木スケルトンによるXPathクエリの並列化とその評価

野村 芳明　江本 健斗　松崎 公紀　胡 振江　武市 正人

スケルトン並列プログラミングは，並列スケルトンと呼ばれる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手法である．本論文では木スケルトンを用いて XPath クエリ処理を実現する．XML 木はさまざまな形状を持ちうるため，XPath クエリ処理の効率的な並列実装は難しい．木スケルトンは木の形状によらない効率的な実装がなされており，本手法による XPath クエリ処理は XML データの形状によらずよい台数効果を示した．

Skeletal parallelism encourages us to develop parallel programs by composing ready-made components called parallel skeletons. In this paper, we propose a parallel implementation of XPath queries by using parallel tree skeletons. Since the parallel tree skeletons are implemented efficiently in parallel for trees of any shape, our implementation of XPath queries has demonstrated good scalability even for the ill-balanced XML trees.

1 はじめに

XML はデータ記述言語として広く利用されており，近年，大量のデータが XML によって管理されるようになってきた．しかし，巨大な XML 文書からユーザが欲しい情報を得ることは容易でない場合が多い．そこで，XML 文書中の特定要素を抽出する機構が必要になる．XPath [2] は，XML 文書の表す木構造を辿ることで XML 文書中の要素を抽出するための言語であり，他の XML 処理言語である XSLT [12] や XQuery [4] の基礎をなすものである．そのため，XPath の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 [7] [11] [18] [19].

しかし，XPath クエリ処理を並列化することによって効率の向上を図るというアプローチの研究はまだ少ない [13] [22]．その理由として，並列プログラムではプロセッサ間の複雑な処理を記述しなければならないことが挙げられる．XML は任意数の子ノードを許す木構造をなしており，このような不均等な形状をなし得るデータ構造に対して，効率的な並列計算を記述することは一般に簡単ではない．特に，XPath クエリ処理に関しては，XML 木の全てのノードを調べる必要のあるクエリが存在すること，あらゆる方向の木の辿り方を指定できること，などが効率的な並列化を難しくしている．

例えば，部分木ごとにデータを各プロセッサに割り振るという自明な分割統治法では，不均等な形状の木に対して均等なデータ分配ができず，十分な並列効果が得られない．また，

```
/descendant::a[descendant::b]
```

という XPath 式は，子孫要素のどれかに要素 `b` をもつような要素 `a` を意味しており，このような XPath 式に対しては枝刈りを用いた探索が効果的に機能しない．

効率の保証された並列プログラムを容易に得るための枠組みとして，スケルトン並列プログラミング [5] [20] がある．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パッケージ化された並列計算を基本単位として組み合わせることで，並列プログラムを構成する．プロセッサ間通信，同期処理，

Parallelization of XPath Queries with Tree Skeletons.
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.
コンピュータソフトウェア, Vol.24, No.3 (2007), pp.51–62.
[論文] 2005 年 8 月 8 日受付.

データや計算資源の配置などは並列スケルトンの中に隠蔽されているため，ユーザはプロセッサ間の処理を意識せずにプログラムを考えることができる．一方で各並列スケルトンはそれぞれ適切な実装が与えられており，並列スケルトンで記述されたプログラムは効率的に動作する．

Skillicorn [22] はスケルトン並列プログラミングの枠組みに基づき，木構造に対する並列スケルトン (木スケルトン) [16][21] を用いたクエリ処理を提案した．しかし，この手法は親子関係のみによるクエリしか扱っていない．そこで本論文では，より複雑なクエリの並列化手法を示す．

本論文の貢献は以下の通りである．

木スケルトンを用いた XPath クエリ処理の記述

本論文では，基本となる数種類の木スケルトンを用いて，木の構造に関した XPath クエリ処理を記述する方法を与える．本手法では, Skillicorn が扱っていない親子関係以外の関係を用いた XPath クエリやノードへの付加条件を含んだ XPath クエリも対象とする．

木スケルトンのパラメータの自動導出

木スケルトンには，パラメータとしていくつかの関数を与える必要があり，それらには並列計算のための条件が要求されている．本論文では，条件を満たす関数を自動導出する方法を与える．

