論　文

木パターンマッチングのための並列アルゴリズム

非会員　太郎良浩次[†]　　正　員　平田　富夫[†]　　正　員　稲垣　康善[†]

A Parallel Algorithm for Tree Pattern Matching

Koji TARORA[†], *Nonmember*, Tomio HIRATA[†] *and* Yasuyoshi INAGAKI[†], *Members*

あらまし　テキスト木 t とパターン木 p が与えられたとき，t の部分木で p と照合するものを見つける処理を木パターンマッチングと言う．本論文では木パターンマッチングを行う並列アルゴリズムを提案する．t と p のサイズをそれぞれ n，m とするとき，提案するアルゴリズムは CREW-PRAM 上で $(mn/\log n)$ 台のプロセッサを使用し $O(\log n)$ 時間で木パターンマッチングを行う．これまでに知られている CREW-PRAM 上でのアルゴリズムは，$(mn/\log^2 n)$ 台のプロセッサを使用し，計算時間が $O(\log^2 n)$ であったので，本論文の結果は計算時間を改善する．また，プロセッサの割当て方を具体的に与えている．

キーワード　並列アルゴリズム，木パターンマッチング，項パターンマッチング，マッチングアルゴリズム

1. まえがき

テキスト木 t とパターン木 p が与えられたとき，t の部分木で p と照合（マッチング）するものを見つける処理を木パターンマッチングと言う．t と p はともにラベル付き順序木で，t の部分木と p が照合するというのは，変数ラベルの付いた p の葉を適当なラベル付き順序木で置き換えたとき，t の部分木と p がラベルを含めて一致することである．

木パターンマッチングは，非手続き的プログラミング言語のインタプリタ，コンパイラのコード発生および最適化，抽象データ型の直接実現システム，記号計算，更に，定理の自動証明などに現れる処理である[(1),(12),(15)]．これらのシステムの計算速度は木パターンマッチングの効率に大きく左右されるため，木パターンマッチングを効率良く実行するアルゴリズムは重要である．

これまでのところ，木パターンマッチングの逐次アルゴリズムに関しては，ストリングマッチングのように線形時間で実行される効率の良いアルゴリズムは知られていない．最初に木パターンマッチングのアルゴリズムを与えたのは文献(16)であるが，非常に限られた木のみを扱っていた．木パターンマッチングの問題を一般的に扱った最初の研究は Hoffman and O'Donnell[(14)] らによるもので，そこで提案されている上昇型と下降型の二つのアルゴリズムの時間計算量は，それぞれ $O(2^m+n)$ と $O(mn)$ である．但し，n，m はそれぞれテキスト木とパターン木のサイズ（頂点数）である．その後，いくつかのアルゴリズムが提案されたが，理論的な意味で素朴なアルゴリズムの効率 $O(mn)$ を改善するものではなかった[(4),(5),(13),(18)]．最近になって，Kosaraju[(19)] は $O(nm^{0.75}\mathrm{polylog}(m))$ 時間のアルゴリズムを与え，素朴なアルゴリズムより真の効率の良いアルゴリズムの存在を示した．その後，Galil[(9)] らはこれを $O(nm^{0.5}\mathrm{polylog}(m))$ に改良したことを報告している．

一方，並列アルゴリズムに関しては Ramesh[(22),(23)] らが次の二つのアルゴリズムを与えている．一つは，EREW-PRAM 上のアルゴリズムで，その性能は，プロセッサ数が $n^{2-\varepsilon}$ で，計算時間は $O(n^{\varepsilon}\log n)$ というものである．但し，ε は $(0<\varepsilon\leqq 2)$ の範囲の任意の実数である．$\varepsilon=0$ の場合，このアルゴリズムはプロセッサ数が n^2 で計算時間が $O(\log^2 n)$ となる．彼らが与えたもう一つのアルゴリズムは，CRCW-PRAM と CREW-PRAM 上で動作する．CRCW-PRAM 上に実現されるときプロセッサ数が $nr/\log n$ で計算時間が $O(\log n)$ となり，CREW-PRAM 上に実現されるときには，プロセッサ数が $nr/\log^2 n$ で計算時間が $O(\log^2 n)$ となる．但し，r はパターン木の頂点で変数ラベルをもつものの数である．なお，文献(22)，(23)では，$r\geqq$

† 名古屋大学工学部，名古屋市
Faculty of Engineering, Nagoya University, Nagoya-shi, 464 Japan

$\log^2 n$ を暗黙に仮定している．

本論文では CREW-PRAM 上で $mn/\log n$ 台のプロセッサを用いた $O(\log n)$ 時間の並列アルゴリズムを提案する．文献(22)，(23)では，入力の木をオイラー展開し，問題を $r+1$ 個のパターンに対するストリングマッチングに帰着している．このストリングマッチングには，Galil[10] による最適並列アルゴリズムを用いているが，CREW-PRAM 上の最適ストリングマッチングアルゴリズムは，これまでのところ計算時間が $O(\log^2 n)$ のものしか知られていない[10],[27]．本論文のアルゴリズムも，文献(22)，(23)と同様に，$r+1$ 個のパターンに対するストリングマッチングを行うが，プロセッサの台数を $mn/\log n$ まで許すことにより（$r=O(m)$ に注意されたい），$O(\log n)$ の計算時間を得ている．木を項(term)に変換して項のパターンマッチングを行っているので，入力が項で与えられる場合にも有効である．また，$m<\log n$ の場合にも対応できるようになっており，文献(22)，(23)でのような暗黙の仮定がない．更にプロセッサの割当て方を具体的に与えている．

2. 準　　備

2.1 木パターンマッチング

Σ を関数記号の集合とし，V ($V\cap\Sigma=\phi$) を変数記号の集合とする．各関数記号 $f\in\Sigma$ には引数の個数 $arity(f)$ が定まっているものとする．引数の個数が 0 の関数記号を定数記号と呼ぶ．

