数理解析研究所講究録
第871巻 1994年 145-153

LR構文解析の並列アルゴリズムについて

椎名 広光　　増山 繁
Hiromitsu Shiina　　Shigeru Masuyama
豊橋技術科学大学　知識情報工学系

1 はじめに

並列構文解析のアルゴリズムは既にいくつか知られており, 特に文献[1]には一般の文脈自由文法[1, 4], Blacket Language[1]及びInput-driven Language[1]に対する並列構文解析のアルゴリズムが紹介されている. しかし, 現在知られている一般の文脈自由文法に対する並列構文解析のアルゴリズムは, n^6 個のプロセッサと $O(\log^2 n)$ 時間を必要とするため, 消費するプロセッサ数が多い. 一方, Blacket LanguageやInput-driven Languageは, $n/\log n$ 個のプロセッサを用いて $O(\log n)$ 時間で並列構文解析が実行可能ではあるが, 文法の生成能力が小さいという問題がある. そこで文脈自由文法の部分クラスの中で文の生成能力の点でコンパイラなどの作成に十分広い文法のクラスであるLR文法[3, 4]に対する効率の良い並列構文解析のアルゴリズムを考案することは, 重要な課題である.

本稿では, LR文法の生成規則がChomsky標準形[3]で表現されているという制限を課すことにより, 並列計算機の理論的なモデルであるCREW P-RAM[1]上で, 効率の良い並列アルゴリズムを考案した. なお, 簡単のため, 図1の文法と入力文字列の例を用いて説明する.

文法　　入力文字列
1: E→AA　4: A→a　　b c c a a
2: A→BA　5: B→b
3: A→CA　6: C→c

但し,
E,A,B,C : 非終端記号
a,b,c : 終端記号

図1: 文法と入力文字列の例

2 LR構文解析の並列アルゴリズムの概要

[アルゴリズムの概要]

Procedure PARALLEL-LR (概要)

前処理 (表の作成): **LR** 状態遷移図 **(LRA)**, 終端記号入力による状態遷移表 **(STTT)**, 非終端記号入力による状態遷移表 **(STTN)** を作成する. (第3章で詳述)

Step 1: 非決定性状態遷移リスト (NDSL, なお, NDSLによって定義される有向グラフを G_1 とする) の作成 (第4章で詳述)

Step 2: **LR** 解析表 **(LRPT)** の作成 (第5章で詳述)

Step 3: 構文解析木の木構造の決定 (第6章で詳述)

(1)**LRPT** から有向グラフ G_2 を作成する.

(2)**pebble game** 法を用いて G_2 から構文解析木に含まれない要素を削除する.

(3)**Euler tour technique** を用いて G_2 から構文解析木に含まれない要素を削除する.

Step 4: 構文解析木の作成 (第7章で詳述) □

3 前処理で作成する表 (並列アルゴリズムの概要の前処理.)

本章では, 並列アルゴリズムを実行するための前処理として作成する表について述べる.

3.1 LR 状態遷移図 (LRA)

この **LR** 状態遷移図 **(LRA)** の状態集合 Q は, スタックに対してポップをする状態集合 Q_{pop} とプッシュをする状態集合 Q_{push} に分けられる. この **LRA** は逐次処理でも同様に作成される (作成法は文献 [3, 4] など参照). 図 1 の文法に対する **LRA** を図 2 に示す.

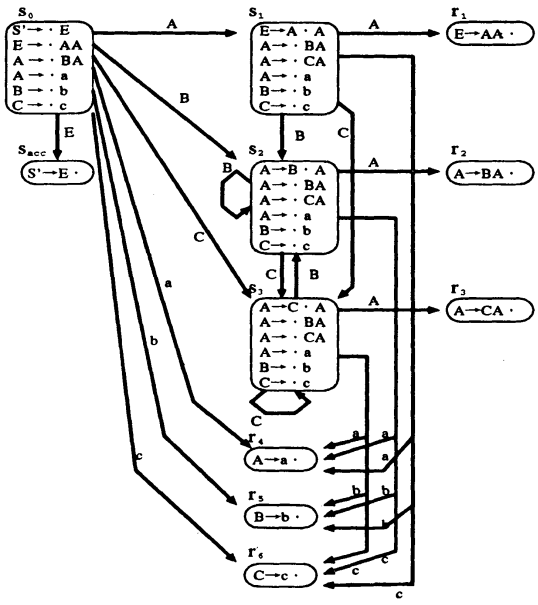


図 2: LR 状態遷移図 (**LRA**) の例

3.2 終端記号入力による状態遷移表 (STTT)

