木スケルトンによるXPathクエリの並列化とその評価

Parallelization of XPath Queries with Tree Skeletons

野村 芳明 †, 江本 健斗 †, 松崎 公紀 †, 胡 振江 †, 武市 正人 †
Yoshiaki NOMURA, Kento EMOTO, Kiminori MATSUZAKI, Zhenjiang HU, Masato TAKEICHI

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
{Yoshiaki_Nomura,Kento_Emoto,kmatsu,hu,takeichi}@mist.i.u-tokyo.ac.jp

XPath は XML の要素指定に広く用いられる言語であり，XPath クエリの効率的な実装が求められている．しかし，XML のような木を扱う計算を効率的に並列化することは簡単ではなく，XPath クエリの並列処理の研究はまだ少ない．そこで，本論文では XPath クエリをオートマトンに変換し，木スケルトンと呼ばれる並列計算パターンを用いることで，効率的な並列化を実現した．さらにこれを実装し評価実験によってその有効性を確認した.

1 はじめに

XML は W3C が規定したタグを用いた文書構造化のための汎用フォーマットである．これはデータ記述言語として広く利用されており，大規模応用データの保管形式としても利用されはじめている．XML を対象とした検索や変換，操作を行うための言語として，XPath [1] や XSLT [10]，XQuery [3] が広く利用されている．これらの中でも，XML データを表す木構造を辿ることで要素や属性といったデータを抽出する XPath は，他の XML 処理言語である XSLT や XQuery の基礎をなすものであり，XPath によるクエリ処理に対する様々な効率化，最適化の研究がなされている [14, 15, 8, 6, 5, 9, 7].

一方，XML は任意数の子を許すような木であり，このような不均等な木に対して，効率的に並列計算を行うことは一般的に簡単ではない．それゆえ，XPath クエリを並列に処理することによって効率化するというアプローチは非常に少ない [19, 11].

そこで本研究では，スケルトン並列プログラミング [4, 17] という並列計算パラダイムの上で， 効率的な XPath によるクエリ処理を実現した．スケルトン並列プログラミングでは，並列スケルトンと呼ばれる， 特定のデータ構造に対する基本的な並列計算パターンを組み合わせてプログラムを構成する．並列スケルトンは高階関数としてのインターフェースを持ち，プロセッサ間通信，データや計算資源の配置，同期処理などをその中に隠蔽しているため，ユーザは並列性をそれほど意識せずに比較的効率の良い並列プログラムを作成することができる．本研究では，木に対する並列スケルトン [18, 13] を用いる．

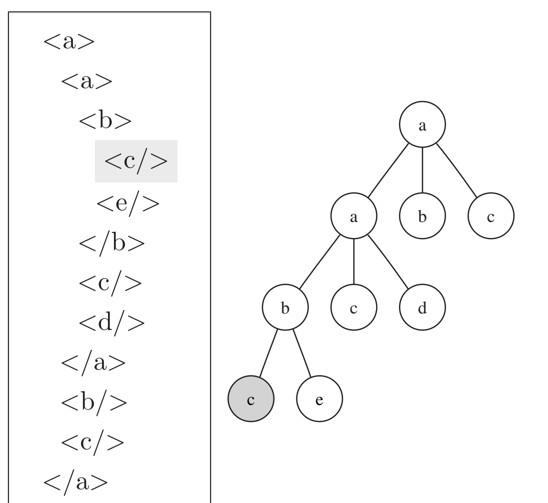
本論文では，親子，子孫／祖先，兄弟といった軸によるロケーションパスだけでなく， これらのロケーションパスや位置指定関数による述語による絞り込みを含むような一般的な XPath クエリを並列に処理する方法を述べる．一般に，このような XPath クエリは，祖先要素に関する条件と子孫要素に関する条件で要素を指定する．例えば，

```
/descendant::a[following-sibling::b]
```

という XPath は，“/descendant::a” という祖先の要素に関する条件と，“following-sibling::b” という子孫の要素に関する条件とで要素を指定する．

XPath クエリを並列化する基本的なアイデアは次の通りである．まず，与えられた XPath 中に祖先方向の軸が存在する場合，クエリ処理の並列化が難しくなるため，子孫方向の軸のみからなる等価な XPath に変換する．これによって，クエリで指定される要素から見ての祖先方向の要素に関する条件と子孫方向の要素に関する条件を明確に分離することができる．次に， 祖先の要素に関する条件は， 木の根から葉への計算を抽象化した並列スケルトンを用いることによって処理を行い， 子孫要素に関する条件は，葉から根への計算を抽象化した並列スケルトンを用いることによって計算を行う．これらの並列スケルトンに渡すための関数は， 条件に対応するオートマトンの状態遷移から自動的に求めることができる．

本論文では， 一般的な XPath クエリを並列化する



“/descendant::a/descendant::b/child::c”