[定義 2.1] 項(term)は以下の(i)，(ii)，(iii)で定義される記号列である．

(i) 定数記号および変数記号は項である．

(ii) $f\in\Sigma$ で $t_1,\cdots,t_q$ が項のとき，$f(t_1,\cdots,t_q)$ も項である．但し，$arity(f)=q$ である．

(iii) 上の(i)と(ii)で定義されるもののみが項である．

以下では，定数記号を表すのに $a,b,c,\cdots$ を用い，他の関数記号を表すのに，$f,g,h,\cdots$ を用いる．変数記号には X，Y，Z などを用いる．また，本論文では，同一の変数記号が 2 度以上現れない項，すなわち，線形(linear)な項のみを取り扱う．

任意の項は自然な方法でラベル付き順序木とみなすことができる．例えば，項 $f(a,g(b,b))$ という項は図 1 のラベル付き順序木とみなすことができる．以下では，項とラベル付き順序木を区別せずに扱い，項を木と呼ぶこともある．項 t が表す木の頂点の個数を t のサイズと言い，$|t|$ と記述する．$t=f(a,g(b,b))$ の場合，そのサイズは $|t|=5$ である．

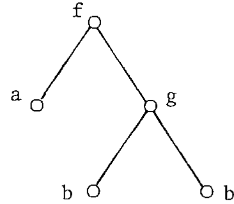

図 1　ラベル付き順序木
Fig. 1　A labeled ordered tree.

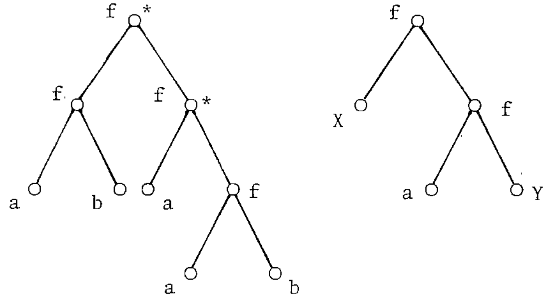

図 2　木パターンマッチング
Fig. 2　Tree pattern matching.

すべての線形な項の集合を T，変数記号が現れない項の集合を T^{-x} と記述する．$t\in T$ の部分列(substring) t' で，$t'\in T$ となっているものを t の部分項(部分木)と呼ぶ．以下では，パターン木は T の要素で，テキスト木は T^{-x} の要素とする．

[定義 2.2] パターン木 p における変数の出現が k 回であるとする．p がテキスト木 t に照合(マッチング)するというのは，k 個の木 $t_1,\cdots,t_k$ ($\in T^{-x}$) が存在し，それらを，p の変数に代入して得られる項が t に一致するときを言う．

テキスト木 t とパターン木 p が与えられたとき，t の部分木で p と照合するものが存在するか否かを判定し，存在するなら t の中での位置をすべて見つけることを木パターンマッチングと言う．

[例 2.1]

$t: f(f(a,b),f(a,f(a,b))$

$p: f(X,f(a,Y))$

とすると，図 2 に示すように，p は t に 2 箇所で照合する．すなわち，t の頂点で * 印の付いた頂点を根とする部分木に照合する．

本論文では入力の順序木は隣接リストで表されるとする．逐次アルゴリズムでは，順序木から（記号列とし

ての)項への変換とその逆の変換はともに線形時間でできるため，アルゴリズムの入力がどちらの形式でも問題はない．しかし，並列アルゴリズムでは入力形式の変換自体が新たな問題となる．以下，**3.**で入力の順序木を項に変換し配列に出力するアルゴリズムを与え，**4.**で入力の木が項の形で配列に与えられる場合の木パターンマッチングアルゴリズムを与える．**5.**はまとめである．

なお，文献(8)や(24)ではテキスト木自身とパターン木との照合を項マッチングと呼んで並列アルゴリズムを与えている．もちろん，これらのアルゴリズムをテキスト木のすべての部分木に並列に適用して木パターンマッチングを行うことは可能である．しかし，入力の与え方とプロセッサの割当て方が並列アルゴリズムでは微妙な問題として存在し，特に計算モデルがEREW-PRAMのときには慎重な扱いが必要である．更には，このような素朴な方法で性能の良い(プロセッサ数の少ない)アルゴリズムを得ることはあまり期待できない．実際，Dwork[8]らの項マッチングアルゴリズムを用いた場合，得られる木パターンマッチングアルゴリズムはプロセッサ数が$O(n^3)$となる．

2.2 計算モデル

並列アルゴリズムを実行する計算モデルとして，ここではPRAM (parallel random access machine)モデルを採用する．PRAMはプロセッサとして可算無限個のRAMを許し，共通の記憶装置(大域記憶)と各RAMに固有の記憶装置(局所記憶)をもつ．各プロセッサは同一のプログラムのもとで同期的に動作する．各プロセッサは自分のプロセッサ番号を知ることができるため，これを利用して他と異なる動作が可能となる．処理すべき仕事へのプロセッサの割当ては，このプロセッサ番号を通して行うことができる．大域記憶内の同じメモリ番地に各RAMからの同時アクセスを許すか許さないかで，concurrent-read(CR), exclusive-read(ER), concurrent-write(CW), exclusive-write(EW)と分類される．その制約の強いものからEREW-PRAM, CREW-PRAM, それにCRCW-PRAMなどがある．本論文で採用するPRAMモデルはCREW-PRAMである．PRAM上のアルゴリズムは，サイズnの入力が与えられたとき，nの多項式個のプロセッサを使って，$O((\log n)^{o(1)})$時間($\log n$の多項式)で計算が終了するとき，効率が良いと言われ，このような並列アルゴリズムをもつ問題のクラスはNCと呼ばれる[21],[25]．

