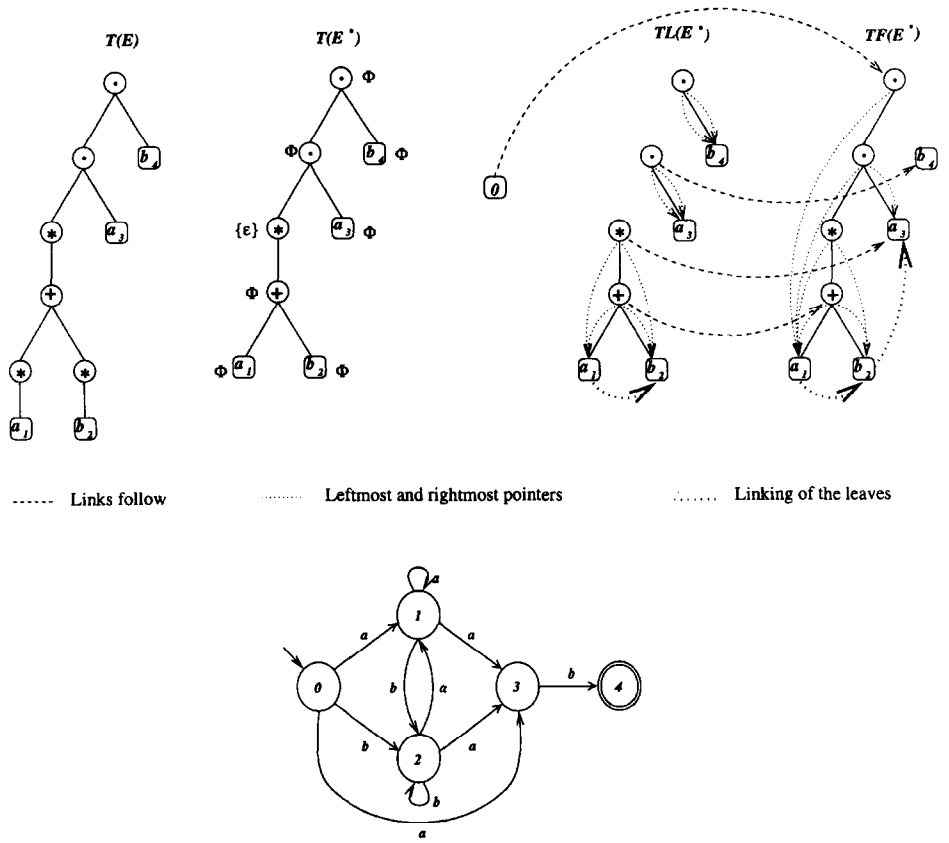


Fig. 3. Modified ZPC algorithm for the expression $E=(a^*+b^*)^*\cdot a\cdot b$.

4.1. Step 1a: $T(E)$ construction

We make use of the optimal parallel algorithm due to Bar-On and Vishkin [1], which computes the syntax tree of an arithmetic expression of size s in $O(\log s)$ time using $O(s/\log s)$ processors. This algorithm works on a completely bracketed expression; it means that each subexpression is enclosed between a left and a right bracket (these brackets form a pair). Bar-On and Vishkin claim that an expression can be converted to an equivalent completely bracketed expression in $O(\log s)$ optimal time using $O(s/\log s)$ processors. The syntax tree $T(E)$ is constructed from the sequence P_E of brackets of the completely bracketed expression E. Each node v of $T(E)$ is associated with the pair of brackets enclosing the subexpression E_v (see Fig. 4). In order to construct $T(E)$, Bar-On and Vishkin define the function *match* which associates to every position i in P_E the position j such that the brackets $P_E(i)$ and $P_E(j)$ form a pair. For example, in the Fig. 4, we have $match(4)=17$ and $match(5)=10$. The function *match* is computed in time $O(\log s)$ with $O(s/\log s)$ processors and so $T(E)$ construction is achieved with the same complexity.