図 1: XML 及び XPath の例

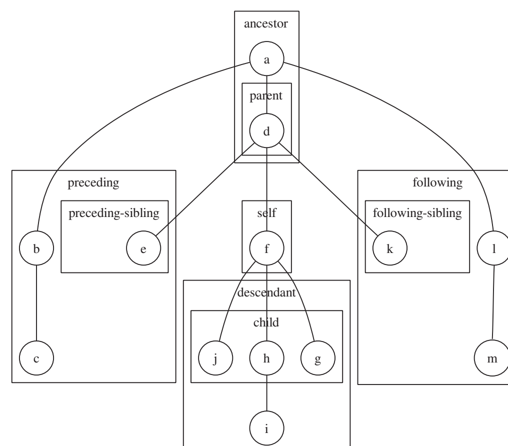


図 2: XPath の Axis

ためのアルゴリズムを示す．また， これを実装し評価実験を行った結果を述べる． 本手法によって並列化された XPath クエリは， 良い台数効果を示した．

本論文の構成を以下に示す．第 2 節では XPath について述べ，扱う XPath の範囲を定める．第 3 節では今回用いる並列スケルトンについて述べる．第 4 節では並列スケルトンを用いて XPath クエリを行う方法について述べる．第 5 節では第 4 節の手法を実装し評価実験を行い， その結果を示す．第 6 節ではまとめと今後の課題について述べる．

2 XPath

本節では XPath について簡単に説明し，本論文で対象とする XPath を定める．

近年，商業の情報化および計算機性能の向上にともない，大量のデータがデータベースに蓄えられるようになってきた．そのため，木構造データベースの表現の一種である XML 文書のサイズは非常に巨大なものとなり，それをユーザが直接扱うことは容易ではない場合が多い．そこで，XML 文書中の特定箇所のみを抽出する機構が必要になる．XML 文書は木構造をなすため，その特定要素を指し示すのにルート要素からの経路を用いることができる．そのような経路記述言語として XPath が W3C によって勧告され，XSLT および XQuery などで広く使用されている．例えば，XPath `/descendant::a/descendant::b/child::c` により，図 1 に示される XML と対応する木構造に対して影のついた要素が指し示される．このように XPath がある要素を指し示している場合，その要素は XPath にマッチしていると言う．また，ひとつの XPath にマッチする要素は複数存在し得る．

XML 文書中の各々の要素と，その XML の木構造のノードは一対一に対応しているため，以後，本論文ではノードと要素という言葉を区別せずに用いる．

図 3 に本論文で扱う XPath の構成を BNF で示す．XPath は複数のステップからなり，ステップはコンテクストノードからの方向を示す軸 (Axis)，ノードの種類や名前を表すノードテスト，ノードを絞り込むための条件である述語 (Predicate) から構成される．さらに，軸については子，子孫を表す `child`，`descendant`，親，祖先を表す `parent`，`ancestor`，文書中で自分自身より後に存在するノードを表す `following`，その反対の `preceding`，弟ノードを表す `following-sibling`，兄ノードを表す `preceding-sibling`から構成される．各軸の関係は図 2 のようになっている．

また，XML 文書はあるノードが同じ名前の子を複数持ちうるため，それらを区別しようとすれば，どの位置に存在するのかという情報を扱う必要がある．本論文では，そのような位置に関する述語記述を行うための関数 `position`と `last`も扱う対象として含める．

XPath ::= “/” *Step* | “/” *Step XPath*
Step ::= *LocationStep* | *LocationStep* “[” *Predicate* “]”
LocationStep ::= *Axis* “::” *NodeTest*
Axis ::= *ForwardAxis* | *ReverseAxis*
ForwardAxis ::= “child” | “descendant” | “following””| “following-sibling”
ReverseAxis ::= “parent” | “ancestor” | “preceding” | “preceding-sibling”
NodeTest ::= *String* | *Wildcard*
Wildcard ::= “*”
Predicate ::= *LocationPath* | *Function*
LocationPath ::= *LocationStep* | *LocationStep* “/” *LocationPath*
Function ::= “position()” *BinOp Exp*
BinOp ::= “=” | “!=” | “<” | “<=” | “>” | “>=”
Exp ::= *Num* | “last()” | “last()” “-” *Num*

図 3: 本論文で扱う XPath

3 スケルトン並列プログラミング

本節では，まず，表記法などを説明した後，XPath のクエリ処理を記述するための並列スケルトンの定義を示す．

3.1 用語および表記法

まず最初に，本論文全体に共通の用語および表記法について説明する．なお，これらの表記法は関数型プログラミング言語 Haskell[2] に基づいている．

関数

関数適用は，関数と引数の間に空白を置くことによって表す．すなわち，$f\ a$ は $f(a)$ を意味する．関数はカリー化されており，左結合的である．従って，$f\ a\ b$ は $(f\ a)\ b$ の意味である．二項演算子は $\oplus$，$\otimes$ などで表し，$(a\oplus)$，$(\oplus b)$，$(\oplus)$ のように括弧で囲むことにより関数とすることができる．

$$a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$$

関数適用は最も優先順位が高く，$f\ a \oplus b$ は $f\ (a \oplus b)$ ではなく $(f\ a) \oplus b$ となる．

リスト

リストは同じ型のデータが一列に並んだものであり，空リストか，リストに要素を付け加えたものとして表現される．要素のデータ型が α であるリストのデータ型は次のように定義される．

$$\begin{aligned} \textbf{data}\ \mathit{List}\ \alpha &= \mathit{Nil} \\ &\mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha) \end{aligned}$$