PRAMは並列計算の理論的モデルとして定着しているが，多くのプロセッサが同時に大域記憶にアクセスできるためハードウェアとして実現性が高くないという見方もある．しかし，Module Parallel ComputerやButterfly Networkなどのように実現性の高いとされる多くの並列計算モデルは，PRAMの1ステップを$O((\log n)^{o(1)})$ステップ程度で模倣できることが知られており[11],[20]，従って，ここで考える効率の良いPRAMアルゴリズムはこれらの計算モデルに対してもやはり「効率の良い」アルゴリズムを与えるといってよい．

3. 順序木から項への変換

3.1 変換アルゴリズム

ここでは，入力として順序木が与えられたとき，それを(記号列としての)項に変換する並列アルゴリズムを与える．順序木は隣接リストで与えられるとする．以下で述べるアルゴリズムは，Tarjan and Vishkin[26]らによる木のオイラー閉路(木の各辺を互いに逆向きの2本の有向辺と見たときのオイラー閉路)を用いる．そのため，文献(26)と同様に入力の隣接リストはサーキュラーリストになっているとする．また，頂点iと頂点jを結ぶ辺は，頂点iの隣接リストには辺(i,j)として現れ，頂点jの隣接リストには辺(j,i)として現れるとする．入力の表し方の詳細については文献(26)を参照されたい．アルゴリズムを述べる前に次の定義が必要である．

[定義3.1] 順序木tの頂点vの最右深さ(rightmost depth) $\mathrm{rd}(v)$を次のように定義する．

(i) vがtの根のとき，または，vが自分の親$p(v)$の最右の子でないときは$\mathrm{rd}(v)=0$である．

(ii) vが親$p(v)$の最右の子のとき，$\mathrm{rd}(v)=\mathrm{rd}(p(v))+1$である．

図3に各頂点の最右深さを示す．変換アルゴリズムは次の3ステップからなる．入力の順序木のサイズはnとする．

① 入力の順序木の各頂点vのプリオーダ番号(preorder number) $\mathrm{pn}(v)$と最右深さ$\mathrm{rd}(v)$を計算する．

② 各頂点のラベルを，その頂点のプリオーダ番号の順に配列に格納する．

③ 括弧とカンマを以下のようにして挿入する．

(i) 定数記号以外の関数記号の右に左括弧を挿入する．

(ii) 定数記号および変数記号の右に，その記号に対応する頂点の最右深さの数だけ右括弧を挿入する．

(iii) 定数記号，変数記号または右括弧の右に関数

記号または変数記号があれば，それらの間にカンマを挿入する．

[例 3.1] 図 3 の順序木に上のアルゴリズムを適用したときの各ステップの実行結果は次のようになる．ステップ 1 の後に，プリオーダ番号と最右深さが図 3 のように各頂点に対し求まる．次に，ステップ 2 を実行すると，配列には次のように記号列が格納される．

$fafab$

更に，ステップ 3 の(i)を実行すると，

$f(af(ab$

となり(ii)の後では次のようになる．

$f(af(ab))$

最後に(iii)を実行して次の項が得られる．

$f(a,f(a,b))$

上のアルゴリズムで，ステップ 1 のプリオーダ番号は，文献(26)で与えられた方法で計算する．最右深さについても同様にして求めることができることを次節で示す．このステップは n 台のプロセッサで，$O(\log n)$ 時間で計算できるが，文献(7)の方法を適用すれば，計算時間は同じで，プロセッサ数を $n/\log n$ に減らすことが可能である．詳しくは次節で述べる．ステップ 2 は n 台のプロセッサで，$O(1)$ 時間で実行できるので，いわゆる Brent の定理を適用して，$n/\log n$ 台のプロセッサで $O(\log n)$ の時間で計算できる[3],[11]．Brent の定理とは，合計で q 個の基本ステップの計算を何台かのプロセッサで並列に t 時間で計算できるなら，p 台のプロセッサで，$O(q/p+t)$ 時間で並列計算ができるというものである．但し，Brent の定理では，q 個の基本ステップの分割の仕方やプロセッサの割当て方に言及していないので，その適用に際しては注意が必要である．ステップ 3 の(i)，(ii)，(iii)は，プリフィクス計算を用いて，それぞれ $n/\log n$ 台のプロセッサで，$O(\log n)$ 時間で計算可能である．プリフィクス計算とは，数列 $a_0, a_1, \cdots, a_{n-1}$ が配列に入っているとき，$a_0, a_0+a_1, a_0+a_1+a_2, \cdots, a_0+a_1+\cdots+a_{n-1}$ を並列に計算する手法である．2 分木状に計算を進めることにより，$n/\log n$ 台のプロセッサを用い $O(\log n)$ 時間で計算できる[11]．

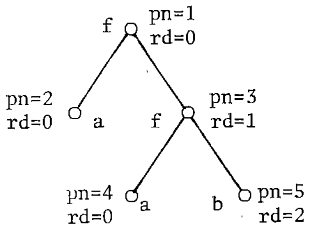

図 3　プリオーダ番号と最右深さ
Fig. 3　Preorder numbering and rightmost depth.