LRA の各状態 $s_i(\in Q_{push})$ から与えられた LR 文法中の全ての終端記号 a_k 1 文字を入力として次の入力を受け付ける状態 $s_j(\in Q_{push})$ に遷移するまで, それぞれ逐次処理の LR 構文解析と同じ動作 (文献 [3, 4] など参照) を実行することにより 得られる 3 つ組 (a_k, s_i, s_j) をまとめた表が終端記号入力による状態遷移表 **(STTT)** である.

(終端記号, 最初の状態, 最後の状態)	
(a, s_0, s_1)	(a, s_2, s_{acc})
(a, s_1, s_{acc})	(a, s_3, s_1)
(a, s_2, s_1)	(a, s_3, s_{acc})
(b, s_0, s_2)	(b, s_3, s_2)
(b, s_1, s_2)	
(b, s_2, s_2)	
(c, s_0, s_3)	(c, s_3, s_3)
(c, s_1, s_3)	
(c, s_2, s_3)	

図 3: 終端記号入力による状態遷移表 (**STTT**) の例

3.3 非終端記号入力による状態遷移表 (STTN)

LRA の状態 $s_i(\in Q_{push})$ から非終端記号 $A_i(\in N)1$ つで状態 $s_j(\in Q_{push})$ へ遷移できる場合, この状況の変化を **shift** 動作による遷移と呼ぶことにする.

また, LR 構文解析の動作が **LRA** の状態 s_i から開始し, 状態 $r_{i+1}, r_{i+2}\ldots(\in Q_{pop})$ へ遷移してから状態 s_j に到達する場合, この状況の変化を **reduce** 動作による遷移と呼ぶことにする.

非終端記号入力による状態遷移表 **(STTN)** は, **LRA** の各状態 s_i から各非終端記号 A_i によって遷移を開始させ, 次の終端記号の入力を受け付ける状態 s_j に遷移する場合, 状態間の遷移の種類を付加してまとめたものである.

但し, 非終端記号 A_i によって**shift**動作でも**reduce**動作でも遷移する場合 (s_i, s_j, **shift**), (s_i, s_j, **reduce**) を **STTN** に追加する.

(最初の状態, 最後の状態, 動作の種類)			
(s_0, s_1, shift)	(s_0, s_2, shift)	(s_0, s_3, shift)	(s_0, s_4, shift)
(s_1, s_2, shift)	(s_1, s_3, shift)	(s_1, s_{acc}, reduce)	
(s_2, s_2, shift)	(s_2, s_3, shift)	(s_2, s_1, reduce)	(s_2, s_{acc}, reduce)
(s_3, s_2, shift)	(s_3, s_3, shift)	(s_3, s_1, reduce)	(s_3, s_{acc}, reduce)

図 4: 非終端記号入力による状態遷移表 (**STTN**) の例

4 NDSL の作成 (並列アルゴリズムの概要の Step 1)

NDSL はある有向グラフ $G_1 = (V_1, E_1)$, 節点集合: V_1, 辺集合: E_1, を例えば隣接リスト [1] 等のデータ構造として表現したものとみなせるので, 以降, NDSL と G_1 を同一視する.

NDSL の節点集合 V_1 は, 各文字 a_i に対して対応付けられる **STTT** の最初と最後の状態を以下のように合成することで作成される. その合成処理の実行は, 入力文字 a_{i-1} の遷移の最後の状態 s^e_{i-1} と入力文字 a_i の遷移の最初の状態 s^b_i とが等しい場合, 要素 $v(i, s^b_i)$ を V_1 に格納する.

要素間をつなぐ辺集合 E_1 は, 隣合う入力の節点 $v(i-1, s_{i-1})$ と節点 $v(i, s_i)$ に対して, $v(a_i, s_{i-1}, s_i)$ が **STTT** 内にあるならば, $e(v(i-1, s_{i-1}), v(i, s_i))$ を E_1 に格納して作成される.