リストについては，次に示すような省略記号を用いることがある．データ型 α に対して $[\alpha]$ は $\mathit{List}\ \alpha$ を表し，適当な要素 a とリスト as に対して $[\,]$，$(a : as)$ はそれぞれ Nil，$\mathit{Cons}\ a\ as$ を表す．また，リストの略記法として，リストの各要素を並べて次のように表記する．

$$1 : 2 : 3 : 4 : [\,] = [1, 2, 3, 4]$$

二分木

二分木は内部ノードが全てちょうどふたつの子を持つような木である．要素のデータ型が α であるような二分木のデータ型は次のように定義される．

$$\begin{aligned} \textbf{data}\ \mathit{BTree}\ \alpha &= \mathit{Leaf}\ \alpha \\ &\mid \mathit{Node}\ \alpha\ (\mathit{BTree}\ \alpha)\ (\mathit{BTree}\ \alpha) \end{aligned}$$

rose tree

rose tree は内部ノードが任意数個の子を持つような木である．要素のデータ型が α であるような rose tree のデータ型はリストを用いて次のように定義される．

$$\textbf{data}\ \mathit{RTree}\ \alpha = \mathit{RNode}\ \alpha\ [\mathit{RTree}\ \alpha]$$

$$
\begin{aligned}
&\mathsf{map}\ k_L\ k_N\ (\mathit{Leaf}\ n) &&= \mathit{Leaf}\ (k_L\ n)\\
&\mathsf{map}\ k_L\ k_N\ (\mathit{Node}\ n\ l\ r) &&= \mathit{Node}\ (k_N\ n)\ (\mathsf{map}\ k_L\ k_N\ l)\ (\mathsf{map}\ k_L\ k_N\ r)\\
&\mathsf{zip}\ (\mathit{Leaf}\ n)\ (\mathit{Leaf}\ n') &&= \mathit{Leaf}\ (n, n')\\
&\mathsf{zip}\ (\mathit{Node}\ n\ l\ r)\ (\mathit{Node}\ n\ l\ r) &&= \mathit{Node}\ (n, n')\ (\mathsf{zip}\ l\ l')\ (\mathsf{zip}\ r\ r')\\
&\mathsf{reduce}\ k_L\ k_N\ (\mathit{Leaf}\ n) &&= k_L\ n\\
&\mathsf{reduce}\ k_L\ k_N\ (\mathit{Node}\ n\ l\ r) &&= k_N\ n\ (\mathsf{reduce}\ k_L\ k_N\ l)\ (\mathsf{reduce}\ k_L\ k_N\ r)\\
&\mathsf{uAcc}\ k_L\ k_N\ (\mathit{Leaf}\ n) &&= \mathit{Leaf}\ (k_L\ n)\\
&\mathsf{uAcc}\ k_L\ k_N\ (\mathit{Node}\ n\ l\ r) &&= \mathit{Node}\ (\mathsf{reduce}\ k_L\ k_N\ (\mathit{Node}\ n\ l\ r))\ (\mathsf{uAcc}\ k_L\ k_N\ l)\\
& && \qquad (\mathsf{uAcc}\ k_L\ k_N\ r)\\
&\mathsf{dAcc}\ (\oplus)\ g_l\ g_r\ c\ (\mathit{Leaf}\ n) &&= \mathit{Leaf}\ c\\
&\mathsf{dAcc}\ (\oplus)\ g_l\ g_r\ c\ (\mathit{Node}\ n\ l\ r) &&= \mathit{Node}\ c\ (\mathsf{dAcc}\ (\oplus)\ g_l\ g_r\ (c \oplus g_l\ n)\ l)\\
& && \qquad (\mathsf{dAcc}\ (\oplus)\ g_l\ g_r\ (c \oplus g_r\ n)\ r)
\end{aligned}
$$

図 4: 二分木に対する並列スケルトンの定義

3.2 二分木に対する並列スケルトン

二分木上の並列スケルトン [18] は二分木に対する基本的な操作を並列に行うものである．二分木に対する重要な並列スケルトンは，map，zip，reduce，uAcc，dAcc の 5 つである．これらの形式的な定義は図 4 に示すとおりである．本論文では，主にこれらを組み合わせることで，XPath のクエリ処理を実現する．

並列スケルトン map はふたつの関数 k_L，k_N を受け取り，二分木の全ての葉に関数 k_L を，全ての内部ノードに関数 k_N を適用する．並列スケルトン zip は同じ形のふたつの木を受け取り，対応するノードを組にした木を返す．並列スケルトン reduce は木を受け取り，葉に対して k_L を，内部ノードに対して k_N をそれぞれ適用しながら，ボトムアップな計算により縮約してひとつの値を返す操作である．並列スケルトン uAcc は各ノードに対して，そのノードをルートノードとするような部分木に対して reduce を適用した値を割り当てた木を返す．並列スケルトン dAcc はルートノードから葉に向かって，値を累積させていく計算である．値の更新は，二項演算子 $\oplus$，左の子への関数 g_l，右の子への関数 g_r を使って行われる．

これらの並列スケルトンのいくつかは，効率的な実装を保証するために条件を課している [12]． 並列スケルトン reduce，uAcc では，その引数である関数 k_N について，次の等式を満たす関数 ϕ，ψ_L，ψ_R，G が存在することが必要である．

$$
\begin{aligned}
k_N\ n\ l\ r &= G\ (\phi\ n)\ l\ r\\
G\ n\ l\ (G\ r_n\ r_l\ r_r) &= G\ (\psi_L\ n\ l\ r_n)\ r_l\ r_r\\
G\ n\ (G\ l_n\ l_l\ l_r)\ r &= G\ (\psi_R\ n\ r\ l_n)\ l_l\ l_r
\end{aligned}
$$