3.2 最右深さの計算

前節のアルゴリズムのステップ 1 の詳細を述べる．まず，Tarjan and Vishkin[26] らによるプリオーダ番号の計算方法を簡単に紹介し，次に，そこで開発された走査リストを用いて最右深さが計算できることを示す．

文献(26)では，まず，順序木のオイラー閉路を構成し，根に戻る辺の後ろで切って一方向リストを作る．これを走査リスト (traversal list) と呼ぶ．図 3 の順序木の走査リストを図 4 に示す (簡単のため，図 4 ではプリオーダ番号を頂点の名前としている)．

次に，走査リストに対し順位付け (list ranking) を行う．すなわち，各要素がリストの先頭から何番目に位置するかを計算する．頂点 i と j 間を結ぶ辺は，走査リストには，辺 (i,j) と辺 (j,i) として 2 回現れる．走査リストの先頭に近い方 (つまりランクの小さい方) を前進辺 (advance edge) と呼び，遠い方 (ランクの大きい方) を後退辺 (retreat edge) と呼ぶ．前進辺に 1 を割り当て，後退辺に 0 を割り当てて，プリフィックス計算を行うと，前進辺のみに着目したときの (走査リスト内での) 順位がわかる．前進辺 (i,j) の順位に 1 を加えたものが頂点 j のプリオーダ番号である．

文献(26)ではここまでの作業が n 台のプロセッサで $O(\log n)$ 時間でできることを示している．しかし，Cole and Vishkin[7] らの方法を適用すれば，計算時間はこのままで，プロセッサの台数を $n/\log n$ に減らすことが可能である．上のアルゴリズムの各ステップを詳しく見てみよう．走査リストの構成は，n 台のプロセッサで $O(1)$ 時間でできる．従って，Brent の定理により，$n/\log n$ 台のプロセッサを用い，$O(\log n)$ 時間で実行可能

	(1,2)	(2,1)	(1,3)	(3,4)	(4,3)	(3,5)	(5,3)	(3,1)
advance/retreat	a	r	a	a	r	a	r	r
0/1 assignment	0	0	0	0	0	0	1	1

図 4　走査リスト
Fig. 4　A traversal list.

である．プリフィクス計算をするには，走査リストの要素をリストでの並びと同じ順に配列に格納しておく．前節で述べたように，配列のデータに対するプリフィクス計算は $n/\log n$ 台のプロセッサで，$O(\log n)$ 時間で実行できる．配列への格納は，走査リストの順位付けができていれば，走査リストの構成と同じプロセッサ数，同じ計算時間でできる．問題となるのは，走査リストの順位付けである．リストの順序付けは，一般にダブリングと呼ばれる手法で行われる[11]．これは n 台のプロセッサがあれば $O(\log n)$ 時間で実行できるが，プロセッサの台数を $n/\log n$ に減らすのは，一見すると不可能のように思える．しかし，文献(7)で Cole and Vishkin らは，非常に巧妙なアルゴリズムでこれができることを示した．このアルゴリズムは時間計算量の定数係数が非常に大きいため，実用上の観点からは問題があるが，その後，文献(2)はこれを解決している．以上のことはすべて EREW-PRAM で実現可能である．但し，文献(7)と(2)では PRAM が unary 表現の数を定数時間で binary 表現に変換できる命令をもっていると仮定している[6]．

さて，走査リストを用いると最右深さもプリフィクス番号を求めるのと同じプロセッサ数，同じ計算時間で得られることを以下に示す．

走査リストの各要素は，その順位に従って配列に格納されているとする．後退辺で，その後続の要素がやはり後退辺であるものに 1 を割り当てる．走査リストの最後の要素（これも後退辺である）にも 1 を割り当てる．その他の辺に 0 を割り当てる（図 4）．このとき，辺 $e=(i,j)$ が後退辺だとすると，e を含め e より右側（リストの後方）に連続する 1 の個数が頂点 i の最右深さである．図 5 のアルゴリズムは各後退辺について，その右側にはじめて現れる 0 の位置を求め，最右深さを計算する．0 と 1 の割当てが入った配列を $A(0,\cdots,n)$ とする（この n は配列のサイズで，木のサイズの約 2 倍の大きさである．また，$A(n)=0$ と初期設定される）．このアルゴリズムはまず，A をサイズ $\log n$ のブロックに分割し，各ブロックに 1 台のプロセッサを割り当てる．次に，各プロセッサは担当のブロックをスキャンし，各位置 i で，そこより右の 0 で自ブロック内の最も近いものを指すようにポインタを設定する．そのような 0 がないときには，右隣のブロックの先頭を指すように設定する．このように設定したポインタに対しダブリングを行う．最後に，もう一度各ブロック内をスキャンし，各位置 i について rd(i) を求める．このアルゴリズムの計算時間は $O(\log n)$ 時間で，プロセッサ数は $n/\log n$ である．なお，本論文のアルゴリズムでは簡単のため $\log n$，$n/\log n$ などはすべて整数であると仮定している．整数でない場合でも，アルゴリズムを局部的に変更することで容易に対応でき，そのときの計算時間は定数倍にしかならない．

```
procedure rightmost_depth
for all j, 0 ≤ j ≤ n/log n − 1 in parallel do
begin { j はプロセッサ番号 }
    if j = 0 then B(n/log n) ← n/log n;
    zeroposition ← (j + 1) log n;
    for i ← (j + 1) log n − 1 downto j log n do
        begin
          if A(i) = 0 then zeroposition ← i;
          B(i) ← zeroposition;
        end
    for i ← 0 to log n − 1 do {doubling}
        B(j log n) ← B(B(j log n));
    for i ← j log n to (j + 1) log n − 1 do
        if B(i) = (j + 1) log n then
                rd(i) ← B(B(i)) − i
        else rd(i) ← B(i) − i;
end
```

図 5 最右深さの計算
Fig. 5 Computing rightmost depths.