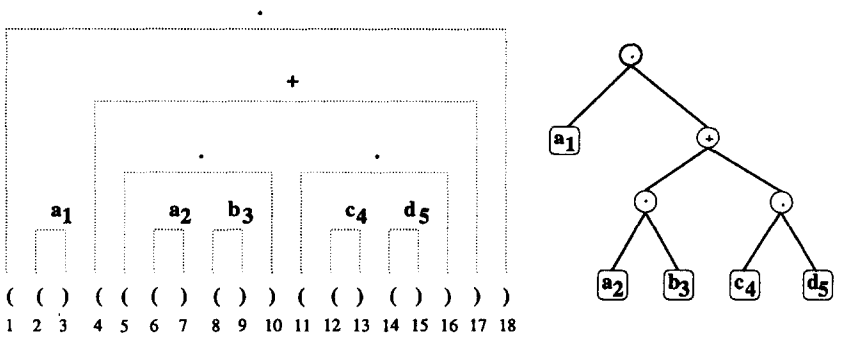


Fig. 4. Syntax tree of the expression $E = a \cdot (a \cdot b + c \cdot d)$ and its corresponding sequence of brackets.

4.2. Step 2: Null computation

We adopt the parallel algorithm that Gibbons and Rytter [4] give for the evaluation of an algebraic expression. This algorithm is optimal: its time complexity is $O(\log s)$ using $O(s/\log s)$ processors.

4.3. Step 3: TL(E) and TF(E) construction

As mentioned in the previous section, $TL(E)$ is deduced from $T(E)$ by deleting every link between a node v labeled "$\cdot$" to its left child v_l, if $Null(E_{v_r}) = \emptyset$. Dually, $TF(E)$ is deduced from $T(E)$ by deleting every link between a node v labeled "$\cdot$" to its right child v_r, if $Null(E_{v_l}) = \emptyset$. Links deletion is easily performed in $O(1)$ time using $O(s)$ processors. In both forests, every node v points to the leftmost leaf (*leftmost*(v) pointer) and to the rightmost leaf (*rightmost*(v) pointer) in the tree rooted at v. The computation of *leftmost* and *rightmost* pointers requires a particular attention. We present an optimal algorithm for achieving this step. We describe it in the case of the computation of *leftmost* pointers in $TL(E)$ and we outline adjustments for computation of *rightmost* pointers and for computation in $TF(E)$. Let us first illustrate it on an example.

Example 3. Let $E = ((a)(((a)(b)) + ((c)(d))))$. We compute $T(E)$ and $TL(E)$.

It appears that leftmost pointers cannot be easily computed from a standard traversal of $TL(E)$. So, we make a copy $T'(E)$ of $T(E)$ in which we permute the subtrees rooted at v_l and v_r every time a link from a node v to its left child v_l is deleted in $TL(E)$ (Fig. 5). Now, we can use the trace of the top-down right suffix traversal of $T'(E)$ to draw leftmost pointers. In our example, that trace is equal to $a_1 c_4 d_5 \cdot a_2 b_3 \cdot + \cdot$. The last leaf appearing before a node v in the trace is the leftmost leaf of v.

trace	a_1	c_4	d_5	$\cdot$	a_2	b_3	$\cdot$	$+$	$\cdot$
leftmost	a_1	c_4	d_5	d_5	a_2	b_3	b_3	b_3	b_3

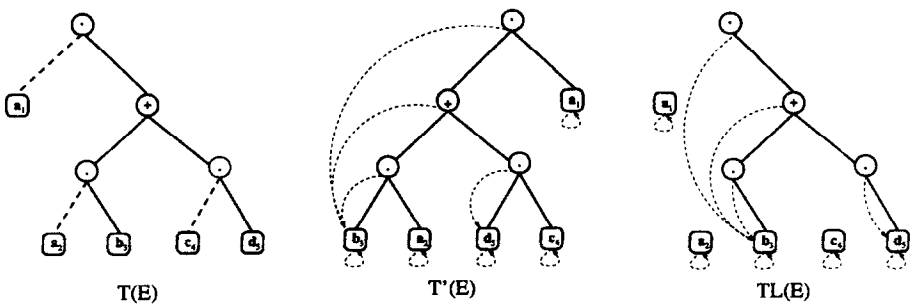


Fig. 5. Computing *leftmost* in $TL(E)$ via $T'(E)$.

We now detail the two substeps of this algorithm (computation of *trace*, computation of *leftmost* pointers).