また，並列スケルトン dAcc では，二項演算子 $\oplus$ は単位元を持つ結合的な演算子であることが必要である．

これらの条件が成立するとき並列スケルトンは効率的に実行できる． 例えば， ノード数 n の木に対してプロセッサが十分に使用できるときには， map，zip は $O(1)$ の並列時間で実行でき， reduce，uAcc，dAcc は $O(\log n)$ の並列時間で実行できる．

4 XPath クエリの並列化

本節では，木スケルトンを用いて XPath クエリを実現する方法を示す．与えられた XPath は [14] の rare アルゴリズムによって ReverseAxis を含まない形に変換されているものとし，以降では ForwardAxis で表された XPath でのクエリを扱う．

4.1 `child`, `descendant`のみからなる XPath クエリの並列化

最初に最も簡単な XPath クエリとして `child`, `descendant`のみで構成されるものを扱う．説明には例として

```
/descendant::a/descendant::b/child::c
```

という XPath を用いる．この XPath は例えば図 1 の XML に対して影のついたノードを指定する．

表 1: child，descendantに対応する正規表現

child::a	a
descendant::b	.*b

各ノードが XPath にマッチするかどうかの判定は，木のノードをルートから各ノードまで辿った結果得られるものであるから，この処理は dAcc で表せると考えられる．Skillicorn[19] は親子関係のみの簡単なクエリに対して，ルートノードからマッチするノードまで辿る際のノード列を受理するようなオートマトンを用意し，その遷移状態の表の計算を考えることで，dAcc のための関数および演算子を与えた．ここではこれを拡張し，子孫関係についても対応するオートマトンを求め，同様の計算を行う．

オートマトンは，XPath に対応する正規表現を作り，それを変換することで求めることができる．child は次に辿るノードを，descendant は何個か後に辿るノードを指定するので，それぞれに対応する正規表現は表 1 のようになる．従って，例に挙げた XPath に対する正規表現は “.*a.*bc” となり，これを変換して得られる決定性有限状態オートマトンは図 5 のようになる．初期状態は S_0，受理状態は S_3 であり，遷移の#は a，b，c 以外のものに対する遷移である．

決定的有限オートマトンは，表 2 のように，現在の状態と読み込む文字に対する遷移先状態の表として表現できる．そのため，dAcc による計算では，ある入力に対応した遷移表の一列を，遷移前の状態と遷移後の状態の組 (s_{in}, s_{out}) のリストとして扱う．XPath クエリ処理は一般の木（rose tree）に対する dAcc である $\mathsf{dAcc_r}$ を用いて以下のように表される．ただし，二分木の場合とは異なり，子に適用する関数をひとつしか取らないことに注意する．

$$\mathrm{dAcc_r}\ (\oplus)\ \mathit{maketable}\ \mathit{idtable}$$

ここで $\mathit{maketable}$ は，表 2 のようなオートマトンの遷移表からノードの値に対応した列を返す関数であり，$\mathit{idtable}$ は恒等写像を表す遷移表である．また，$\oplus$ は表を結合することで次のノードの値に対する遷移を計算する演算子であり，次のように定義される．

$$\begin{aligned} t_1 \oplus t_2 = \{(s_{in1}, s_{out2}) \mid\ & (s_{in1}, s_{out1}) \leftarrow t_1, \\ & (s_{in2}, s_{out2}) \leftarrow t_2 \\ & \textbf{where}\ s_{out1} = s_{in2}\} \end{aligned}$$

$\oplus$ は結合性を持ち，単位元は $\mathit{idtable}$ であるから，この dAcc は並列に計算することができる．

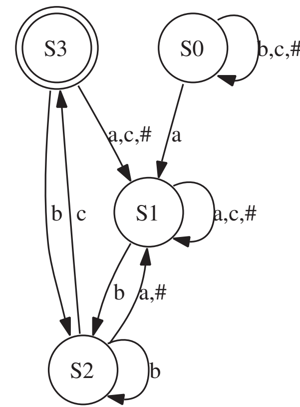


図 5: “/descendant::a/descendant::b/child::c” に対応するオートマトン

表 2: 図 5 のオートマトンに対応する表

	a	b	c	#
S_0	S_1	S_0	S_0	S_0
S_1	S_1	S_2	S_1	S_1
S_2	S_1	S_2	S_3	S_1
S_3	S_1	S_2	S_1	S_1

この計算により各ノードに対して，ルートノードから辿ったときのオートマトンの遷移先が得られる．XPath がマッチするノードは，初期状態 S_0 からの遷移先が受理状態であるノードとなる．以上の処理により目的のノードを見つけることができる．

4.2 following-siblingを含む XPath クエリの並列化

次に child，descendantに following-sibling，followingを加えた全 ForwardAxis から構成される XPath を考える．

dAcc や uAcc は木を親子関係で上下に辿る計算であり，兄弟関係のような横方向の処理にはそのまま用いることはできない．そこで，XML 木に対して兄弟方向へ辿ることの出来る木への変換を行う．図 6 は図 1 に変換を施した様子である．変換された木は二分木になっており，元の木における長男，弟の関係が変換された木では左の子，右の子の関係になっている．各ノードには左の子であるか右の子であるかというラベル，L，R が付加されており，これによ