（注意） 図 5 は CREW-PRAM のアルゴリズムである．EREW-PRAM の場合はダブリングのときに同時読込みをしないように変更する必要がある．これは，リストの終端を指すようになったポインタに関してはダブリングを止めるようにすればよい．

以上のことより，次の補題を得る．

[補題 1] 順序木から項への変換は，順序木のサイズを n とすると，$n/\log n$ 台の EREW-PRAM プロセッサを用いて $O(\log n)$ 時間で並列計算できる．但し，PRAM は unary 表現の数を定数時間で binary 表現に変換できる命令をもっていると仮定する．

4. 項の並列パターンマッチング

4.1 アルゴリズムの概略

ここでは，入力が（記号列としての）項で与えられる場合の木パターンマッチングアルゴリズムを述べる．項 t のサイズは，t の記号列としての長さと定義する（サイズ n の順序木を項に変換すると，そのサイズは n より大きくなるが，たかだか定数倍である）．

具体的なアルゴリズムを与える前に，その概略を述べる．入力のテキスト項 t とパターン項 p のサイズをそれぞれ，$n, m\ (n>m)$ とする．パターン p の中に現

れる変数を p の先頭から順に $X_1, X_2, \cdots, X_r$ とする．X_k と X_{k+1} の間に挟まれる p の部分列を p_k と記述する．但し，p_0 は p のプリフィクス，p_r は p のサフィクスである．

テキスト項 t の部分項で，t の先頭から $i\,(0 \leqq i < n)$ 番目の記号から始まるものを t_i と記述する（i 番目の記号が括弧やカンマの場合は t_i は定義されない）．

項パターンマッチングアルゴリズムは，各 $i\,(0 \leqq i < n)$ について並列に，p と t_i が照合するか否かを調べる．p と t_i が照合するのは，各 $p_k\,(0 \leqq k \leqq r)$ が t_i の $r+1$ 個の部分列 $t_{i_0}, t_{i_1}, \cdots, t_{i_r}$ と一致し，しかも t_{i_j} と $t_{i_{j+1}}$ で

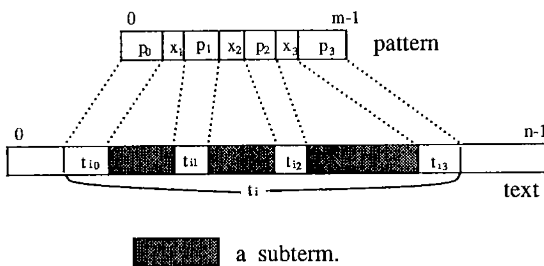

図6　項パターンマッチング
Fig. 6　Term pattern matching.

挟まれた部分が t_i の部分項になっているときである（図6）．このとき，各 $k\,(0 \leqq k < r)$ について p_k と p_{k+1} は正しい位置で t の部分列 t_{i_k}，$t_{i_{k+1}}$ と照合していると言う．従って，p と t_i の照合は，各 $k\,(0 \leqq k \leqq r)$ について p_k と t_{i_k} の照合（ストリングマッチング）を行い，次に照合に成功した p_k と p_{k+1} が正しい位置で照合していることを調べるという処理を行えばよいことになる．提案する並列アルゴリズムは，テキスト項とパターン項に前処理を行うことにより，これらを効率良く実行する．

4.2　パターン項とテキスト項の前処理

入力のテキスト項 t とパターン項 p はそれぞれ大きさ n と m の1次元配列 $text(0, \cdots, n-1)$ と $pattern(0, \cdots, m-1)$ に与えられる．パターン項に対する前処理を図7のように行う．ここでは，各変数 X_i の p における位置を配列 $location(1, \cdots, r)$ に求め，次に，各 $k\,(1 \leqq k \leqq r)$ について変数 X_k から変数 X_{k+1} までの距離（$= |p_k|$）を計算している．最後にプリフィクス計算により，$sumlen(i) = |p_0| + |p_1| + \cdots + |p_i|$ を計算している．これはパターンマッチング時のプロセッサ割当てに用いられる．プリフィクス計算を除くとその他の処理は m 台のプロセッサで $O(1)$ 時間で実行できる．従って，この前処理は m 台のプロセッサで

procedure *preprocessing_pattern*
begin
1　**for** all j,　$0 \leq j \leq m-1$　**in parallel do**
　　　if ($pattern(j)$ が変数記号) **then**　$A(j) \leftarrow 1$
　　　else　$A(j) \leftarrow 0$;
2　配列 $A(0, \cdots, m-1)$ に対してプリフィクス計算を行ない、結果を配列 $C(0, \cdots, m-1)$ に格納する;
　　{ つまり $C(i) = |\{j | A(j) = 1, j \leq i\}|$ となる }
3　$r \leftarrow C(m-1)$;
4　**for** all j,　$0 \leq j \leq m-1$　**in parallel do**
　　begin
　　　if ($pattern(j)$ が変数記号) **then**
　　　　　$location(C(j)) \leftarrow j$;
　　　$location(0) \leftarrow -1$;
　　　$location(r+1) \leftarrow m$;
　　end;　{ $location(i)$ は変数 X_i の位置 }
5　**for** all k,　$0 \leq k \leq r$　**in parallel do**
　　$length(k) \leftarrow location(k+1) - location(k) - 1$;
　　{ つまり $length(k)$ は p_k の長さとなる }
6　配列 $length(0, \cdots, m-1)$ に対してプリフィクス計算を行ない、結果を配列 $sumlen(0, \cdots, m-1)$ に格納する;
　　{ つまり $sumlen(i) = |p_0| + \cdots + |p_i|$ }
7　$sumlen(-1) \leftarrow 0$;
end　*preprocessing_pattern*

図7　パターンの前処理
Fig. 7　Preprocessing of pattern.

procedure *preprocessing_text*
begin
1　**for** all j,　$0 \leq j \leq n/\log n - 1$　**in parallel do**
　　　for　$i \leftarrow j \log n$　**to**　$(j+1)\log n - 1$　**do**
　　　　if　($text(i)$ がかっこ)　**then**　$A(i) \leftarrow 1$
　　　　else　$A(i) \leftarrow 0$;
2　配列 $A(0, \cdots, n-1)$ に対してプリフィクス計算を行ない、結果を配列 $C(0, \cdots, n-1)$ に格納する;
3　**for** all j,　$0 \leq j \leq n/\log n - 1$　**in parallel do**
　　　for　$i \leftarrow j \log n$　**to**　$(j+1)\log n - 1$　**do**
　　　　if　($text(i)$ がかっこ)　**then**
　　　　　begin
　　　　　　$D(C(i)) \leftarrow text(i)$;
　　　　　　$C^{-1}(C(i)) \leftarrow i$;　{ C^{-1} は配列 }
　　　　　end;
4　配列 D の記号列に対し、[11, Section 3.1] の方法で $match(i)$ を計算する;
5　**for** all j,　$0 \leq j \leq n/\log n - 1$　**in parallel do**
　　　for　$i \leftarrow j \log n$　**to**　$(j+1)\log n - 1$　**do**
　　　　begin
　　　　　$correspond(i) \leftarrow i$;
　　　　　if　($text(i)$ が左かっこ)　**then**
　　　　　　$correspond(i-1) \leftarrow C^{-1}(match(C(i)))$;
　　　　end;
end　*preprocessing_text*

図8　テキストの前処理
Fig. 8　Preprocessing of text.