評価実験による台数効果の確認

提案手法を実装して評価実験を行った．本手法によって並列化された XPath クエリ処理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 節では対象とする XPath を定める．第 3 節では本手法で用いる木スケルトンについて説明する．第 4 節では木スケルトンを用いて XPath クエリを処理する方法を述べる．第 5 節では提案手法を実装し，評価実験を行った結果を示す．第 6 節では関連研究について述べる．第 7 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 [2] について簡単に説明し，本論文で対象とする XPath を定める．

XML は木構造をなすため，親子関係などの木上での関係により，ノードの辿り方を用いて特定要素を指し示すことができる．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 [12] および XQuery [4] などで広く使用されている．

本論文では，W3C によって定められた XPath のうち，木の構造に関するクエリを扱う．図 1 に本論文で扱う XPath 式の構成を BNF で示す．

XPath 式は複数のステップからなる．ステップは木の辿り方を示す軸，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付加条件である述語から構成される．ノードテストにはノード名またはワイルドカードを記述する．任意のノードを意味するワイルドカードは “*”で表す．軸については，自分自身を表す `self`，子および子孫を表す `child` と `descendant`，親および祖先を表す `parent` と `ancestor`，子孫または自分自身を表す `descendant-or-self`，祖先または自分自身を表す `ancestor-or-self`，XML 文書中で自分より後に存在するノードを表す `following`，自分より前に存在するノードを表す `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 要素を逆向きに辿る軸 `parent`，`ancestor`，`ancestor-or-self`，`preceding`，`preceding-sibling` を逆方向軸と呼ぶ．

XPath を用いたクエリの例を図 3 に示す．XPath 式

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

により，ルートノードから子孫ノードの `a`，その子孫ノードの `b`，その子ノードの `c` と辿られるノードが指し示される．XPath 式がある要素を指し示している場合，その要素は XPath 式にマッチしていると言う．一般に，1 つの XPath 式にマッチする要素は複数存在し得る．本論文で扱う XPath クエリ処理は，図 3 のようにマッチするノードにマーク付けした木を返す処理とする．

述語には，限定されたロケーションパスもしくはノード集合関数の式を記述する．述語中のロケーションパスは軸とノードテストを連ねたものであり，対象となるノードから辿れるノードに条件を付ける．本論文では述語のネストは考えない．例えば，XPath 式

PathExpr	::=	*StepExpr* (*StepExpr*)∗
StepExpr	::=	*AxisStep* (“[” *PredicateExpr* “]”)?
AxisStep	::=	“/” *Axis* “::” *NodeTest*
Axis	::=	*ForwardAxis* | *ReverseAxis*
ForwardAxis	::=	“self” | “child” | “descendant” | “descendant-or-self”
		| “following” | “following-sibling”
ReverseAxis	::=	“parent” | “ancestor” | “ancestor-or-self”
		| “preceding” | “preceding-sibling”
NodeTest	::=	*String* | “∗”
PredicateExpr	::=	*LocationPath* | *RelationExpr*
LocationPath	::=	*Axis* “::” *NodeTest* (*AxisStep*)∗
RelationExpr	::=	“position()” *GeneralComp Expr*
GeneralComp	::=	“=” | “!=” | “<” | “<=” | “>” | “>=”
Expr	::=	*Num* | “last()” | “last()” “−” *Num*