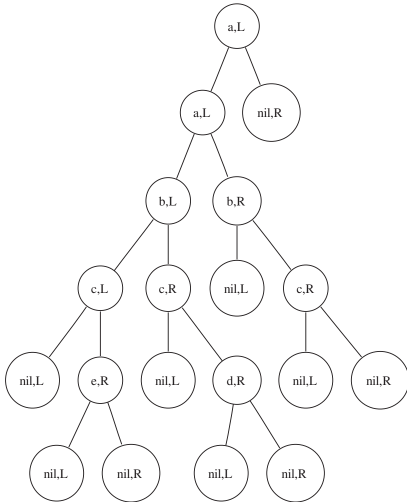


図 6: XML 木の二分木表現

表 3: 二分木表現における各軸に対応する正規表現

`child::a`	(.,L)(.,R)*(a,R) | (a,L)
`descendant::b`	(.,L)(.,.)*(b,.) | (b,L)
`following-sibling::c`	(.,R)*(c,R)

り，変換後の二分木における親子関係が，元の木において兄弟，親子どちらの関係なのかは識別可能である．

変換後の木において `child`，`descendant`，`following-sibling`は表 3 のように対応する正規表現が存在する．すなわち，`child`は最初に長男方向に辿った後，弟方向に何回か辿って得られるノード，`descendant`は最初に長男方向に辿った後，任意の方向に何回か辿って得られるノード，`following-sibling`は弟方向に何回か辿って得られるノードに相当する．従って，これら 3 つの軸に関しては `child`，`descendant`のみの場合と同様の計算によってクエリ処理を行うことができる．

`following`に関しては祖先，子孫，兄弟の 3 方向を辿らなければならないため，そのものを扱うことは難しい．しかし，“`following::a`” は

```
ancestor-or-self::*/following-sibling::*
/descendant-or-self::a
```

と書き換えることにより除去することができる．従って，ここでは陽に扱わないことにする．

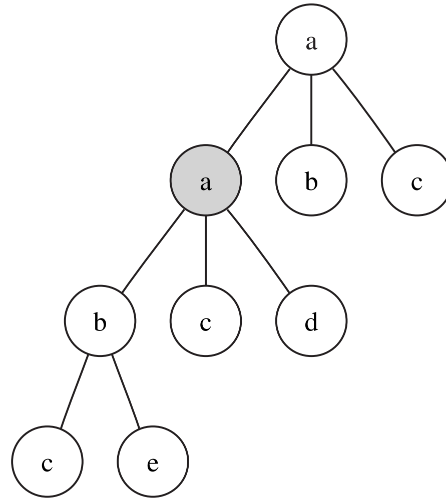


図 7: “`/descendant::a[following-sibling::b]`” にマッチするノード

4.3 述語を含む XPath クエリの並列化

最後に述語が指定された XPath を考える．ただし，ここで考えるのは

```
/descendant::a[following-sibling::b]
```

のような述語がひとつであり，述語に先行するロケーションパスが 1 ステップのみものである．すなわち，指定しているノードに対し，その子孫のノードに関して条件がついているものを考える．例に挙げた XPath がマッチするのは図 7 に示したノードである．

ここまでで扱った XPath は祖先のノードに関して条件がついており，ルートから葉の方向への計算を行う dAcc で扱った．従って，子孫のノードに関して条件のつく述語を含むパスは，葉からルート方向への計算を行う uAcc で扱えると考えられる．やはりここでも，オートマトンの状態遷移表を用いてこれを実現する．

マッチさせるノードに述語部分もひとつながりにした XPath,

```
/descendant::a/following-sibling::b
```

に対応するオートマトンを考える．このオートマトンは図 8 のようになる．弟方向の軸も扱うのでオートマトンは二分木表現に対するものである．この時，初期状態から子孫を辿ることによって受理状態まで遷移でき得るようなノードが該当するノードである．従って，各ノードにおいて，各状態から遷移でき得る状態が分かればよい．

あるノードから遷移でき得る状態集合は，そこのノードの値で遷移した後，左に辿った場合のそこか

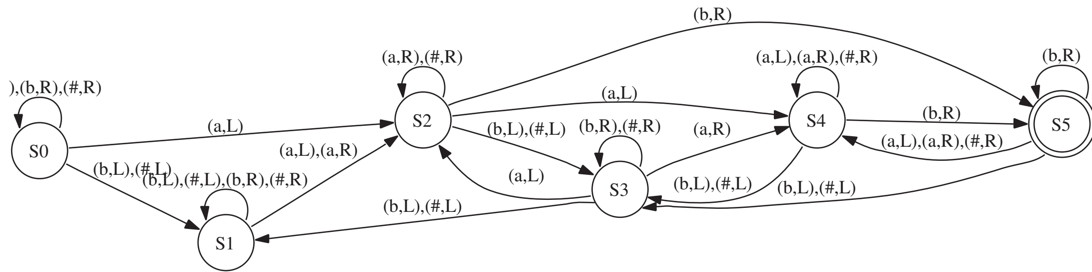


図 8: “`/descendant::a[following-sibling::b]`” に対応するオートマトン

表 4: uAcc で用いる表

	(a,L)	(a,R)	(b,L)	(b,R)	(#,L)	(#,R)
S_0	$\{S_2\}$	$\{S_0\}$	$\{S_1\}$	$\{S_0\}$	$\{S_1\}$	$\{S_0\}$
S_1	$\{S_2\}$	$\{S_2\}$	$\{S_1\}$	$\{S_1\}$	$\{S_1\}$	$\{S_1\}$
S_2	$\{S_4\}$	$\{S_2\}$	$\{S_3\}$	$\{S_5\}$	$\{S_3\}$	$\{S_2\}$
S_3	$\{S_2\}$	$\{S_4\}$	$\{S_3\}$	$\{S_3\}$	$\{S_1\}$	$\{S_3\}$
S_4	$\{S_4\}$	$\{S_4\}$	$\{S_3\}$	$\{S_5\}$	$\{S_3\}$	$\{S_4\}$
S_5	$\{S_4\}$	$\{S_4\}$	$\{S_3\}$	$\{S_5\}$	$\{S_3\}$	$\{S_4\}$