$O(\log m)$ 時間で実行できる．

テキスト項 t の i ($0 \leqq i \leqq n-1$) 番目の記号から始まる長さ l の部分列を $text(i, l)$ と記述する．テキスト項 t の前処理では，関数 $correspond(i)$ の値を計算する．$correspond(i)$ は，$text(i)$ が関数記号または定数記号のとき，部分項 t_i の終わる位置を表す．すなわち，$correspond(i) = i + |t_i| - 1$ である（従って，$correspond(i)$ は1対1対応の関数である．$text(i)$ が括弧やカンマのときは $correspond(i)$ の値は不定とする）．

左括弧と右括弧のみからなる長さ n の記号列に対し，次のような関数 $match(i)$ の計算が，$n/\log n$ 台のプロセッサを用いて $O(\log n)$ 時間で計算できることが知られている[11]．$match(i)$ は，i 番目の記号が左（右）括弧のとき，それに対応する右（左）括弧の位置を値とする．テキスト項 t の前処理を図8のように行う．これらのステップは $n/\log n$ 台のプロセッサで $O(\log n)$ 時間で実行できる．

4.3 項パターンマッチング

前処理によって得られた情報を用いて，項パターンマッチングを以下の三つのステップで実行する．

（ステップ1） p の部分列 p_k ($0 \leqq k \leqq r$) の各々と t の間で並列にストリングマッチングを行い関数 $match(i, k)$ を計算する．$match(i, k)$ は，p_k が $text(i, |p_k|)$ と照合（ストリングマッチング）するとき true の値をとり，そうでないときは false の値をとる関数である（テキストの前処理で用いた関数 $match(i)$ と名前が似ているが，異なるものである）．

ステップ1を実行するアルゴリズムを図9に示す．ここでは，各 p_k に対し，$|p_k|\, n/\log n$ 台のプロセッサを割り当てる．$|p_k| \geqq \log n$ のとき，t の各位置 i と p_k とのマッチングを $|p_k|/\log n$ 台のプロセッサで調べる．これは，p_k を長さ $\log n$ のブロックに分割し，それぞれのマッチングを1台ずつで実行する．プロセッサの割当てで問題となるのは，$|p_k| < \log n$ の場合である．このとき，$|p_k|\, n/\log n < n$ なので，t のすべての位置にプロセッサを割り当てることができない．そこで，次のような割当てを行う．t を長さ $\log n$ の区間（$n/\log n$ 個ある）に分割し，各区間に対し1台のプロセッサを割り当てる．このプロセッサは長さ $2\log n$ の部分列（担当の区間とその右隣の区間）と p_k とのストリングマッチングを逐次型のアルゴリズムで実行する．このストリングマッチングではテキスト，パターンとも長さが $O(\log n)$ なので，Knuth-Morris-Pratt のアルゴリズム[17]を用いて $O(\log n)$ 時間で実行できる．使用したプロセッサの総数は $\sum_{k=1}^{r} |p_k| \cdot n/\log n \leqq mn/\log n$ である．よって，関数 $match(i, k)$ は PRAM 上で $mn/\log n$ 台のプロセッサを用い $O(\log n)$ 時間で計算できる．

（ステップ2） $text(i, |p_k|)$ とのストリングマッチング

procedure *match*
for all j, $0 \leq j < mn/\log n$ **in parallel do**
begin
1　二分探索により $sumlen(k-1) \cdot n/\log n \leq j < sumlen(k) \cdot n/\log n$ なる k を見つける;
　　{ プロセッサ j は p_k を担当する }
2　$l \leftarrow j - sumlen(k-1) \cdot n/\log n$;
3　**if** $|p_k| \geq \log n$ **then**
　　begin
　　　$i \leftarrow \lfloor l/(|p_k|/\log n) \rfloor$;
　　　$s \leftarrow l - i \cdot |p_k|/\log n$;
　　　$text(i, |p_k|)$ と p_k の s 番目のブロックとの照合を逐次的に調べる．すべてのブロックで照合しているとき（これはダブリングを用いて調べる）、$match(i, k) \leftarrow$**true** とする;
　　end
　else if $l < n/\log n$ **then**
　　$text(l \cdot \log n, 2\log n)$ と p_k のストリングマッチングを Knuth-Morris-Pratt のアルゴリズムを用いて行ない、$match(i, k)$ $(l \cdot \log n \leq i < (l+1)\log n)$ の値を求める;
end

図9 マッチングアルゴリズム（ステップ1）
Fig. 9 Macthing algorithm (Step 1).

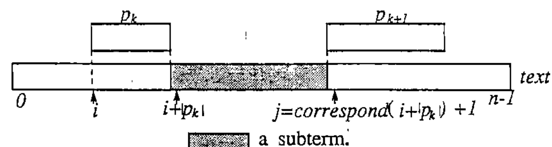

図10 関数 *correspond* (i)の使用法
Fig. 10 Use of *correspond* (i).

