• 1400012081
còpia 1

The design of a parallel algorithm to solve the word problem for the free partially commutative groups

Joaquim Gabarró

Report LSI–91–39

FACULTAT D'INFORMÀTICA
BIBLIOTECA
R. 9.173 15 OCT. 1991

The design of a parallel algorithm to solve the word problem for the free partially commutative groups

J. Gabarró *

Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo 5 , 08028 Barcelona
Spain

Schloβ Dagstuhl / **Draft** / October 10 1991

Abstract. We develop a parallel algorithm to solve the word problem for free partially commutative groups. These groups were introduced by C. Wrathall to generalize free groups. We represent the elements of these groups as a certain type of acyclic labeled graphs called dependency graphs. These graphs were introduced by A.Mazurkiewicz to model concurrent systems.

- First we study the parallel complexity of some basic problems arising in the study of dependency graphs. Such as correctness, isomorphism and relations with traces. Parallel algorithms are developed to solve all of them.
- Second we consider the combinatorial properties of free partially commutative groups. To do this we associate to every group a rewriting system over dependency graphs. Finally we apply all these ideas to solve in parallel the word problem.

Special emphasis is given in the design of the algorithms. The modular approach is widely used to obtain readable programs. It seems that many of the structuring techniques developed in sequential programming can be used directly in the PRAM context.

Keywords: PRAMS, data parallel algorithms, modular design, trace theory, free groups, free partially commutative groups, word problems, parallel complexity classes, NC^*, NL^*, AC^0, complete problems.

* Research supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

The parallel complexity of some dependency graphs problems.

We would like to develop a parallel algorithm to solve word problem for the free partially commutative groups. To do this we need:

- Basic facts on complexity classes in the line of [BDG88] and [BDG90]. The fundamental paper on parallel complexity classes is [Co85]. In [ABGS91] we present how to reason, in terms of complexity theory, about sequential and parallel programs.
- Elements of partially commutative monoids [Ma88] and free partially commutative groups [Wr88].
- How to write data parallel programs and reason about them. Examples of data parallel algorithms are in [HS86]. In [GG91] it is introduced an axiomatic way to reason about these programs. It seems that the design of data parallel and sequential algorithms is very close.

Let us recall some basic concepts on traces and dependency graphs. A basic reference is [Ma88]. Given a finite set of events Σ and a symmetric and irreflexive relation $\theta \subseteq \Sigma \times \Sigma$ we call the pair (Σ, θ) concurrent alphabet. The relation $\overline{\theta} = \Sigma \times \Sigma \setminus \theta$ is called *conflict relation*. The elements of the quotient monoid Σ^*/θ are called traces. Sometimes is better to represent traces by graphs, which makes explicit the ordering of symbol occurrences within the traces.

Dependency graphs. A triple $\gamma = (V, R, \varphi)$ is a dependency graph over (Σ, θ) when V is a finite set called the nodes of γ, $R \subseteq V \times V$ is the set of arcs and $\varphi : V \to \Sigma$ is the labelling, if the following conditions holds:

- *Acyclicity* written as $R^+ \cap \{(v,v) \mid v \in V\} = \emptyset$.
- *Dependence connectivity* given by

$$R \cup R^{-1} \cup \{(v,v) \mid v \in V\} = \{(v,v') \in V \times V \mid (\varphi(v), \varphi(v')) \in \overline{\theta}\}.$$

The number of nodes of γ written as $\#(\gamma)$ is $\#(V)$. We write $\Gamma(\Sigma, \theta)$ for the set of dependency graphs. Two dependency graphs γ and δ are isomorphic, $\gamma \simeq \delta$, when there exists a bijection of nodes preserving labelling and arc connections. Given a word $w = x_1 \ldots x_n \in \Sigma^*$ we define the dependency graph associated to w as $d(w) = (V_w, R_w, \varphi_w)$ where $V_w = \{1, \ldots, n\}$, $\varphi_w(i) = x_i$ and $R_w = \{(i,j) \mid (i < j) \wedge (x_i, x_j) \in \overline{\theta}\}$. A linearization of γ is a word w such that $\gamma \simeq d(w)$.

Composition. Given $\gamma = (V_1, R_1, \varphi_1)$ and $\delta = (V_2, R_2, \varphi_2)$ with $V_1 \cap V_2 = \emptyset$, the composition $\gamma \circ \delta = (V, R, \varphi)$ is defined as:

- $V = V_1 \cup V_2$.
- $R = R_1 \cup R_2 \cup \{(v_1, v_2) \in V_1 \times V_2 \mid (\varphi_1(v_1), \varphi_2(v_2)) \in \overline{\theta}\}$.
- $\varphi = \varphi_1 \cup \varphi_2$.