らの遷移先集合と，右に辿った場合のそこからの遷移先集合の和集合となる．このことから，動的計画法を用いてボトムアップに遷移先集合を求める計算を，uAcc によって記述してやればよい．今回は，遷移前の状態に対し遷移後の状態が複数あるので，計算で扱う値は表 2 のような表に対し要素を集合にした 1 次元多い表 4 のようなものとなる．この処理を以下に示す．

uAcc k_L k_N
　　where $k_L\ n \quad = \mathit{maketable}\ n$
　　　　　$k_N\ n\ l\ r = \mathit{maketable}\ n\ \oplus\ (l\ \otimes\ r)$

ここで *maketable* は各ノードの値に対し表 4 のような表を返す関数であり，$\otimes$ は表同士を集合和をとってマージする演算子，$\oplus$ は表の遷移を結合する演算子である．

ここまでの議論を組み合わせ，全ての表現を含む XPath の処理を説明する．ここでは例として

```
/descendant::a[descendant::b]/child::c
/preceding-sibling::e
```

という XPath を用いる．

まずは，前処理として `following`や ReverseAxis を等価な XPath に書き換え，これらを含まないものにする．これにより，XPath は

```
/descendant::a[descendant::b]
/child::d[following-sibling::e]
```

となる．また，XML 木は二分木表現に変換しておく．

次に XPath を述語で分解し，

```
/descendant::a[descendant::b] ,
/child::d[following-sibling::e]
```

のそれぞれにマッチするノードを uAcc によって求める．この計算は述語の数だけ別々に uAcc を繰り返すと効率が悪いので，各分解された XPath で扱う遷移表を全て組にしたものを計算に用いることで 1 回の uAcc でまとめて行う．

最後に述語を満たしている `a` を `a’`，述語を満たしている `d` を `d’` とし，“`/descendant::a’/child::d’`” を dAcc で求めてやればよい．

以上より，uAcc と dAcc を 1 回ずつ行うことにより，全てのロケーションパスのクエリ処理を効率的に並列化することができる．

4.4　関数を含む XPath クエリの並列化

XPath ではノードを指定する際に関数を使用することができる．XPath の関数のうち，ノードの位置指定をするのに使用される関数，`position`，`last`について，それらを並列処理するためのアルゴリズムを以下に与える．

`position`関数はノードが何番目の子供であるかを指定する．例えば図 9 の XML に対し，

```
/descendant::c[position()=last()]
```

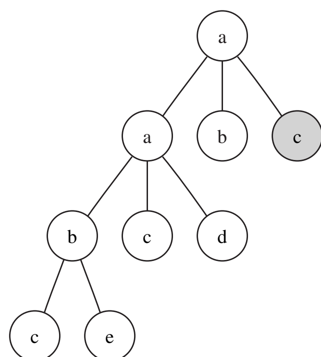


図 9: “/descendant::c[position()=last()]” にマッチするノード

という XPath クエリは影のついたノードを選択する．ただし，last関数はその兄弟の中で最も右端にあるノードの position関数の値を返す関数である．

各ノードが何番目の子供であるかを求めるには弟方向にノードを辿る必要がある．したがって，二分木表現に変換された木 (図 6) に対し，図 10 のような結果を得る処理を与えればよい．すなわち，弟方向に辿る際に値を 1 増やし，長男方向に辿る際に値を 1 に戻しながら，ルートノードから葉に向かって値を計算する．これは，以下の dAcc スケルトンによって行うことが可能である．

$$
\begin{aligned}
&\mathsf{dAcc}\ (\oplus)\ g_l\ g_r\ 1\\
&\qquad \textbf{where}\ g_l\ x \ = 1\\
&\qquad\qquad g_r\ x \ = -1\\
&\qquad\qquad a \oplus b = \mid b > 0 \qquad\qquad\quad = b\\
&\qquad\qquad\qquad\quad \mid b \leq 0\ \&\&\ a > 0 = a - b\\
&\qquad\qquad\qquad\quad \mid b \leq 0\ \&\&\ a \leq 0 = a + b
\end{aligned}
$$

この $\oplus$ は結合的な演算子であり，上記の dAcc スケルトンは効率的に計算することができる．

positionの値として lastが指定されている場合にはこの値も求める必要がある．この計算は position を求めた後の木 (図 10) に対して次の uAcc スケルトンを使って求めることができる．

$$
\begin{aligned}
&\mathsf{uAcc}\ k_L\ k_N\\
&\qquad \textbf{where}\ k_L\ n \qquad = 0\\
&\qquad\qquad k_N\ n\ l\ r = \textbf{if}\ r == 0\ \textbf{then}\ n\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ r
\end{aligned}
$$

lastの値はその兄弟における右端のノード，すなわち弟を持たないノード，の値である．弟を持たないノードにおいてその右の子は葉であるので，k_N は葉に対する返り値 0 によって分岐し，右端のノードの場合は自分の値を，それ以外のノードの場合は弟の結果を返すようにしている．この関数が並列スケルトンの条件を満たすことは比較的簡単に検証できる．適用結果は図 11 のようになる．