4.3.1. Substep 1: Computation of trace

Subtrees permutations are not physically realized; they are rather performed via a marking of the opening brackets of the sequence P_E. For a node v, if its link to v_l is deleted in $TL(E)$, the opening bracket of v_l (resp. v_r) is marked by r (resp. l); otherwise the opening bracket of v_l (resp. v_r) is marked by l (resp. r). By convention, the opening bracket of the root is marked by l.

In our example we have $P_E = (\,)(((\,)(\,))((\,)(\,))))$ and after subtrees permutations we get $P_E = ({}_l({}_r)({}_l({}_l({}_r)({}_l))({}_r({}_r)({}_l)))))$.

Let $rank(v)$ be the rank of the node v in a top-down right suffix traversal of $T'(E)$. Our aim is to compute $rank(v)$ without constructing P'_E nor $T'(E)$. Let $A[i]$ be the position of the opening bracket $P_E[i]$ in the sequence of opening brackets of $T'(E)$. We first compute $A[i]$ with the help of *brother* function defined as follows:

$$brother(i) = \begin{cases} \text{match}(\text{match}(i)+1)+1 & \text{if } P_E[match(i)+1] = ')', \\ \text{match}(i)+1 & \text{otherwise.} \end{cases}$$

Then we get $rank(v)$ by computation of prefix sums of the array A. We finally use $rank()$ to compute the *trace* of a top-down right suffix traversal of $T'(E)$. The resulting procedure is

```
begin
  Computation of positions in the sequence of opening brackets P'_E
  forall 1 ≤ i ≤ |P_E| pardo
    case P_E[i] of
      (_l: begin A[i]: = 1; A[match[i]]: = −A[i]; end;
      (_r: begin A[i]: = (match[brother[i]] − brother[i] + 1)/2;
        A[match[i]]: = −A[i]; end;
    end;
  Computation of ranks in T'(E)
```

```
  prefix-sums of A
  Collecting trace
  forall 1 ⩽ i ⩽ |P_E| pardo trace[i] := node[i];
end
```

where *node*[*i*] is the symbol associated with brackets $P[i]$ and $P[match[i]]$.

Example 4. We consider the expression $E = a(ab + cd)$. Prefix sums on A are denoted by $\Sigma_p(A)$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P_E	$($	$($	$)$	$($	$($	$($	$)$	$($	$)$	$)$	$($	$($	$)$	$($	$)$	$)$	$)$	$)$
node	$\cdot$	a_1	a_1	$+$	$\cdot$	a_2	a_2	b_3	b_3	$\cdot$	$\cdot$	c_4	c_4	d_5	d_5	$\cdot$	$+$	$\cdot$
A	1	8	−8	1	1	2	−2	1	−1	−1	4	2	−2	1	−1	−4	−1	−1
$\Sigma_p(A)$	1	9	1	2	3	5	3	4	3	2	6	8	6	7	6	2	1	0
trace	a_1	c_4	d_5	$\cdot$	a_2	b_3	$\cdot$	$+$	$\cdot$									

Our claim is that this procedure correctly computes the trace of a top-down right suffix traversal of $T'(E)$. The proof comes from Lemmas 1 and 2.

Lemma 1. $\forall i, 1 \leqslant i \leqslant |P_E|, \sum_{k=i}^{match(i)} A[k] = 0$.

Lemma 2. *Let v be a node and i be the position of the opening bracket in P_E associated with the node v in $T'(E)$. We have*: $rank(v) = \sum_{k=1}^{i} A[k]$.

Proof. We shall note $r(i) = \sum_{k=1}^{i} A[k]$. Lemma 2 is verified for $T'(E)$ root since $rank(v) = r(1) = 1$. Consider node v at position i and let us assume that Lemma 2 holds for nodes whose positions are less or equal to i. Let g and d be positions in P_E of opening brackets, respectively, associated with v_l and v_r in $T'(E)$ ($g = brother[d]$). We are going to show that Lemma 2 holds for v_r and v_l. There are two possibilities:

Case 1: $g \leqslant d$ In this case $g = i + 1$. By induction $r(i) = rank(v)$. $T'(E)$ traversal is such that $rank(v_l) = rank(v) + 1$. So we get $rank(v_l) = r(i) + 1$. As $A[g] = 1$ (by initialization) and $g = i + 1$, we can write $rank(v_l) = \sum_{k=1}^{g} A[k]$.