Denoting the empty graph as $\lambda = (\emptyset, \emptyset, \emptyset)$, we have that $(\Gamma(\Sigma, \theta), \circ, \lambda)$ is a monoid. Identifying dependency graphs with classes in Σ^*/θ we get an isomorphism between Σ^*/θ and $\Gamma(\Sigma, \theta)$.

Hiding. Let us consider how to hide some events $V \subseteq V'$ in a dependency graph $\gamma = (V', R, \varphi)$. The result is written as $\gamma \setminus V = (V, R \cap V \times V, \varphi \mid V)$. Given a graph and a vertex v of this graph the set of ancestors and descendants [Ja88] are:

$$< v >^* = \{v' \mid v' \xrightarrow{*} v\} \qquad \Delta^*(v) = \{v' \mid v \xrightarrow{*} v'\}$$

We adapt these ideas to deal with a dependency graph $\delta = (V, R, \varphi)$ defining, for $v \in V$:

- The dependency graph $< v, \delta >^* = \delta \setminus \overline{< v >^*}$ determined by taking all the ancestors of v in γ.
- The dependency graph $\Delta^*(v, \delta) = \delta \setminus \overline{\Delta^*(v)}$ obtained by taking all the descendants of v in γ.

The figure 1 give us examples of basic operations in $\Gamma(\Sigma, \theta)$.

Problem 1: The dependency graphs problem for concurrent alphabets, called *DEPENDENCY_ GRAPHS_ CORRECTNESS* is the following:

Input: A concurrent alphabet (Σ, θ) and $\gamma = (V, R, \varphi)$.

Output: It is true that $\gamma \in \Gamma(\Sigma, \theta)$?.

Lemma 2: The problem *DEPENDENCY_ GRAPHS_ CORRECTNESS* is NL^* complete.

Proof. First let us prove that:

$$DEPENDENCY_GRAPH_CORRECTNESS <_{NC^1} TRANSITIVE_CLOSURE.$$

We represent the circuit given the NC^1 reduction as a PRAM program (figure 2). This program can be easily unfolded to give us a circuit.

Let us prove completeness. Let M be a log space Turing machine working in time n^k. We assume that M has a clock, in such a way that every configuration has a well defined time t, we write $c(t)$. The initial configuration is $c_i(0)$ and the unique final configuration is $c_f(n^k)$. The only possible transitions are from time t to time $t+1$.

Let us reduce this problem to *DEPENDENCY_GRAPH_CORRECTNESS* using NC^1 reductions. Given $< M, c_i(0), c_f(n^k) >$ let us construct $< (\Sigma_M, \theta_M), \gamma_M >$ such that

$$c_i(0) \overset{*}{\vdash} c_f(n^k) \Longleftrightarrow \gamma \notin \Gamma(\Sigma_M, \theta_M)$$

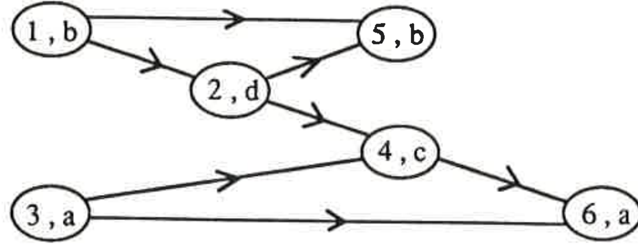
- The concurrent alphabet is in bijection with the configurations. It is written as

$$\Sigma_M = \{x_{c(t)} \mid c(t) \text{ is a configuration } \}$$

All the letters can be easily constructed in parallel assigning a different processor to every letter.