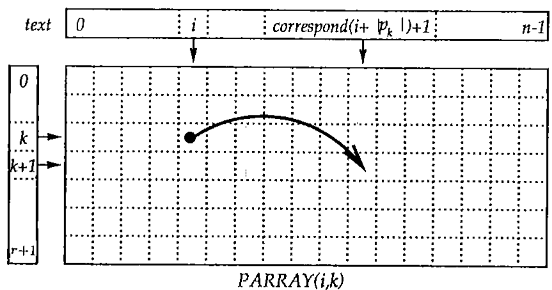

図11 ポインタアレー*PARRAY*
Fig. 11 Pointer array *PARRAY*.

に成功した記号列 p_k はその次に続くべき記号列 p_{k+1} の t 内での位置を関数 *correspond* によって知ることができる (図 10). 従って, $j=correspond(i+|p_k|)+1$ の位置で $match(j,k+1)$ の値が true であれば, p_k と p_{k+1} は互いに正しい位置で t の部分列と照合していることがわかる. 各 k $(0\leqq k<r)$ について p_k と p_{k+1} が正しい位置で照合していることを $O(\log n)$ 時間で調べるためにポインタの 2 次元配列 $PARRAY(i,k)$ $(0\leqq i<n)$ $(0\leqq k<m)$ を用意する. $match(i,k)=$true のとき配列の要素 $PARRAY(i,k)$ が要素 $PARRAY(j,k+1)$ を指すように設定する (図 11). このようにポインタを設定するとこれらのポインタにより連なっている $PARRAY$ の要素は線形リストを構成する. このとき, 長さ $r+1$ のリスト (要素の個数が $r+1$ のリスト) が生じていることと p と t の部分項が照合していることが等価である.

(ステップ 3) 長さ $r+1$ のリストの存在をダブリングを用いて調べる.

ステップ 2 とステップ 3 を実行するアルゴリズムを図 12 と図 13 に示す.

```
procedure setpointer
for all j, 0 ≤ j < mn/log n in parallel do
begin
   k ← ⌊j/(n/log n)⌋; { 担当する p_k を決める }
   l ← j − k·n/log n;
   for u ← 0 to log n − 1 do
     begin
       PARRAY(l·log n + u, k) ← (l·log n + u, k);
       if match(l·log n + u, k) =true and k < r
       then PARRAY(l·log n + u, k)
              ← (correspond(l·log n + u + |p_k|) + 1,
                 k + 1);
       if match(l·log n + u, r) =true and k = r
       then PARRAY(l·log n + u, k) ← (0, k + 1);
     end
end
```

図 12 マッチングアルゴリズム (ステップ 2)
Fig. 12 Matching algorithm (Step 2).

```
procedure doubling
for all j, 0 ≤ j < mn/log n in parallel do
begin
   if m ≥ log n then
      begin
         s ← ⌊j/n⌋;{j は s log n 行目を担当する }
         i ← j − sn;
         for u ← 0 to log n + ⌈log(r + 2)⌉ do
             PARRAY(i, s log n)
               ← PARRAY(PARRAY(i, s log n));
         if PARRAY(i, 0) = (0, r+1) and s = 0
             then match(i, r + 1) ←true;
      end
   else for u ← 0 to log n/m − 1 do
         begin
            i ← j(log n/m) + u;
            for k ← 0 to r do
                PARRAY(i, 0)
                   ← PARRAY(PARRAY(i, 0));
            if PARRAY(i, 0) = (0, k + 1)
                then match(i, r + 1) ←true;
         end;
end
```

図 13 マッチングアルゴリズム (ステップ 3)
Fig. 13 Matching algorithm (Step 3).

ステップ 3 のプロセッサ割当てで問題となるのは $m<\log n$ の場合である. ここでは, $m\geqq\log n$ のときとそうでないときで, リストのたどり方を切り換える. まず, $m\geqq\log n$ の場合には図 11 の $PARRAY$ において, $\log n$ 行ごとに n 台のプロセッサを割り当てる. 割当てを受けた行では, 各プロセッサが一つの列を担当する. 各プロセッサは自分の属す位置から $\log n-1$ 行下がるまでポインタをたどり, その後通常のダブリングを行う. $m<\log n$ の場合には, 次のようにプロセッサを割り当てる. $PARRAY$ の 0 行目を $\log n/m$ の長さで区切り, 各区間に 1 台のプロセッサを割り当てる. このプロセッサは担当の区間に始点があるリストをすべて逐次的にたどって長さ $r+1$ のリストがあるかを調べる. 一つの区間に始点を置くリストの長さの総計はたかだか $(r+1)\cdot\log n/m(<\log n)$ なので, このプロセッサの計算時間は $O(\log n)$ である.

前節で述べたように, パターン項に対する前処理は m 台のプロセッサを用いて $O(\log m)$ 時間で実行でき, テキスト項に対する前処理は $n/\log n$ 台のプロセッサを用いて $O(\log n)$ 時間で実行できる. $m<n$ としてよいので, これらの前処理は $mn/\log n$ 台のプロセッサを用いて $O(\log n)$ 時間で実行できる. よって, 補題 1 と合わせて次の定理を得る.

[定理 1] 木パターンマッチングはパターン木とテキスト木のサイズがそれぞれ m, n のとき, CREW-PRAM 上で $mn/\log n$ 台のプロセッサを使って $O(\log n)$ 時間で計算することができる.

5. む す び

サイズがそれぞれ m と n のパターン木とテキスト木に前処理を行うことによって, CREW-PRAM 上で $mn/\log n$ 台のプロセッサを用いて $O(\log n)$ 並列時間で木パターンマッチングができることを示した. 項が順

序木や配列でなく，dagで与えられる場合の効率の良い並列アルゴリズムは今後の課題である．

謝辞 貴重な御意見と文献を御教示頂いたお2人の査読者に感謝致します．なお，本研究の一部は文部省科学研究費補助金によって行われた．

文 献

(1) Aho A. V. and Ganapathy M. : "Efficient tree pattern matching : an aid to code generation", Proc. 18th ACM Symp. on Principles of Programming Languages, pp. 334-340 (1984).