図 1 本論文で扱う XPath 式

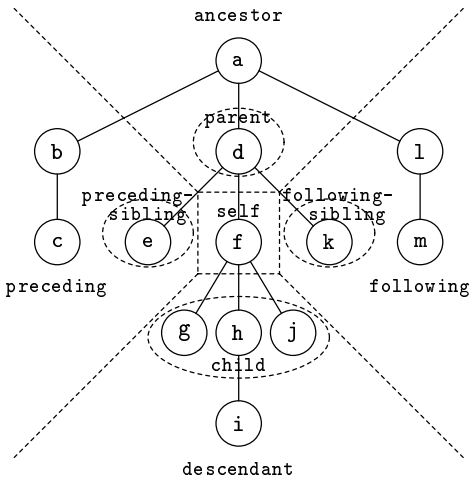

図 2 ノード f から見た各軸の表す位置関係

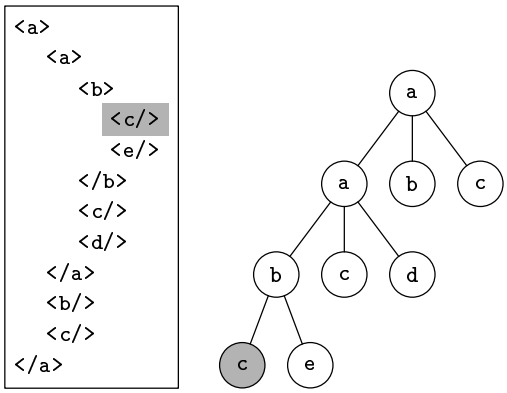

図 3 “/descendant::a/descendant::b/child::c” は影のついた要素を指定する．

`/descendant::a[following-sibling::b]`
は図 4 のようにルートノードの子孫であるノード a のなかで弟にノード b を持つものを指定する．ノード集合関数 position は，述語の直前のノードがそこまでの XPath 式にマッチするもののなかで何番目に現れるものであるかを条件に付加する．また，ノード集合関数 last は述語の直前までの XPath 式にマッチするノードの総数を表す．例えば，XPath 式

`/descendant::b[potision()=last()]`
は図 5 のようにノード b のうちで XML 文書内の最後に現れるノードを指定する．

ステップを連ねて記述された XPath 式の特徴として，XPath 式に child 軸と descendant 軸しか使われていないとき，クエリの各ステップは述語の部分とその手前の部分で，指し示す要素の子孫方向の要素に関する条件と祖先方向の要素に関する条件に分離できる．例えば，

`/descendant::a[child::b]`

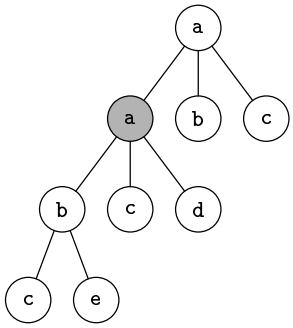

図 4 “/descendant::a[following-sibling::b]” は影のついた要素を指定する．

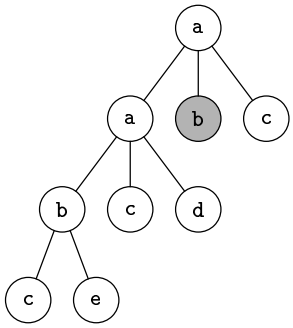

図 5 “/descendant::b[potision()=last()]” は影のついた要素を指定する．

という XPath 式では，“/descendant::a” の部分は「a はルートノードの子孫として辿られる」という祖先要素に関する条件を，“[child::b]” の部分は「a は子供に b を持つ」という子孫要素に関する条件を与えている．

3 木スケルトン

スケルトン並列プログラミング [5] は並列スケルトンと呼ばれる基本的な並列計算パターンを組み合わせて並列プログラムを構成する手法である．木を対象とする並列スケルトンを木スケルトン [16] [21] と呼ぶ．

本節では，本論文における表記法を導入し，重要な木スケルトンの定義を与える．

3.1 用語および表記法

本論文での表記法は，関数型プログラミング言語 Haskell [3] 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 $\oplus$ で表し，$(\oplus)$ 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．すなわち，$(\oplus)\ a\ b = a \oplus b$ である．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．また，関数合成は $\circ$ で表され，$(f \circ g)\ a = f\ (g\ a)$ である．

二分木

二分木は内部ノードが全てちょうど 2 つの子を持つような木である．葉ノードのデータ型が α，内部ノードのデータ型が β であるような二分木のデータ型は

$$\textbf{data}\ \mathit{BTree}\ \alpha\ \beta =$$
$$\mathit{Leaf}\ \alpha \mid \mathit{Node}\ \beta\ (\mathit{BTree}\ \alpha\ \beta)\ (\mathit{BTree}\ \alpha\ \beta)$$

と定義される．

3.2 二分木上の木スケルトン

二分木上の木スケルトン [21] 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．重要な二分木上の木スケルトンには，map，zip，reduce，uAcc (upwards accumulate)，dAcc (downwards accumulate) の 5 つがある．これらの形式的な定義を図 6 に示す．

木スケルトン map は 2 つの関数 k_L，k_N を受け取り，二分木の全ての葉ノードに関数 k_L を，全ての内部ノードに関数 k_N を適用する．木スケルトン zip は同じ形の 2 つの木を受け取り，対応するノードを組にした木を返す．木スケルトン reduce は 2 つの関数 k_L，k_N を受け取り，二分木の葉ノードに対して 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全ノードの値を縮約して 1 つの値を返す．木スケルトン uAcc は葉ノードからルートノードに向かって値を累積させていく計算を行う．この計算結果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 reduce を適用した値を割り当てた木になっている．木スケルトン dAcc はルートノードから葉ノードに向かって値を累積させていく計算を行う．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