[**NDSL** の作成アルゴリズム]

Procedure MAKE-NDSL

Step 1: (NDSL の節点集合 V_1, 辺集合 E_1 を作成)

(1) $V_1 \leftarrow \phi$(空集合); $E_1 \leftarrow \phi$(空集合);

(2) **for all** $1 < i \leq n$, $s_i \in Q_{push}$ **in parallel do**

{NDSL の節点集合 V_1 を求める }

(3) **if STTT** 中の a_{i-1} に対応する最後の状態 s^e_i $= a_i$ に対応する最初の状態 s^b_i **then**

(4) $V_1 \leftarrow V_1 \cup \{v(i, s_i)\}$;

(5) $V_1 \leftarrow V_1 \cup \{v(1, s_0)\}$; $V_1 \leftarrow V_1 \cup \{v(n+1, s_{acc})\}$;

(6) NDSL の辺集合 E_1 を求める. □

以上の *Procedure MAKE-NDSL* によって NDSL が作成される. 図 1 の文法と入力文字列 *bccaa* に対する NDSL を図 5 に示す.

$S_0 \longrightarrow S_2 \longrightarrow S_3 \longrightarrow S_3 \longrightarrow S_1 \longrightarrow S_{acc}$

図 5: NDSL の例.

5 LR 解析表 (LRPT) の作成 (並列アルゴリズムの概要の Step 2.)

NDSL の要素の組合せから **LRPT** を作成する. **LRPT** は, 4 次元の配列 $LRPT[i,j,s_i,s_j]$ $(i<j, s_i, s_j \in Q_{push})$ からなり, 2 つの $LRPT[i,j,s_i,s_j].shift$ と $LRPT[i,j,s_i,s_j].reduce$ の要素に分けられている.

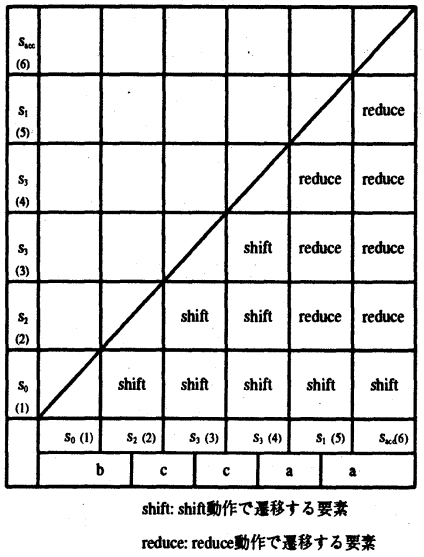
[LR 解析表 (LRPT) のアルゴリズム]

Procedure MAKE-LRPT

Step 1: (LRPT に状態遷移の可能性を登録)

(1) **for all** $1 \leq i < j \leq n+1$, $s_i, s_j \in Q_{push}$ **in parallel do**
begin
(2) **if** $v(i,s_i) \in V_1$ **and** $(s_i, s_j, shift) \in$ **STTN then**
(3) $LRPT[i,j,s_i,s_j].shift \leftarrow$ **true**;
(4) **if** $v(i,s_i) \in V_1$ **and** $(s_i, s_j, reduce) \in$ **STTN then**
(5) $LRPT[i,j,s_i,s_j].reduce \leftarrow$ **true**;
end □

図 5 に対する **LRPT** を図 6 に示す.



shift: shift動作で遷移する要素

reduce: reduce動作で遷移する要素

図 6: **LRPT** の例

6 構文解析木の木構造の決定 (並列アルゴリズムの概要の Step 3.)

このステップでは, 始めに **LRPT** の要素を節点 $v(\in V_2)$ とし, **LRPT** の要素間の遷移の可能性をその節点間の有向辺 $e(\in E_2)$ で表現した有向グラフ G_2 を作成する (詳細は, 第 **8.1** 参照). なお, G_2 は明らかに閉路を持たない. さらに, 有向グラフ G_2 は, 枝を 2 つしか持たない二分木 T_1 に同型な幾つかのグラフに分解可能である.