On the other hand, $T'(E)$ traversal is such that $rank(v_r) = rank(v) + t(v_l) + 1$, where $t(v_l)$ is the number of nodes in the subtree rooted at v_l. As $t(v_l)$ is equal to the number of pairs of brackets inside the pair associated to v_l (including this pair), it is easy to verify that we have $t(v_l) = (match[g] - g + 1)/2$. Therefore, it comes $rank(v_r) = rank(v) + (match[g] - g + 1)/2 + 1$.

Let us consider now $r(d) = \sum_{k=1}^{d} A[k]$. We have

$r(d) = (\sum_{k=1}^{i} A[k]) + (\sum_{k=i+1}^{d-1} A[k]) + A[d]$.

As $g = i + 1$ and $match[g] = d - 1$, Lemma 1 implies $\sum_{k=i+1}^{d-1} A[k] = 0$. Moreover, by initialization $A[d] = 1 + (match[g] - g + 1)/2$. So we have $r(d) = r(i) + (match[g] - g + 1)/2 + 1$.

By induction $rank(v)=r(i)$, so we have $rank(v_l)=r(d)$.

Case 2: $g>d$.

The demonstration is similar as in the first case. □

4.3.2. Substep 2: Computation of leftmost pointers

Computation of *leftmost* pointers is achieved via a marking of *trace* based on the following lemma:

Lemma 3. *Let T be a syntax tree and M be a marking of the trace of a right suffix traversal of T. We shall assume T is not void. Suppose internal nodes are initially marked by* 0 *and leaves by* 1. *Then prefix sums of M give the same mark to nodes having the same leftmost leaf.*

Proof. The initial value of M can be seen as a word of the language $(10^*)^+$. For each subexpression 10^* of such a word, nodes marked by 0 (if any) have the same leftmost leaf, which is the node marked by 1. All of these nodes (and only them) will be identically marked by prefix sums of M. □

We associate a processor to each element of *trace*. Each processor determines its leftmost leaf by performing the following sequence:

```
begin
   Initialization of marking B
   forall 1 ⩽ i ⩽ s pardo
      if trace[i] is a letter then B[i]:=1 else B[i]:=0;
   Computation of marking B
   prefix-sums of B
   Collecting leftmost pointers
   forall 1 ⩽ i ⩽ s pardo
      if trace[i] is a letter then leaf[B[i]]:=trace[i];
   forall 1 ⩽ i ⩽ s pardo leftmost[i]:=leaf[B[i]];
end
```

We shall complete the description of Step 3 by the following remarks:

1. The computation of the *rightmost* pointers can be done in a similar way by calculating the order of the opening brackets in a prefix traversal of $T(E)$ and by achieving the prefix sums in B from right to left.
2. The computation of *leftmost* and *rightmost* pointers in the forest $TF(E)$ is deduced from the construction presented on $TL(E)$.
3. The computation of linking of the leaves inside a same tree, works as follows. Associate a processor to each node v in $T(E)$ which performs the following sequence:

```
begin
   join rightmost(v_l) to leftmost(v_r) in TL(E)
   join rightmost(v_l) to leftmost(v_r) in TF(E)
end
```

4.3.3. Conclusion of Step 3:

The computation of $TL(E)$ and $TF(E)$ (*leftmost* and *rightmost* pointers, leaves linking) can be achieved in $O(\log s)$ time using $O(s/\log s)$ processors (same complexity as for prefix-sums).

4.4. Step 4: Computation of the follow links representing Δ_v

In order to create the links representing Δ_v, to each node v we associate a processor which performs the following sequence:

```
begin
   case v of
      · : join v_l in TL(E) to v_r in TF(E)
      * : join v_s in TL(E) to v_s in TF(E)
   end
end
```

This sequence is achieved in constant time using $O(s)$ processors.

4.5. Conclusion

The result of these successive steps is illustrated by Fig. 6. With respect to the complexity of each step, we can state the following theorem:

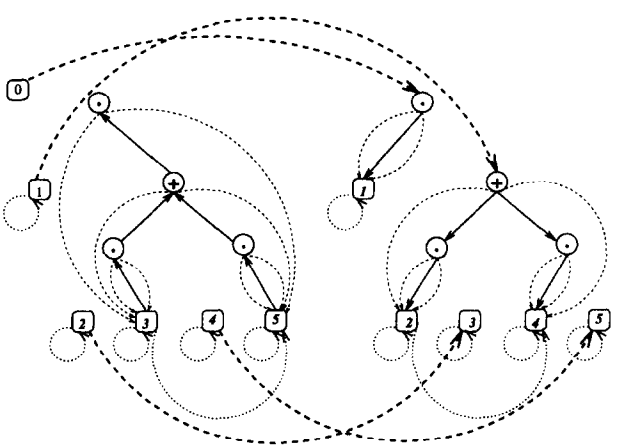


Fig. 6. $TL(E)$ and $TF(E)$ for $E = a(ab + cd)$.