Conflicts	
a	a , c
b	b , d
c	a , c , d
d	b , c , d

w = b d a c b a

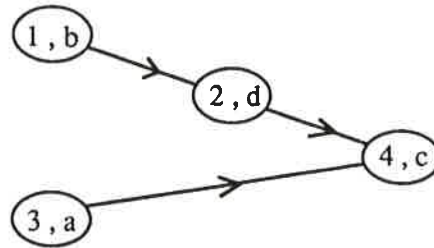


Dependency graph d (w)

$<4>^* = \{1,2,3,4\}$

$\overline{<4>^*} = \{5,6\}$

$<4, d(w)>^* = d(w) \setminus \{5,6\}$



$\Delta^*(2) = \{2,4,5,6\}$

$\overline{\Delta^*(2)} = \{1,3\}$

$\Delta^*(2, d(w)) = d(w) \setminus \{1,3\}$

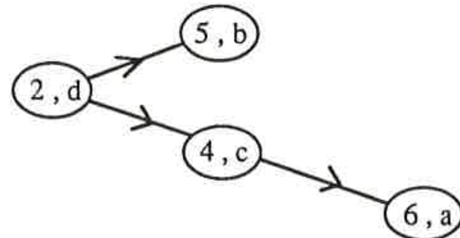


Figure 1. Basic operations in $\Gamma(\Sigma, \theta)$.

- The conflict relation $\overline{\theta_M}$ can also be easily constructed in parallel. To do this we assign a processor to a couple of letters.

$$\begin{aligned}\overline{\theta_M} = &\{(x_{c(t)}, x_{c(t)}) \mid c(t) \text{ is a configuration}\} \\ &\cup \{(x_{c(t)}, x_{c(t+1)}) \mid c(t) \vdash c(t+1)\} \cup \{(x_{c(t+1)}, x_{c(t)}) \mid c(t) \vdash c(t+1)\} \\ &\cup \{(x_{c_f(n^k)}, x_{c_i(0)}), (x_{c_i(0)}, x_{c_f(n^k)})\}.\end{aligned}$$

```
procedure Correct_Dependency_Graph
    ( θ   : array [x . . y , x . . y] of proc of bool;
      R   : array [1 . . n , 1 . . n] of proc of bool;
      φ   : array [1 . . n] of proc of [x . . y]
    )     : bool;

from NL import Transitive_Closure
    ( M   : array [1 . . n , 1 . . n] of proc of bool
    )     : array [1 . . n , 1 . . n] of proc of bool;
    { returns the transitive closure of M }

var cycle       : array [1 . . n] of proc of bool;
    i, j        : integer;
    acyclicity  : bool;
    dep         : array [1 . . n , 1 . . n] of proc of bool;
    dep_connect : bool;
begin
    { test the acyclicity using as oracle the transitive closure }
    for all 0 ≤ i ≤ n do in parallel
        cycle[i] := Transitive_Closure (R[i, i])
    end ;
    acyclicity := ⋀_{1≤i≤n} ¬cycle[i];
    { test de dependence connectivity }
    for all 1 ≤ i, j ≤ n do in parallel
        dep[i, j] := ((R[i, j] ∨ R[j, i] ∨ (i = j)) ≡ (¬θ[φ[i], φ[j]])
    end ;
    dep_connect:= ⋀_{1≤i,j≤n} dep[i, j];
    Correct_Dependency_Graph:= acyclicity ∧ dep_connect
end Correct_Dependency_Graph.
```

Figure 2. Testing correctness on dependency graphs.

When this relation is constructed we construct $\theta = \Sigma_M \times \Sigma_M \setminus \overline{\theta_M}$ in parallel.

- The dependency graph $\gamma_M = (V_M, R_M, \varphi_M)$ is

$$V_M = \{v_{c(t)} \mid c(t) \text{ is a configuration}\}.$$
$$R_M = \{(v_{c(t)}, v_{c(t+1)}) \mid c(t) \vdash c(t+1)\} \cup \{(v_{c_f(n^k)}, v_{c_i(0)})\}.$$
$$\varphi_M(v_{c(t)}) = x_{c(t)}.$$

When $c_i(0) \overset{*}{\vdash} c_f(n^k)$ we have $\Delta^*(c_i(0)) \cap < c_f(n^k) >^* \neq \emptyset$ and R_M has the following cycle and γ_M cannot be a dependency graph.

$$(v_{c_i(0)}, \ldots, v_{c_f(n^k)}, v_{c_i(0)})$$

Reciprocally $c_f(n^k)$ does not follows from $c_i(0)$ the graph R does not have any cycle and

$$\begin{aligned}
&R_M \cup R_M^{-1} \cup \{(v_{c(t)}, v_{c(t)}) \mid v_{c(t)} \in V_M\} = \\
&\{(v_{c(t)}, v_{c(t+1)}), (v_{c(t)}, v_{c(t+1)}) \mid c(t) \vdash c(t+1)\} \\
&\cup \{(v_{c_f(n^k)}, v_{c_i(0)}), (v_{c_f(n^k)}, v_{c_i(0)})\} \cup \{(v_{c(t)}, v_{c(t)}) \mid v_{c(t)} \in V_M\} = \\
&\{(v, v') \in V_M \times V_M \mid (\varphi_M(v), \varphi_M(v')) \in \overline{\theta_M}\}
\end{aligned}$$

therefore γ_M is a dependency graph. That is all. □

Given a dag G and a vertex v we call *rank* (v) the length of longest path arriving to v. We call *DAG_RANK* this problem. It is NC^1 reducible to *k_CONNECTIVITY* because:

$$\textit{rank}\ (v) = \max\{\max\{k \mid \text{connect}(v, v', k)\} \mid v' \in V\}.$$