そして次に, このステップでは, **pebble game** 法を G_2 に適用して G_2 中の **pebble** 節点を削除する. 削除後の有向グラフを G_3 とする時, G_3 は, 次の条件 **TIC** を満たす.

[条件 TIC]
各節点に到達する有向辺が高々2つしかない有向グラフである.

6.1 有向グラフ G_2 の作成

LRPT の要素間に成り立つ構文解析の遷移の可能性に関する以下の2つの性質から, 有向辺 $e(\in E_2)$ を付加して, 有向グラフ $G_2 = (V_2, E_2)$ を作成する. 有向グラフ G_2 は, 構文解析木の枝と節点の候補者で, 後に構文解析木に属さない枝と節点を削除するのに使われる.

[性質 1] 5つの **LRPT** の要素を以下のように節点 v_1,v_2,v_3,v_4, 及び, v_5 で略記する.
$v_1 \leftarrow LRPT[i,k,s_i,s_k].shift$, $v_2 \leftarrow LRPT[i,k,s_i,s_k].reduce$, $v_3 \leftarrow LRPT[i,j,s_i,s_j].shift$,
$v_4 \leftarrow LRPT[j,k,s_j,s_k].reduce$, $v_5 \leftarrow LRPT[j,j+1,s_j,s_{j+1}].shift$.

これらの節点 v_1 から節点 v_5 は, **LRPT** 上で図7(a)のように配置され, 要素の位置関係に関して次の4つの条件が共通に成立しているものとする.

[条件 1] $i < j$, $j+1 < k$, [条件 2] $v_3 =$ **true**, [条件 3] $v_4 =$ **true**, [条件 4] $v_5 =$ **true**.

上記の4つの条件に加えて次の2つの場合のいずれかが成り立つ時, それぞれの要素間に遷移の可能性があるといえる.

[場合 1] $v_1 =$ **true**
[場合 1] は, 上の4つの条件が共に成り立つ場合で, 節点 v_3 から節点 v_1 への **shift** 動作による遷移の可能性と節点 v_4 から節点 v_1 への **reduce** 動作による遷移の可能性がある. よって, 辺 $e(v_4,v_1)$, $e(v_3,v_1)$ として表し, E_2 に格納する.

[場合 2] $v_2 =$ **true**.
[場合 2] は, 上の4つの条件が共に成り立つ場合で, 節点 v_3 から節点 v_2 への **shift** 動作による遷移の可能性と節点 v_4 から節点 v_2 への **reduce** 動作による遷移の可能性がある. また, 節点 v_3 から節点 v_2 と節点 v_4 から節点 v_2 への遷移の可能性をそれぞれ辺 $e(v_4,v_2)$,$e(v_3,v_2)$ として表し, E_2 に格納する.

[性質 2] これらの節点 v_1 から節点 v_4 は, **LRPT** 上で図7(b)のように配置され, 要素の位置関係に関して次の2つの条件が共通に成立しているものとする.

[条件 1] $v_3 =$ **true**, [条件 2] $v_4 =$ **true**.

上記の2つの条件に加えて次の2つの場合のいずれかが成り立つ時, それぞれの要素間に遷移の可能性があるといえる.

[場合 1] $v_1 =$ **true**.
[場合 1] は, 上の3つの条件が共に成り立つ場合で, 節点 v_3 から節点 v_1 への **shift** 動作による遷移の可能性と節点 v_4 から節点 v_1 への **reduce** 動作による遷移の可能性がある. よって, 辺 $e(v_4,v_1)$, $e(v_3,v_1)$ として表し, E_2 に格納する.

[場合 2] $v_2 =$ **true**.
[場合 2] は, 上の3つの条件が共に成り立つ場合で, 節点 v_4 から節点 v_2 への **shift** 動作による遷移の可能性と, また節点 v_4 から節点 v_2 への **reduce** 動作による遷移の可能性がある. よって, 辺 $e(v_4,v_2)$,$e(v_3,v_2)$ として表し, E_2 に格納する.