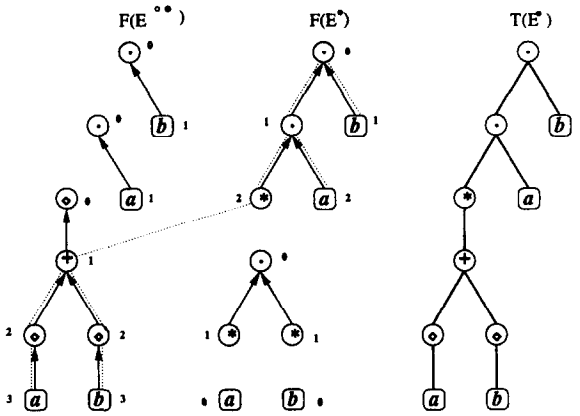


Fig. 7. The forests associated to $(a^*b^*)^*ab$.

Theorem 3. *Let E be a regular expression of size s, verifying SNF condition. The Glushkov automaton of E represented by the forest of Lasts, the forest of Firsts and the follow links can be computed by an optimal parallel algorithm of time complexity* $O(\log s)$ *using* $O(s/\log s)$ *processors on a CREW-PRAM.*

5. Computation of the star-normal form

Let E be a regular expression and $T(E)$ its syntax tree. We assume that the function *father*() has been computed on $T(E)$. The problem is the following: given $T(E)$, build $T(E^\bullet)$, the syntax tree of the star-normal form $E^\bullet$ of E. We consider the forests $F(E^\bullet)$ and $F(E^{\circ\bullet})$ associated to the expressions $E^\bullet$ and $E^{\circ\bullet}$ (see Section 3) and constructed as follows:

(a) $F(E^\bullet)$ and $F(E^{\circ\bullet})$ are initialized by a copy of $T(E)$,

(b) The computation of $E^\bullet$ according to Theorem 3 partitions $F(E^\bullet)$ (resp. $F(E^{\circ\bullet})$) into subtrees inside which internal nodes are evaluated without jumping in $F(E^{\circ\bullet})$ (resp. $F(E^\bullet)$). We modify the function *father*() in both forests in order to represent these partitions. Moreover, in $F(E^{\circ\bullet})$, we replace the operator "$\cdot$" by the operator "$+$" with respect to the definition of $(G \cdot H)^{\circ\bullet}$ and we replace the operator "$*$" by the identity operator "$\diamond$" with respect to the definition of $(E^*)^{\circ\bullet}$. Fig. 7 gives the representation of the function *father*() in the forests $F(E^\bullet)$ and $F(E^{\circ\bullet})$ for the expression $E = (a^*b^*)^*ab$.

We denote by $v(E^\bullet)$ (resp. $v(E^{\circ\bullet})$) the node of $F(E^\bullet)$ (resp. $F(E^{\circ\bullet})$) corresponding to the node v of $T(E)$. We associate a processor to every node v in $T(E)$ and each processor performs the following sequence:

```
begin
  case symbol(v) of
    ·: case (Null(E_vl),(Null(E_vr)) of
```

```
        (∅,∅)      : begin father(v_l(E°•)) :=nil; father(v_r(E°•)):=nil; end;
        ({ε},∅)    : father(v_l(E°•)):= nil;
        (∅,{ε})    : father(v_r(E°•)):= nil;
      end
  *:  begin father(v_s(E•)):= nil; symbol(v(E°•)):='◇' end;
 end;
end
```

This construction is achieved in constant time using $O(s)$ processors.