(2) Anderson R. J. and Miller G. L. : "Deterministic parallel list ranking", Lecture Notes in Computer Science, **319**, pp. 81-90, Springer-Verlag (1988).

(3) Brent R. P. : "The parallel evaluation of general arithmetic expressions", J. Assoc. Comput. Mach. **21**, 2, pp. 201-206 (1974).

(4) Burghardt J. : "A tree pattern matching algorithm with reasonable space requirements", Lecture Notes in Computer Science, **299**, pp. 1-15, Springer-Verlag (1988).

(5) Chase D. R. : "An improvement to bottom-up tree pattern matching", Proc. 14th ACM Symp. on Principles of Programming Languages, pp. 168-177 (1987).

(6) Cole R. and Vishkin U. : "Deterministic coin tossing with applications to optimal parallel list ranking", Information and Computation, **70**, pp. 32-53 (1986).

(7) Cole R. and Vishkin U. : "Approximate and exact parallel scheduring with applications to list, tree and graph problems", Proc. 27th IEEE Symp. on Foundations of Computer Science, pp. 478-491 (1986).

(8) Dwork C., Kanellakis P. C. and Stockmeyer L. : "Parallel algorithms for term matching", SIAM J. Comput., **17**, 4, pp. 711-731 (1988).

(9) Dubiner M., Galil Z. and Magen E. : "Faster tree pattern matching", Proc. 31th IEEE Symp. on Foundations of Computer Science, pp. 145-149 (1990).

(10) Galil Z. : "Optimal parallel algorithms for string matching", Information and Control **67**, pp. 144-157 (1985).

(11) Gibbons A. and Rytter W. : "Efficient Parallel Algorithms", Cambridge University Press (1988).

(12) Guttag J., Horowitz E. and Musser D. R. : "Abstract data types and software validation", Commun. ACM, **21**, 12, pp. 1048-1064 (1978).

(13) 平田富夫，稲垣康善："木パターンマッチングアルゴリズム"，情処学研究報告，88-AL-4-2 (1988).

(14) Hoffmann C. M. and O'Donnell M. J. : "Pattern matching in trees", J. Assoc. Comput. Mach. **29**, 1, pp. 68-95 (1982).

(15) Hoffmann C. M. and O'Donnell M. J. : "An interpreter generator using tree pattern matching", Proc. 6th ACM Symp. on Principles of Programming Languages, pp. 169-179 (1979).

(16) Karp R. M., Miller R. E. and Rosenberg A. : "Identification of repeated patterns in strings, trees and arrays", Proc. 4th ACM Symp. on Theory of Computing, pp. 125-136 (1972).

(17) Knuth D., Morris J. and Pratt V. : "Fast pattern matching in strings", SIAM J. Comput., **6**, 2, pp. 323-350 (1977).

(18) Kojima K. : "A pattern matching algorithm in binary trees", Lecture Notes in Computer Science, **147**, pp. 99-114 (1983).

(19) Kosaraju S. R. : "Efficient tree pattern matching", Proc. 30th IEEE Symp. on Foundations of Computer Science, pp. 178-183 (1989).

(20) 宮野 悟："並列化とその限界—理論的側面から"，コンピュータソフトウェア，**7**, 1, pp. 2-15 (1990).

(21) Pippenger N. : "On simultaneous resource bounds", Proc. 20th IEEE Symp. on Foundations of Computer Science, pp. 307-311 (1979).

(22) Ramesh R. and Ramakrishnan I. V. : "Optimal speedups for parallel pattern matching in trees", Proc. Rewriting Technique and Applications (Lecture Notes in Computer Science 256), Springer, pp. 274-285 (1987).

(23) Ramesh R. and Ramakrishnan I. V. : "Parallel tree pattern matching", J. Symbolic Computation, **9**, 4, pp. 485-501 (1990).

(24) Ramesh R., Verma R. M., Krishnaprasad T. and Ramakrishnan I. V. : "Term matching on parallel computers", J. Logic Programming, **6**, 3, pp. 213-228 (1989).

(25) Stockmeyer L. and Vishikin U. : "Simulation of parallel random access machines by circuits", SIAM J. Comput., **13**, pp. 409-422 (1984).

(26) Tarjan R. E. and Vishkin U. : "An efficient parallel biconnectivity algorithm", SIAM J. Comput., **14**, 4, pp. 862-874 (1985).

(27) Vishkin U. : "Optimal parallel pattern matching in strings", Information and Control, **67**, pp. 91-113 (1985).

(平成3年7月16日受付，4年3月3日再受付)

太郎良浩次

平3名大・工・情報卒．同年(株)リクルート入社．在学中，並列アルゴリズムに関する研究に従事．

平田　富夫

昭51東北大・工・通信卒．昭56同大大学院博士課程了．工博．豊橋技術科学大学助手を経て，現在，名大・工・情報工学科助教授．昭59～60英国ウォーリック大客員研究員（ブリティッシュカウンシルスカラー）．グラフアルゴリズム，データ構造，並列アルゴリズムの研究に従事．著書「アルゴリズムとデータ構造」（森北出版）．

稲垣　康善

昭37名大・工・電子卒．昭42同大大学院博士課程了．同大助教授，三重大教授を経て，昭56年より名大教授．この間，スイッチング回路理論，オートマトン・言語理論，計算論，ソフトウェア基礎理論，並列処理理論，代数的仕様記述・検証とプログラム自動生成などの研究に従事．工博．40年度後期稲田賞受賞．著書「符号理論」（コロナ社，共著），「オートマトン・形式言語理論と計算論」（岩波書店，共著）など．IEEE，ACM，EATCS，情報処理学会，電気学会，日本ソフトウェア科学会，人工知能学会，日本OR学会各会員．