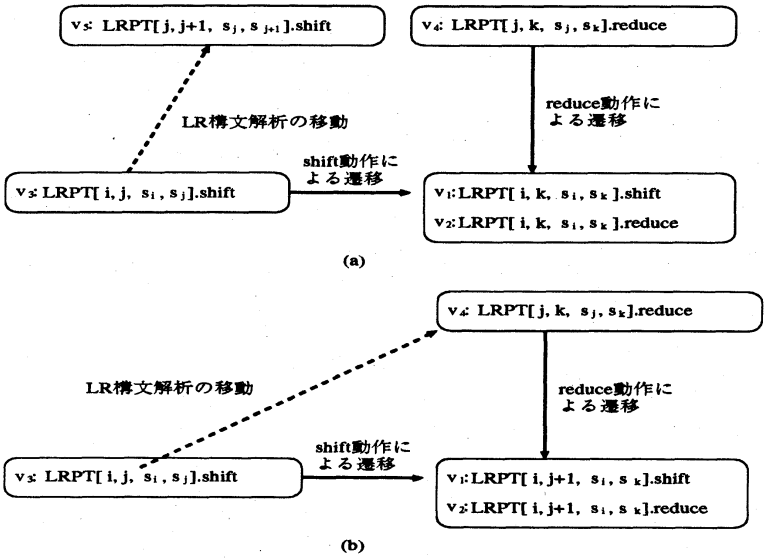


図 7: **LRPT** 上での配置

以上の 2 つの性質にあたる **LRPT** 内の遷移の可能性を有向辺 $e(\in E_2)$ として表現し, また **LRPT** の要素を節点 v $(\in V_2)$ として有向グラフ $G_2 = (V_2, E_2)$ を作成する. 節点集合 V_2 は **LRPT** の要素そのものであるが, 辺集合 E_2 は節点集合の組合せの一部分からなる表である. つまり, $E_2 \subset$ (**LRPT** の要素の集合) $\times$ (**LRPT** の要素の集合) である.

[有向グラフ G_2 作成のアルゴリズム]

Procedure MAKE-G_2

Step 1: (有向グラフ G_2 の節点集合 V_2 を作成)

$V_2 \leftarrow V_1$;

Step 2: (有向グラフ G_2 の辺集合 E_2 の作成)

(1) **if** 性質 1 が成立 **then**

[性質 1] によって, 有向グラフ G_2 の有向辺を追加する;

(2) **if** 性質 2 が成立 **then**

[性質 2] によって, 有向グラフ G_2 の有向辺を追加する. □

6.2 有向グラフ G_2 の節点の削除

有向グラフ G_2 から **pebble game** 法によって構文解析木に含まれない不要な節点を削除する. **pebble game** 法では, 有向グラフ G_2 を作成した時に同時に付した有向辺 ($e(v_2,v_1)$, $e(v_3,v_1) \in E_2$) の有向辺の元の節点 v_2,v_3 $(\in V_2)$ まで正しく構文解析できているならば, 節点 v_3 $(\in V_2)$ においても正しく構文解析ができることを利用して, 節点を削除する.

[定義]

(1) $COND(v_2) \leftarrow v_1$ は, v_2 から v_1 へのリスト COND を付ける.

(2) $v_i = LRPT[i,j,s_i,s_j].shift$ または $LRPT[i,j,s_i,s_j].reduce$, $v_{i+1} = LRPT[i,k,s_i,s_k].shift$, $v_{i+2} = LRPT[k,j,s_k,s_j].reduce$ とした時, $v_i \vdash v_{i+1}v_{i+2}$ は, 有向辺 $e(v_2,v_1)$ と有向辺 $e(v_3,v_1)$ が存在し, かつ, 2 つの非終端記号で $s_i \rightarrow s_k \rightarrow s_j$ へ遷移することができる場合, **true** となる.

[**LRPT** の節点の削除アルゴリズム]

Procedure DELETE-STATE

Step 1: (削除 **1: pebble game** 法による削除)
有向グラフ $G_2 = (V_2, E_2)$ に **pebble game** 法を適用して条件 **TIC** を満たす有向グラフ G_3 を作成する.