The problem is now to deduce $T(E^\bullet)$ from $F(E^\bullet)$ and $F(E^{\circ\bullet})$. We denote by $v^\bullet$ the node of $T(E^\bullet)$ corresponding to the node v of $T(E)$. We associate a processor to each node v of $T(E)$. Each processor must decide whether $v^\bullet = v(E^\bullet)$ or $v^\bullet = v(E^{\circ\bullet})$. A sequential solution to this problem would be typically recursive. We need some technique to make a local decision. This technique is illustrated on a simplified case where trees are replaced by lists, as shown in Fig. 8.

Let $L_a = (a_0, a_1, \ldots, a_n)$ and $L_b = (b_0, b_1, \ldots, b_n)$ be two lists. We arbitrarily suppress some links in each list, with respect to the condition:

$$\forall i,\ 0 \leqslant i < n,\ (next(a_i) = nil) \wedge (next(b_i) = nil) = false$$

L_a and L_b are now collections of lists. Our aim is to compute the list $L = (c_0, c_1, \ldots, c_n)$ from L_a and L_b such that:

- $c_0 = a_0$
- $\forall i,\ 0 \leqslant i < n$, if $c_i = a_i$

$$\text{then } c_{i+1} = \begin{cases} a_{i+1} & \text{if } next(a_i) \neq nil \\ b_{i+1} & \text{otherwise} \end{cases} \qquad \text{else } c_{i+1} = \begin{cases} b_{i+1} & \text{if } next(b_i) \neq nil \\ a_{i+1} & \text{otherwise} \end{cases}$$

Let x be an element of list. We call rank of x the distance $d(x)$ of x to the head of the list. The assignment of a_i or b_i to c_i can be locally decided from $d(a_i)$ and $d(b_i)$, using the property: $d(a_i) \geqslant d(b_i) \Leftrightarrow c_i = a_i$.

This technique extends to the case of forests (see Fig. 7); $d(v)$ is the distance of node v to the root of the tree it belongs to; $d(v)$ is computed by doubling technique, in $O(\log s)$ time using $O(s)$ processors. Each processor performs the following sequence:

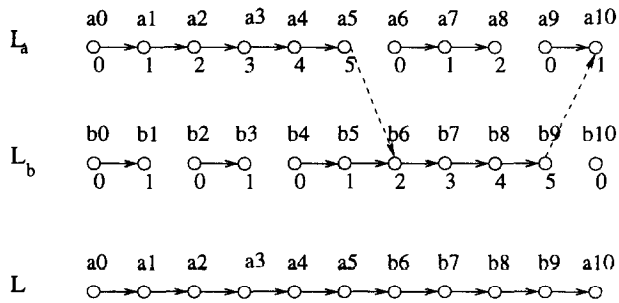


Fig. 8. "Lists stitching".

```
begin
   if d(v(E°•)) > d(v(E•))
   then symbol(v•):= symbol(v(E°•))
   else symbol(v•):= symbol(v(E•));
end
```

Finally we can state the following theorem:

Theorem 4. *$T(E^\bullet)$ can be computed from $T(E)$ in time $O(\log s)$ using $O(s)$ processors on a CREW-PRAM.*

6. Computation of the sets *Follow*(E,x)

In this section we assume that E is in star-normal form. We show that, in this case, it is possible to compute the set $Follow(E,x)$, for $x \in Pos(E)$, as a linked list L_x of states in time $O(\log s)$ using $O(s/\log s)$ processors in a CREW-PRAM model. The position x is also a leaf of $TL(E)$. We consider the nodes $\lambda_1, \lambda_2, \ldots, \lambda_m$ which are ancestors of x in the tree containing x, and which are heads of a follow link. Let us denote $\Phi_x = \{\phi_1, \phi_2, \ldots, \phi_m\}$ the set of associated tails. As E is in star-normal form, the follow links going from the ancestors of x are such that the sets of positions of the subtrees rooted at $\phi_1, \phi_2, \ldots, \phi_m$ are disjoint sets. Thus, the list L_x can be computed by the following algorithm:

```
begin
1  Construction of the list Φx = {φ1, φ2, ..., φm}
2  forall 1 ⩽ i ⩽ m − 1 pardo
3      join rightmost(φi) to leftmost(φi+1)
4  join Lx to leftmost(φ1)
end
```