Considering a Turing machine with a clock working in log space, we can reduce the problem of acceptance to *DAG_RANK*. Therefore *DAG_RANK* is NL^* complete under NC^1 reductions.

Problem 3: The equivalence for dependency graphs problem for concurrent alphabets, called *DEPENDENCY_ GRAPHS_ISOMORPHISM* is the following:

Input: A concurrent alphabet (Σ, θ) and $\gamma = (V, R, \varphi)$ and $\delta = (V', R', \varphi')$

Output: It is true that $\gamma, \delta \in \Gamma(\Sigma, \theta)$ and $\gamma \simeq \delta$?.

Lemma 4: Given $\gamma = (V, R, \varphi)$ and $\delta = (V', R', \varphi')$ in $\Gamma(\Sigma, \theta)$, the following conditions are equivalent:

- $\gamma \simeq \delta$.
- for all i such that $0 \le i \le \max\{\#(\gamma), \#(\delta)\}$ we have

$$\{\varphi(v) \mid v \in V, \textit{rank}\ (v) = i\} = \{\varphi'(v) \mid v \in V', \textit{rank}\ (v) = i\}$$

Proof. Let $\gamma = (V, R, \varphi)$ in $\Gamma(\Sigma, \theta)$ then $\textit{rank}\ (v) = \textit{rank}\ (v')$ implies $\varphi(v) \neq \varphi(v')$. Therefore the mapping $\textit{pairing} : V \to V'$ defined by

$$\textit{pairing}(v) = \{v' \in V' \mid \textit{rank}\ (v) = \textit{rank}\ (v') \wedge \varphi(v) = \varphi'(v')\}$$

give the desired isomorphism. □

Lemma 5: The problem *DEPENDENCY_ GRAPHS_ISOMORPHISM* is NL^* complete.

Proof. The program given in the figure 3 show us that the above problem belongs to NL^*.

To prove completeness it is enough to see that $\gamma \simeq \gamma$ is equivalent to $\gamma \in \Gamma(\Sigma, \theta)$. That concludes the proof. □

```
procedure Equivalent_Dependency_Graph
      (θ    : relation;
       γ    : labeled_graph;
       δ    : labeled_graph;
      )     : bool;

from NL* import Rank(...); Correct_Dependency_Graphs (...);
var parsing     : bool;
    i, j        : integer;
    pairing     : array [1 . . , 1 . . ] of proc of bool;
begin
    parsing:= Correct_Dependency_Graph(θ, γ) ∧ Correct_Dependency_Graph(θ, δ);
    if parsing then
        for all 0 ≤ i, j ≤ max{#(γ), #(δ)} do in parallel
            pairing[i, j] := (Rank[i] = Rank [j]) ∧ (φ(i) = φ'(j))
        end
    end
    Equivalent_Dependency_Graph := parsing ∧_C (⋀_i(⋁_j(pairing[i, j])))
end Equivalent_Dependency_Graph.
```

Figure 3. Testing isomorphism on dependency graphs.

Problem 6: Given a dependency graph how to obtain a linearization of it. We call this problem *DEPENDENCY_ GRAPHS_ to_TRACES*.

Input: A concurrent alphabet (Σ, θ) and $\gamma \in \Gamma(\Sigma, \theta)$.

Output: A word $w \in \Sigma^*$ such that $d(w) \simeq \gamma$.

Given a dependency graph $\gamma = (V, R, \varphi)$ there are many possible linearization (any topological sort is a possible linearization). Between all the solutions we choose the *Cartier Foata normal form* [CF69], given by the following fact (we write $w \overset{*}{\sim} w'$ to denote $d(w) \simeq d(w')$) [La79] :

For every word $w \in \Sigma^+$ exists a unique factorization $w \overset{*}{\sim} w_0 w_2 \ldots w_k$ such that:

(1) For every $0 \leq i \leq k$ the word w_i is a monomial of degree 1 on each of its letters (for every $x \in \Sigma$, $| w_i |_x \leq 1$) and any two letters of w_i commute.