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

図 6 二分木に対する並列スケルトンの定義

行われる．

これらの木スケルトンの実装は tree contraction アルゴリズム [1][17] によって与えられる [15]．この tree contraction アルゴリズムは木の形状によらず計算量の変わらない計算手順を与えている．

木スケルトン reduce，uAcc，dAcc は，それらが並列に計算できることを保証するために引数の関数や演算子に条件を課している．木スケルトン reduce，uAcc 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 が存在することを要求する．

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

また，木スケルトン dAcc は，二項演算子 $\oplus$ が単位元を持つ結合的な演算子であることを要求する．

各スケルトンの計算コストについて簡単に示す．木のノード数を n，プロセッサ数を p とし，渡される関数の計算時間がいずれも定数時間であるとする．このとき，map，zip は $\mathrm{O}(n/p)$，reduce，uAcc，dAcc は $\mathrm{O}(n/p + \log p)$ の時間でそれぞれ並列に計算できる．ただし，計算時間にはデータ分散のコストは含まないものとする．

4 XPath クエリの並列化

本節では，スケルトン並列プログラミングの枠組みに基づく XPath クエリ処理の並列化について示す．すなわち，前節で説明した木スケルトンによる XPath クエリ処理の記述を行う．本手法の基本的なアイデアは以下の通りである．

第 2 節で述べたように，`child` 軸と `descendant` 軸のみを用いた XPath 式のクエリは，各ノードの祖先要素に関する条件と子孫要素に関する条件に分離して考える．これらは，各ノードに対し，祖先ノードの値を用いた計算を行う dAcc，子孫ノードの値を用いた計算を行う uAcc を用いて扱う．

その他の方向に木を辿る軸に関しては，与えられた XPath 式および XML 木に前処理を施すことによって `child` 軸，`descendant` 軸のみの場合と同様に処理する．

木を辿らない `self` 軸は，直前のステップまでが指し示すノードと同じノードに条件を与えているから，1 つ前のノードテストの述語とみなして処理を行う．また，`descendant-or-self` 軸については，軸を `descendant` にした XPath 式と `self` にした XPath 式の 2 つを用意する．結果はそれぞれのクエリ処理

を行った結果の各ノードについて論理和をとったものとなる．

4.1 前処理

第 2 節で定めた XPath は，一般の XPath の軸全てを含んでいる．これらを上記アイデアで扱うための前処理について説明する．

まず，与えられた XPath 式中に逆方向軸が含まれていた場合，rare アルゴリズム [18] によって逆方向軸の除去を行う．このアルゴリズムは与えられた XPath 式に対して逆方向軸を含まない等価な XPath 式を与える．例えば，

```
/descendant::b/parent::a
```