(1) 葉に当たる $LRPT[i, i+1, s_i, s_j].shift$ を **pebble** する.
(2)**repeat** $\log n$ **times**
begin
(3) $loop \leftarrow loop + 1$;
(4) **for all** $1 \leq i, j \leq k \leq n+1,\ s_i, s_j, s_k \in Q_{push}$ **in parallel do**
begin
(5) $v_1 \leftarrow$``$LRPT[i, k, s_i, s_k].shift$''; $v_2 \leftarrow$``$LRPT[i, k, s_i, s_k].reduce$'';
$v_3 \leftarrow$``$LRPT[i, j, s_i, s_j].shift$''; $v_4 \leftarrow$``$LRPT[j, k, s_j, s_k].reduce$'';
$v_5 \leftarrow$``$LRPT[j, j+1, s_j, s_{j+1}].shift$'';
(6) **if** v_1, v_2, v_3, v_4, v_5 が図 7(a) の配置を満たしている **then**
begin
(7) **if** $v_1 \vdash v_3 v_4$ **and** v_3 が $loop - 1$ 回目に **pebble** された要素 **then**
$COND(v_1) \leftarrow v_4$;
(8) **if** $v_2 \vdash v_3 v_4$ **and** v_3 が $loop - 1$ 回目に **pebble** された要素 **then**
$COND(v_2) \leftarrow v_4$;
end
(9) **if** v_1, v_2, v_3, v_4 が図 7(b) の配置を満たしている **then**
begin
(10) **if** $v_1 \vdash v_3 v_4$ **then**
$COND(v_1) \leftarrow v_3$;
(11) **if** $v_2 \vdash v_3 v_4$ **then**
$COND(v_2) \leftarrow v_3$;
end
end
{手続き **activate**}
(12) **for all** $v \in V_2$ **in parallel do**
$COND(v) \leftarrow COND(COND(v))$;
{手続き **square**}
(13) **for all** $v \in V_2$ **in parallel do**
$COND(v) \leftarrow COND(COND(v))$;
{手続き **square**}
(14) **for all** $v \in V_2$ **such that**
$COND(v)$ が **pebble** されている **in parallel do**
begin
v を **pebble** する;
v が **pebble** された時の loop 回数を記録する;
{手続き **pebble**}
end

end {**pebble game 法**を有向グラフ G_2 に適用する. }

(15) $V_3 \leftarrow \phi$(空集合); $E_3 \leftarrow \phi$(空集合);
for all $v \in V_2$ **in parallel do**
if $pebbled(v) =$ **pebble then**
$V_3 \leftarrow V_3 \cup \{v\}$;
for all $e(v_1, v_2) \in E_3$ **in parallel do**
$E_3 \leftarrow E_3 \cup \{e(v_1, v_2)\}$;
{**pebble** されていない節点を有向グラフ G_2 から削除し, **条件 TIC** を満たす有向グラフ G_3=(V_3, E_3) を作成する. }

Step 2: Euler tour technique によって削減する. □

$LRPT[1, n+1, s_0, s_{acc}].shift$ を構文解析木の根として, 根から順に **Euler tour technique** を用いて深さ優先に探索し, 各節点に順序を付ける. 番号の付けられた節点と節点間をつなぐ辺を有向グラフ G_4 とする. LRPT 上における有向グラフ G_4 を図 8 に示す.

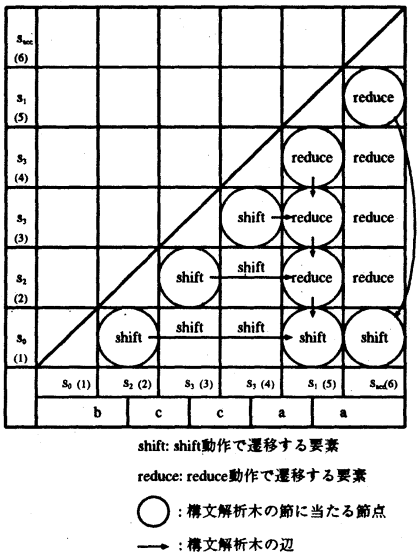


図 8: 有向グラフ G_4

7 構文解析木の作成 (並列アルゴリズムの概要の Step 4.)

構文解析木を求めるのに Blacket Languages に対する構文解析アルゴリズムを用いるので, まず入力文字列を括弧付きの入力文字列に変換する. 変換は残っている **LRPT** の要素の shift 要素を左括弧と reduce 要素を右括弧とみなして左右括弧を入力文字列に追加する. 追加後, Blacket Languages に対する構文解析アルゴリズムを括弧付きの入力文字列に適用する.