(2) For every $0 \leq i < k$, each letter of w_{i+1} is in conflict with some letter of w_i. That means each letter of w_{i+1} either coincides or does not commute with some letter of w_i.

We write $Cartier_Foata\ (w) = w_0 w_2 \ldots w_k$. Given $\gamma = (V, R, \varphi)$ with $k = \max\{rank(v) \mid v \in V\}$ and $w \in \Sigma^+$ such that $\gamma \simeq d(w)$ we have [AR88]

$$\forall\, 0 \leq i \leq k \ :\ w_i = \{\varphi(v) \mid v \in V \wedge rank(v) = i\}$$

therefore

$$Cartier_Foata\ (w) = \{\varphi(v) \mid v \in V \wedge rank(v) = 0\} \cdots \{\varphi(v) \mid v \in V \wedge rank(v) = k\}.$$

```
procedure Cartier_Foata
      (γ    : labeled_graph
      )     : word;

from NL* import Rank(...);
var  k, i         : integer;
     w            : array [1 . . ] of proc of word;
begin
     k:= max {Rank(i) | 1 ≤ i ≤ #(γ)};
     for all 0 ≤ i, j ≤ k do in parallel
          w[i] := a word with letters {φ(j) | ∀ j : Rank[j] = i}
     end ;
     Cartier_Foata := w[1] ... w[k]
end Cartier_Foata.
```

Figure 4. Obtaining the Cartier Foata normal form.

This characterization can be worked out to obtain a parallel algorithm, figure 4 and we get the following result:

Lemma 7: The problem *DEPENDENCY_ GRAPHS_ to_TRACES* belongs to NL^*.

Problem 8: The problem *TRACES_ to_DEPENDENCY_ GRAPHS*, taking a word and outputting the dependency graph is:

Input: A concurrent alphabet (Σ, θ) and $w \in \Sigma^*$.

Output: The dependency graph $d(w)$.

Lemma 9: The problem *TRACES_ to_DEPENDENCY_ GRAPHS* belongs to AC^0.

Proof. All the sentences given in the figure 5 can be coded with operations like "count" or "enumerate". These two operations belongs to AC^0. That is all. □

The precedent lemmas give us an algorithm in NL^* to test trace equivalence $w \overset{*}{\sim} w'$. However it is possible to find a better one belonging to TC^0 to solve this problem [AG90].

Free partially commutative groups and reduction systems

Given a finite alphabet $\Sigma = \{x, y, z, \ldots\}$ we take a copy $\overline{\Sigma} = \{\overline{x}, \overline{y}, \overline{z}, \ldots\}$ and we write

$$\hat{\Sigma} = \Sigma \cup \overline{\Sigma}$$

Given an element $z \in \hat{\Sigma}$ we write $\overline{z}$ as usual. That means if $z = x \in \Sigma$ then $\overline{z} = \overline{x} \in \overline{\Sigma}$, but $z = \overline{x} \in \overline{\Sigma}$ then $\overline{z} = x \in \Sigma$. Given a concurrent alphabet (Σ, θ) we extend the commutation relation to deal with the elements of $\hat{\Sigma}$:

$$\hat{\theta} = \theta \cup \{(\overline{x}, y), (x, \overline{y}), (\overline{x}, \overline{y}) \mid (x, y) \in \theta\}$$

```
procedure Trace_Dependency_Graph
      (w    : word ;
      )     : labeled_graph;
var  x, y          : char ;
     i, j          : integer ;
     φ             : array [1 . . ] of proc of char;
     R             : array [1 . . , 1 . . ] of proc of bool;
begin
     for all 1 ≤ i ≤ #(w) do in parallel
          φ[i] := the i^th letter of w
     end ;
     for all 0 ≤ i, j ≤ #(w) do in parallel
          x:= the i^th letter of w;
          y:= the j^th letter of w;
          R[i, j] := (i > j) ∧ ((x, y) ∈ θ̄)
     end ;
     Trace_Dependency_Graph := ([1 . . #(w)] , R , φ)
end Trace_Dependency_Graph.
```

Figure 5. Constructing a dependency graph from a trace.

and we consider the concurrent alphabet $(\hat{\Sigma}, \hat{\theta})$, introduced in [Wr88]. The set of dependency graphs is written $\Gamma(\hat{\Sigma}, \hat{\theta})$. Now we transform $\Gamma(\hat{\Sigma}, \hat{\theta})$ in a reduction system. $(\Gamma(\hat{\Sigma}, \hat{\theta}), \Rightarrow)$. See [Ja88] for basic definitions on reduction systems.