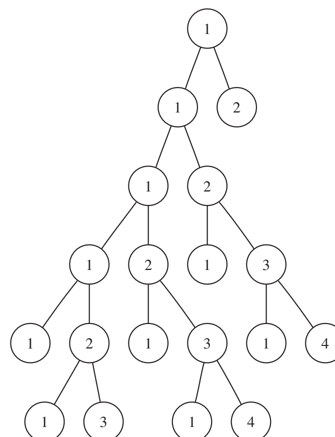


図 10: position関数の適用結果

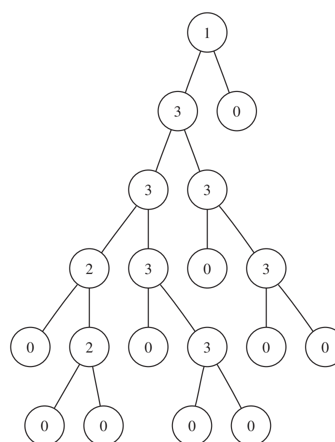


図 11: last 関数の適用結果

以上によりすべてのノードに対して，position，last関数を計算することができた．この値を使うことによって述語に位置指定関数が与えられた XPath クエリに対する計算も行うことができる．すなわち，述語に位置指定関数を含む場合は，前処理として上記の uAcc スケルトン，dAcc スケルトンを実行し，これらの結果を zip スケルトンによってもとの木の各ノードに割り当てておく．こうすることで，位置指

定関数を用いた述語の処理を，通常のロケーションパスにおけるノードの名前と同様にして実行することが可能となる．

5 評価実験

本節では，前節で示した方法に基づいて XPath クエリを並列に処理するプログラムを実装し，評価実験によってその並列化の効果を確認する．

5.1 プログラムの実装

実装したプログラムは以下のような動作を行う．まず，入力として XPath を受け取り，対応するオートマトンを生成する．次に，その結果といくつかの関数を並列スケルトンに渡し，クエリ対象の XML に対して必要な計算を並列に実行させる．最後に，並列スケルトンから計算結果を受け取り，その結果をユーザに返す．なお，並列スケルトンの実装としては我々の開発した並列スケルトンライブラリ「助っ人」[16] を用いている．

5.2 実験内容

分散並列計算機環境として，16 台の均質な PC から構成される PC クラスタを用いた．各 PC は Pentium4 3.0GHz (Hyper Threading ON) の CPU, 1GB のメモリを持ち，これらがギガビットイーサネットで接続されている．OS は Linux 2.6.8，コンパイラは gcc 2.95，MPI ライブラリは mpich 1.2.6 を用いた．

実験は，表 5 の XPath クエリを表 6 に示す XML データに対して並列実行し，その実行時間を測定する．ただし，既に分散された XML に対するクエリの並列実行を想定し，データの分散にかかる時間は実行時間に含めない．

実験に用いる XML は，適度な幅と深さをもつ “normal.xml”，極端に高い “mono.xml”，極端に低く幅の広い “flat.xml” という特徴的な形をもつデータである．また，各々のノード数は 10000 である．

本実験では 2 のべき乗個（$1, 2, 4, 8, 16$）のプロセッサを使用する．ただし，実装したプログラムは任意のプロセッサ数で実行可能であることを注意する．

5.3 実験結果

測定された実験結果を表 7 に示す．表中において P は使用したプロセッサ数を表しており，時間の単位は秒である．表中の比は一台のプロセッサでの実行時間を各々の台数での実行時間で割ったもの（速度向上）である．使用したプロセッサ数に対する速度向上を図 12，図 13 に示す．縦軸が速度向上，横軸が使用したプロセッサ数であり，参考のために線形の場合の直線を並べて表示している．

小さな XPath に対する結果の図 12 では “normal.xml” に関して使用する台数にほぼ比例する形で速度向上が起こっており，並列化による台数効果が確認できる．また，極端な形である “mono.xml” および “flat.xml” に関しても，16 台のプロセッサで 10 倍程度の速度向上があり，極端なデータに対する台数効果も確認できる．

同様に，大きな XPath に対する結果の図 13 においても全てのデータに対して台数効果が確認できる．また，速度向上が超線形となっているが，これは大きな XPath の処理のためにメモリを多く必要とするため，一台のプロセッサでの処理が遅くなることによる．このような現象は使用メモリ量が計算機のメモリ量と比べて大きくなる場合にしばしば見られ，並列化による効果が大きいことを示している．

いずれの結果も台数効果がはっきりと現れており，提案するアルゴリズムによる並列化が有効であることが示された．

6 まとめ

本論文では，ロケーションパスや位置指定関数によって定義された XPath クエリを木に対するスケルトンを用いて効率的に並列化する手法を提案した．本手法では，まず，効率的に並列に計算するために XML を二分木に変換する．変換された二分木において，あるノードの祖先に関する条件については dAcc スケルトンを，また，子孫に関する条件については uAcc スケルトンを用いて並列化を行った．並列スケルトンで使用する関数は，XPath を正規表現に変換して対応するオートマトンを作ることで，その状態遷移から求めることができる．並列スケルトンによって低レベルな実装が与えられるため，得られた並列プログラムは台数効果が期待できる．