[構文解析木の作成アルゴリズム]

Procedure MAKE-PARSE-TREE

Step 1: 括弧の数を求める.
Step 2: 括弧付の入力文字列を作成する.
Step 3: 括弧付きの入力文字列に Blacket Languages に対する構文解析アルゴリズムを適用して, 構文解析木を求める. □

8 アルゴリズムの評価

前処理に当たる表の作成は, 入力文字列が与えられる前に文法が与えられた段階で予め1回だけ行なわれる. よって, 構文解析アルゴリズムの評価に含めない.

各 **Procedure** で必要な時間とプロセッサ数は以下の通りである.

[並列アルゴリズムの概要の **Step 1** の **Procedure MAKE-NDSL**]

Procedure MAKE-NDSL は各入力文字に対して最初の状態と最後の状態を求めるので, 定数時間と $O(n)$ 個のプロセッサ数が必要となる.

[並列アルゴリズムの概要の **Step 2** の **Procedure MAKE-LRPT**]

Procedure MAKE-LRPT は, NDSL の状態を組合わせて **LRPT** を作成する手続きで, 定数時間と $O(n^2)$ 個のプロセッサ数が必要となる.

[並列アルゴリズムの概要の **Step 3** の **Procedure MAKE-G_2**]

Procedure MAKE-G_2 は, 有向グラフ G_2 を作成する手続きで, **LRPT** の2番目の引数が同じ要素の中から2つの要素を取り出して辺を G_2 に付加しているので, CRCW P-RAM[1] 上で定数時間と $O(n^3)$ 個のプロセッサ数が必要となる. そこで CRCW P-RAM を CREW P-PRAM でシミュレートする [1] と $O(\log n)$ 時間と $O(n^3)$ 個のプロセッサ数が必要となる.

[並列アルゴリズムの概要の **Step 3** の **Procedure DELETE-STATE**]

Procedure DELETE-STATE の中では, **pebble game 法**を実行している.

pebble game 法の手続き **activate** では, $O(n^3)$ 個のプロセッサと $\log n$ 時間が必要である. また, **pebble game 法**の手続き **square** でリスト COND の数が **LRPT** の要素の数と同じであるため, $O(n^2)$ 個のプロセッサと $\log n$ 時間が必要である. よって, その他の処理を総合すると, **Procedure DELETE-STATE** の処理には, $O(\log n)$ 時間と $O(n^3)$ 個のプロセッサ数が必要となる.

[並列アルゴリズムの概要の **Step 3** の **Procedure MAKE-PARSE-TREE**]

Blacket Languages に対する構文解析アルゴリズムを用いるので, $\log n$ 時間と $O(n/\log n)$ 個のプロセッサ数が必要となる.

以上の **Procedure** の時間とプロセッサ数を総合すると, 本稿で提案した LR 構文解析の並列アルゴリズムは $O(n^3)$ 個のプロセッサを用いて $O(\log n)$ 時間で実行できる.

9 まとめ

以上, 本稿では, CREW P-RAM 上における LR 構文解析の並列アルゴリズムを示した. 本稿で示した並列アルゴリズムの評価より, 本アルゴリズムは, 理論的に効率的な並列アルゴリズムである.

また, 逐次処理の LR 構文解析のアルゴリズムは, 自然言語の構文解析にも応用されており, 本稿で示した並列アルゴリズムも同様に応用できると考えられる.

参考文献

[1] A.Gibbons, W.Rytter: Efficient Parallel Algorithms, Cambridge University Press (1989).

[2] J. van Leeuwen: Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity, The MIT Press (1990).

[3] A.V.Aho, R.Sethi, J.D.Ullman, Compilers (Principles, Techniques, and Tools), Addison-Wesley (1986), 原田 賢一 訳, コンパイラ 1,2(原理・技法・ツール), サイエンス社 (1990).

[4] J. ホップクロフト J. ウルマン: オートマトン 言語理論 計算論 1,2, 野崎, 高橋, 町田, 山崎 訳, サイエンス社 (1984).