Definition 10: Given $\gamma, \delta \in \Gamma(\hat{\Sigma}, \hat{\theta})$ such that $\gamma = (V, R, \varphi)$, we write $\gamma \Rightarrow \delta$ when there are $v, v' \in V$ and $z \in \hat{\Sigma}$ such that:

- $\varphi(v) = z$, $\varphi(v') = \overline{z}$.
- The longest directed path connecting v and v' has length 1.
- $\gamma \setminus \{v, v'\} \simeq \delta$.

The figure 6 give us an example of reduction.

Let us consider some basic properties of $(\Gamma(\hat{\Sigma}, \hat{\theta}), \Rightarrow)$.

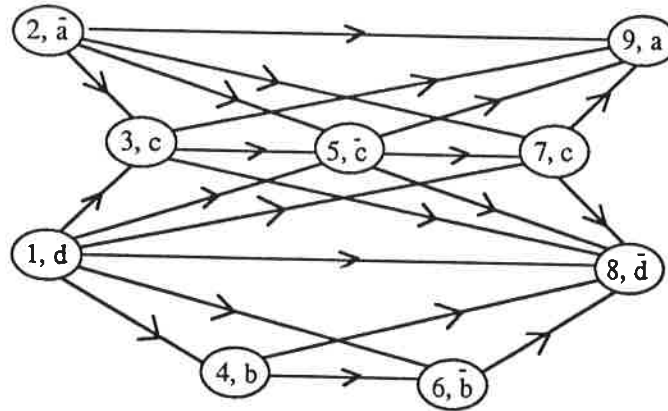
Lemma 11: When $\mu \Leftarrow \gamma \Rightarrow \delta$ then or $\mu \simeq \delta$ or exists γ' such that $\mu \Rightarrow \gamma' \Leftarrow \delta$.

Given γ we write $IRR(\gamma)$ the set of irreducible. By the above lemma we can prove that $IRR(\gamma)$ has only one element. We write $IRR(\Gamma)$ for the set of all irreducible elements in $(\Gamma(\hat{\Sigma}, \hat{\theta}), \Rightarrow)$.

$w = d\,\bar{a}\,c\,b\,\bar{c}\,\bar{b}\,c\,d\,\bar{a}$

Conflicts	
$a, \bar{a}$	$a, \bar{a}, c, \bar{c}$
$b, \bar{b}$	$b, \bar{b}, d, \bar{d}$
$c, \bar{c}$	$a, \bar{a}, c, \bar{c}, d, \bar{d}$
$d, \bar{d}$	$b, \bar{b}, c, \bar{c}, d, \bar{d}$

Dependency graph d (w)



$d(w) \Rightarrow d(w) \setminus \{3, 5\}$

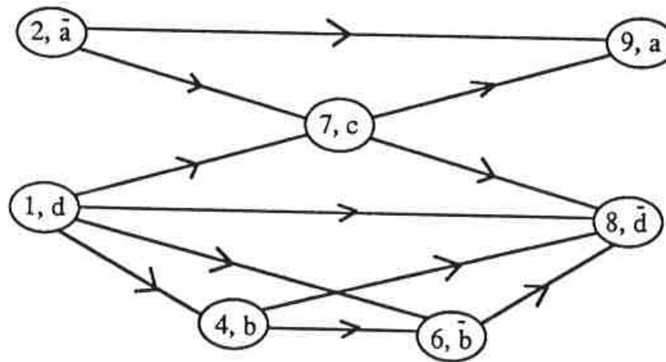


Figure 6. An example of reduction on dependency graphs.

Problem 12: Given a concurrent alphabet an a dependency graph test when it is irreducible. We denote this problem as *TEST_IRR*, it is:

Input: A concurrent alphabet $(\hat{\Sigma}, \hat{\theta})$ and $\gamma = (V, R, \varphi)$.

Output: It is true that $\gamma \in IRR(\Gamma)$?.

Lemma 13: The problem $TEST_IRR$ is NL^* complete.

Definition 14: Given $\gamma = (V, R, \varphi)$ we write $\overline{\gamma} = (V, \overline{R}, \overline{\varphi})$, where $\overline{R} = R^{-1}$ and $\overline{\varphi}(z) = \overline{\varphi(z)}$. We call $\overline{\gamma}$ the invers of γ.