Let us analyze the complexity of this algorithm. First we show that Step 1 can be achieved in time $O(\log s)$ using $O(s/\log s)$ processors. We shall make use of P_E, the sequence of brackets associated to E. Let i be the position in P_E of the opening bracket associated to the node x, and r be the position of the opening bracket associated to the root of the tree of $TL(E)$ containing x. We shall assume that every opening bracket is labeled with the rank of the corresponding node in $T(E)$. Let us consider the subsequence $S = P_E[r] \ldots P_E[i]$. The reduced sequence [4] S' of S can be obtained by deleting the brackets $P_E[j]$ such that $j \in [r,i]$ and $match[j] \in [r,i]$. For example, if $S =)(\,)\,)(($, then $S' =)((\,.$ S' can be computed in time $O(\log s)$ using $O(s/\log s)$ processors [4]. Let us remark that S' is exactly the sequence of opening brackets associated with the ancestors of x. Among these ancestors we eliminate those which are not heads of follow links and then we compute Φ_x, in time $O(\log s)$ using $O(s/\log s)$ processors.

It is easy to see that Steps 2–4 can be achieved in time $O(\log s)$ using $O(s/\log s)$ processors. So the computation of all the sets $Follow(E,x)$ can be done in time $O(\log s)$ using $O(s^2/\log s)$ processors in a CREW-PRAM.

Theorem 5. *Let E be a regular expression of size s. Glushkov automaton associated to E can be computed in time $O(\log s)$ using $O(s^2/\log s)$ processors in a CREW-PRAM.*

7. Conclusion

We have described an optimal parallel algorithm to compute ZPC representation of the star-normal form of an expression and an efficient algorithm to compute the star-normal form of an expression. They combine in an efficient algorithm ($O(\log s)$ time using $O(s)$ processors) to compute the ZPC representation of an expression. We do not know whether there exists an optimal algorithm to compute the star-normal form of an expression, but we provide an optimal algorithm to convert the ZPC representation of an expression verifying *SNF* condition into a table of transitions ($O(\log s)$ time using $O(s^2/\log s)$ processors). Thus we finally get an optimal algorithm to convert a regular expression into its Glushkov automaton, in $O(\log s)$ time using $O(s^2/\log s)$ processors.

References

[1] I. Bar-On, U. Vishkin, Optimal parallel generation of a computation tree form, ACM Transactions on Programming Languages and Systems, 1985, pp. 348–57.

[2] A. Brüggemann-Klein, Regular expressions into finite automata, Theoret. Comput. Sci. 120 (1993) 197–213.

[3] C.-H. Chang, R. Paige, New theoretical and computational results for regular languages, in: Apostolico, Crochemore, Galil, and Manber (Eds.), Lecture Notes in Computer Science, vol. 644, 3rd Ann. Symp. on Combinatorial Pattern Matching Proc., Springer, Berlin, 1992, pp. 88–108.

[4] A.M. Gibbons, W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, Cambridge, MA, 1988.

[5] V.M. Glushkov, The abstract theory of automata, Russian Math. Surveys 16 (1961) 1–53.

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

[7] R. McNaughton, H. Yamada, Regular expression and state graphs for automata, IRA Trans. Electron. Comput. EC-9 1 (1960) 39–47.

[8] J.-L. Ponty, D. Ziadi, J.-M. Champarnaud, A new quadratic algorithm to convert a regular expression into an automaton, in: D. Raymond, D. Wood (Eds.), Lecture Notes in Computer Science, First Workshop on Implementing Automata, WIA'96 London–Ontario, Springer, Berlin, 1997, to appear.

[9] W. Rytter, A note on parallel transformations of regular expressions to nondeterministic finite automata, Inform. Process. Lett. 31 (1989) 103–109.

[10] R. Sedgewick, Algorithms, Addison-Wesley, Reading, MA, 1983.

[11] K. Thompson, Regular expression search algorithms, Common ACM 11(6) (1968) 419–422.

[12] B. Watson, Taxonomies and toolkits of regular language algorithms, CIP-DATA Koninklijke Bibliotheek, Den Haag, Ph.D. Thesis, Eindhoven University of Technology, 1995.

[13] D. Ziadi, Algorithmique parallèle et séquentielle des automates, Thèse de doctorat, Université de Rouen, 1996.

[14] D. Ziadi, J.-L. Ponty, J.-M. Champarnaud, Passage d'une expression rationnelle à un automate fini non-déterministe, Bull. Belg. Math. Soc. 4 (1997) 177–203.