また，本手法によって XPath クエリから並列スケルトンプログラムを自動的に導出するシステムを実装し，その評価実験では導出した並列プログラムは良い台数効果を示した．

今後の課題としては次のようなことが挙げられる．まず，本手法では XPath におけるいくつかの重要な

表 5: 実験に用いた XPath

名前	XPath
XPath-small	`/descendant::*[descendant::b/child::d ]`
XPath-large	`/descendant::*[descendant::b/child::d]` `/descendant::c[descendant::u/child::w]/descendant::f`

表 6: 実験に用いた XML データ

名前	ノード数	特性
normal.xml	10000	適当な幅と高さをもつようにランダムに生成されている.
mono.xml	10000	各ノードがちょうどひとつの子を持つ単調な木の形をもつ.
flat.xml	10000	各ノードの子の数が多く平坦な形のであり，木の高さは高々10 である.

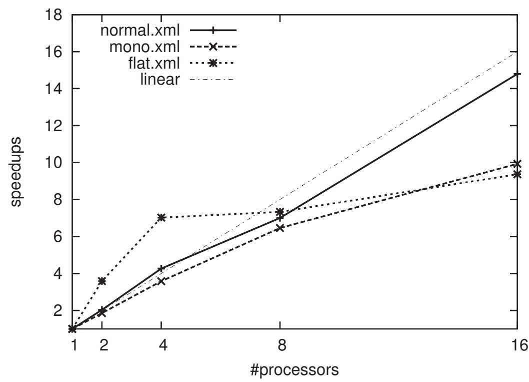


図 12: XPath-small の速度向上

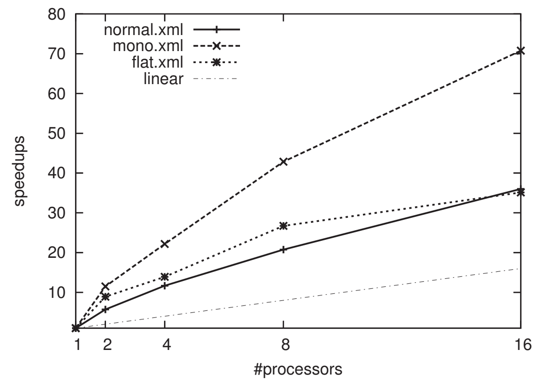


図 13: XPath-large の速度向上

表 7: XPath クエリの並列実行時間測定結果

		$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
XPath	XML	時間	比	時間	比	時間	比	時間	比	時間	比
XPath-small	normal.xml	3.88	1.00	1.91	2.03	0.91	4.26	0.55	7.00	0.26	14.79
	mono.xml	2.80	1.00	1.51	1.86	0.78	3.58	0.43	6.46	0.28	9.93
	flat.xml	4.11	1.00	1.15	3.59	0.59	7.02	0.56	7.33	0.44	9.36
XPath-large	normal.xml	14.56	1.00	2.55	5.71	1.25	11.65	0.70	20.76	0.40	35.99
	mono.xml	26.57	1.00	2.31	11.50	1.20	22.15	0.62	42.85	0.38	70.77
	flat.xml	17.07	1.00	1.92	8.91	1.23	13.90	0.64	26.72	0.49	35.11

位置指定関数を並列化したが，XPath にはその他にも多くの関数を使用することができる．これらの関数は各ノードで独立に計算を行うものであるため，比較的簡単に組み込むことができると考えられる．また，得られた並列プログラムの台数効果はスケルトンによって保証されるが，逐次計算部分などの効率化を行うことも今後の課題である．

参考文献

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon, editors. *XML Path Language (XPath) 2.0*. W3C Working Draft 29, 2004. Available from http://www.w3.org/TR/xpath20/.

[2] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon, editors. *XQuery 1.0: An XML Query Language*. W3C Working Draft 29, 2004. Available from http://www.w3.org/TR/xquery/.

[4] M. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, 1989.

[5] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space effciency. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'03)*, pages 379–390.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'02)*, pages 95–106, 2002.

[7] T. Grust. Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 109–120, 2002.

[8] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 419–430, 2003.

[9] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE'02)*, pages 215–224, 2002.

[10] M. Kay, editor. *XSL Transformations (XSLT) Version 2.0*. W3C Working Draft 5, 2004. Available from http://www.w3.org/TR/xslt20/.

[11] K. Lü, Y. Zhu, and W. Sun. Parallel processing XML documents. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS'02)*, pages 96–105, 2002.

[12] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. In *the International Workshop on "Constructive Methods for Parallel Programming" (CMPP'04)*, pages 109–123, 2004.

[13] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *the Annual European Conference on Parallel Processing (EuroPar'03), LNCS 2790*, pages 789–798. Springer-Verlag, 2003.

[14] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers, LNCS 2490*, pages 109–127. Springer-Verlag, 2002.

[15] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 431–442, 2003.

[16] SkeTo Project. Sketo project home page. http://www.ipl.t.u-tokyo.ac.jp/sketo/, 2005.

[17] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag New York Inc., 2002.

[18] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

[19] D. B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3(1):42–68, 1997.