Lemma 15: Given $\gamma, \delta \in IRR(\Gamma)$, the following conditions are equivalent:

- $\gamma \circ \delta \Rightarrow \rho$.
- There exist $z \in \hat{\Sigma}$ and $\gamma', \delta' \in IRR(\Gamma)$ such that $\gamma \simeq \gamma' \circ z$, $\delta \simeq \overline{z} \circ \delta'$ and $\rho \simeq \gamma' \circ \delta'$.

Lemma 16: Given $\gamma, \delta \in IRR(\Gamma)$, the following conditions are equivalent:

- $\gamma \circ \delta \overset{k}{\Rightarrow} \rho$ and $k \geq 1$.
- There exist $\mu, \gamma', \delta' \in IRR(\Gamma)$ such that $\#(\mu) = k$ and $\gamma \simeq \gamma' \circ \mu$, $\delta \simeq \overline{\mu} \circ \delta'$ and $\rho \simeq \gamma' \circ \delta'$.

Lemma 17: The inverses have the following properties:

- $\overline{\gamma \circ \delta} = \overline{\delta} \circ \overline{\gamma}$.
- $IRR(\gamma \circ \overline{\gamma}) = \lambda$.
- $IRR(\overline{\gamma \circ \delta}) = IRR(IRR(\overline{\delta}) \circ IRR(\overline{\gamma}))$.

Given γ and δ we define $\gamma \star \delta = IRR(\gamma \circ \delta)$. Then $(IRR(\Gamma_{\hat{\Sigma}}), \star)$ is a group isomorphic to the free partially commutative group $G(\theta)$ introduced in [Wr88].

Lemma 18: Given $\gamma, \delta \in IRR(\Gamma)$, there exists a unique $\mu \in IRR(\Gamma)$ such that $\gamma \simeq \gamma' \circ \mu$, $\delta \simeq \overline{\mu} \circ \delta'$ and $IRR(\gamma \circ \delta) \simeq \gamma' \circ \delta'$.

We call this unique μ the *maximum reducible factor* of γ and δ and we write $\mu = MAX_RED(\gamma, \delta)$.

Lemma 19: Given $\gamma, \delta \in IRR(\Gamma)$ such that $\gamma = (V_1, R_1, \varphi_1)$ and $\delta = (V_2, R_2, \varphi_2)$ we write

$$U_1 = \{v \in V_1 \mid \exists v' \in V_2 : \Delta^*(v, \gamma) \simeq \overline{< v', \delta >^*}\}$$

$$U_2 = \{v \in V_2 \mid \exists v' \in V_1 : \Delta^*(v', \gamma) \simeq \overline{< v, \delta >^*}\}$$

then $MAX_RED(\gamma, \delta) \simeq \gamma \setminus \overline{U_1} \simeq \delta \setminus \overline{U_2}$.

```
procedure Compose_Irr
      (γ    : labeled_graph ;
       δ    : labeled_graph ;
      )     : labeled_graph ;

from NL* import Equivalent_Dependency_Graphs (... , ...);
var pairing       : array [1 . . , 1 . . ,] of proc of bool;
    i, j          : integer;
begin
      for all 0 ≤ i, j ≤ max{#(γ), #(δ)} do in parallel
            pairing[i, j] := Equivalent_Dependency_Graphs ( Γ*(γ, i) , <δ, j>* );
      end ;
      Compose_Irr:= (γ \ {i | ∃ j : pairing[i, j]}) ∘ (δ \ {j | ∃ i : pairing[i, j]})
end Compose_Irr.
```

Figure 7. Composing irreducible elements.

Problem 20: Given a concurrent alphabet and two irreducible we define the problem *COMPOSE_IRR* as:

Input: A concurrent alphabet $(\hat{\Sigma}, \hat{\theta})$ and $\gamma, \delta \in IRR(\Gamma)$.

Output: The dependency graph $IRR(\gamma \circ \delta)$?.

Considering the program given in the figure 7, we have the result given in the following lemma.

Lemma 21: The problem *COMPOSE_IRR* belongs to NL^*.

Problem 22: Fixed a concurrent alphabet (Σ, θ) we define *FIND_IRR* as:

Input: Any dependency graph $\gamma \in \Gamma(\Sigma, \theta)$.

Output: The irreducible element $IRR(\gamma)$.

Considering the algorithm given in the figure 8 we obtain the following lemma

Lemma 23: The problem *FIND_IRR* belongs to *NC*.

Problem 24: We define the word problem for free partially commutative monoids, written *WORD_PROBLEM_FPCG* as:

Input: A concurrent alphabet (Σ, θ) and two graphs $\gamma = (V, R, \varphi)$ and $\delta = (V', R', \varphi')$.

Output: It is true that $\gamma, \delta \in \Gamma(\Sigma, \theta)$ and $\gamma \overset{*}{\Longleftrightarrow} \delta$?.

```
procedure Find_Irr
      (γ    : labeled_graph
      )     : labeled_graph ;

from NL* import
      Cartier_Foata (...) ;
      Compose_Irr (...) ;
var   w            : word;
      i, j         : integer;
      δ            : array [1 . . ] of proc of word;
begin
      w:= Cartier_Foata(γ) ;
      for all 1 ≤ i ≤ #(w) do in parallel
            δ[i] := dependency_graph corresponding to i^th letter of w
      end ;
      for j := 1 to log_2 #(w) do
            for all i : (i mod 2^j = 0) do in parallel
                  δ[i] := Compose_Irr (δ[i - 2^(j-1)] , δ[i] ))
            end ;
      Find_Irr := δ[#(w)]
end Find_Irr.
```

Figure 8. Finding $IRR(\delta)$.

```
procedure Word_Problem
      (θ    : relation;
       γ    : labeled_graph;
       δ    : labeled_graph;
      )     : bool;

from NC import
      Find_Irr(...);
      Correct_Dependency_Graphs (...);
      Equivalent_Dependency_Graphs (...);
begin
      Word_Problem:=   Correct_Dependency_Graph(θ, γ) ∧ Correct_Dependency_Graph(θ, δ)
                       ∧_c
                       Equivalent_Dependency_Graph (Find_Irr(γ),Find_Irr(δ))
end Word_Problem.
```

Figure 9. Word problem for free partially commutative monoids.

Lemma 25: The problem WP_FPCG belongs to NC.

Proof. Consider the algorithm given in the figure 9.□

It is also possible to represent traces by a set of sincronizet words called histories. This idea has been developed by A. Mazurkiewicz in [Ma88]. There exists a linear sequential time algorithm [Wr88] to solve word problem for free partially commutative groups. This algorithm uses histories to represent traces and works with a set of sincronizet puhsdowns.

References

[ABGS91] Álvarez, C; Balcázar, J.L.; Gabarró, J; Sántha, M.: Parallel complexity in the design and analysis of concurrent systems. PARLE 91, Lecture Notes in Computer Science **505** (1991) 288–303.

[AG91] Álvarez, C; Gabarró, J.: The parallel complexity of two problems on concurrency. *Information Processing Letters* **38** (1991) 61–70.

[AR88] Aalbersberg, IJ, J; Rozenberg, G.: Theory of traces. *Theoretical Computer Science* **60** (1988) 1–82.

[BDG88] Balcázar, J.L.; Díaz , J. Gabarró, J: *Structural Complexity I.* EATCS Monographs on Theoretical Computer Science, **11**, Springer–Verlag 1988.

[BDG90] Balcázar, J.L.; Díaz , J. Gabarró, J: *Structural Complexity II.* EATCS Monographs on Theoretical Computer Science, **22**, Springer–Verlag 1990.

[Co85] Cook , S.: A taxonomy of problems with fast parallel algorithms. *Information and Control* **64** (1985) 2–22.

[CF69] Cartier, P., Foata, D.: *Problèmes Combinatoires de Commutations et Réarran gements.* Lecture Notes in Mathematics, **85** (1969).

[GG91] Gabarró, J; Gavaldà, R.: An approach to correctness of data parallel algorithms. Report LSI–91–19, Univ. Politècnica de Catalunya.

[HS86] Hillis, D.W.; Steele, G.L.: Data parallel algorithms. *Communications of the ACM* **29** 1170–1183 (1986).

[La79] Lallement, G.: *Semigroups and Combinatorial Applications.* Wiley-Interscience 1979.

[Ja88] Jantzen, M.: *Confluent String Rewriting.* EATCS Monographs on Theoretical Computer Science, **14** Springer–Verlag 1988.

[Ma88] Mazurkiewicz, A.: *Basic Notions of Traces*, Lecture Notes in Computer Science **354** (1989) 285–363.

[Wr88] Wrathall , C.: The word problem for free partially commutative groups. *J. Symbolic Computation* **6** (1988) 99–104.