という XPath 式は

```
/descendant-or-self::a[child::b]
```

へと変換される．

また，`following` 軸に関しては，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
```

と等価であるから，この表現に書き換えることにより除去することができる．書き換えを最初に行っておき，書き換えた結果含まれる `ancestor-or-self` 軸は rare アルゴリズムによって除去すればよい．結果として，再び `following` が含まれてしまうということはない．以下では，特に断わらない限り XPath 式は逆方向軸と following 軸を含まないものとする．

次に，親子関係を処理する木スケルトンを用いて兄弟間の関係を表すクエリを扱うために，与えられた XML 木を二分木表現 [6] へと変換する．図 7 に変換によって得られた二分木の例を示す．変換された二分木では，元の木における親と最初の子の関係は親と左の子の関係に，元の木における左右の兄弟関係は親と右の子の関係に, それぞれ対応する．また，二分木表現における各内部ノードには, 親ノードに対して左の子であるか右の子であるかを表すラベル L，Rを付加する．これは元の木において長男ノードなのか弟ノードなのかをノード名だけから識別可能にするためである．元の木のノードは全て内部ノードになり，葉としてダミーノードを挿入する．元の木のノード数を n とすると，変換後の木のノード数は $2n+1$ になる．変換によって木の形状が大きく変化するが，木スケルトンの計算コストは木の形状によらないため，計算量への影響はない．

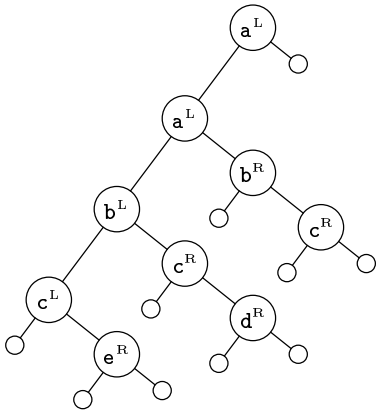

図 7　図 3 の XML 木の二分木表現

4.2 述語を含まない XPath クエリ処理の並列化

最初に簡単な XPath クエリとして述語を含まないものを考え，その並列化を示す．説明には例として

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

という XPath 式を用いる．

XPath 式が述語を含まないとき，各ノードがクエリにマッチするかどうかは，ルートノードから各ノードまでトップダウンに辿ったときのノード列によって判定できる．そこで，マッチするノードまでのノード列を受理するようなオートマトンを用意し，その遷移状態を dAcc によって計算することを考える．

オートマトンには，XPath 式に対応する正規表現から得られる非決定性有限状態オートマトンを用いる．各軸に対応する正規表現は表 1 のようになる．すなわち，`child` に対応する正規表現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` に対応する正規表現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` に対応する正規表現は弟方向に何回か辿って得られるノード，をそれぞれ表現して

表 1 二分木表現における各軸に対応する正規表現 (“•” は任意の値)

child::a	$(\bullet^{\mathrm{L}}\ \bullet^{\mathrm{R}}{*}\ \mathtt{a}^{\mathrm{R}}) \mid \mathtt{a}^{\mathrm{L}}$
descendant::b	$(\bullet^{\mathrm{L}}\ \bullet^{\bullet}{*}\ \mathtt{b}^{\bullet}) \mid \mathtt{b}^{\mathrm{L}}$
following-sibling::c	$\bullet^{\mathrm{R}}{*}\ \mathtt{c}^{\mathrm{R}}$

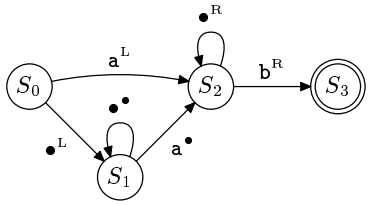

図 8 “/descendant::a/following-sibling::b” に対応するオートマトン (初期状態 S_0，受理状態 S_3)

いる．

表 1 から，例に挙げた XPath 式に対応する正規表現は “$((\bullet^{\mathrm{L}}\ \bullet^{\bullet}{*}\ \mathtt{a}^{\bullet}) \mid \mathtt{a}^{\mathrm{L}})\ \bullet^{\mathrm{R}}{*}\ \mathtt{b}^{\mathrm{R}}$” となる．なお，“•” は任意の値を意味する．この正規表現を変換して得られる非決定性有限状態オートマトンは図 8 のようになる．

次に，各ノードに対してルートノードからのノード列に対する遷移を計算する．この計算は dAcc を用いて

$$\mathsf{dAcc}\ (\oplus)\ \mathit{maketable}\ \mathit{maketable}\ \mathit{idtable}$$

と表される．ここで関数 *maketable* は，ノードの値による遷移の遷移前状態と遷移先状態との組の集合を返す関数である．例えば，図 8 のオートマトンにおいて，入力 $\mathtt{b}^{\mathrm{L}}$ に対して起こる遷移は，$S_0 \to S_1$，$S_1 \to S_1$ であるから，

$$\mathit{maketable}\ \mathtt{b}^{\mathrm{L}} = \{(S_0, S_1), (S_1, S_1)\}$$

となる．各ノードの値に対する *maketable* の値は表 2 のようになる．また，演算子 $\oplus$ は，2 つの遷移を結合することでパスを辿った際の遷移を計算する演算子であり，

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

と定義される．初期値 *idtable*は $\forall i\ S_i \to S_i$ となる遷移集合である．木スケルトン dAcc は演算子の結合性を要求するため，計算では，初期状態からの遷移先だけではなく，各状態からの遷移先を扱っている．演算子 $\oplus$ は結合性を持ち，単位元は *idtable*であるから，この dAcc は並列に計算することができる．

表 2 *maketable* の値 (#は a，b 以外の値)

$\mathtt{a}^{\mathrm{L}}$	$\{(S_0, S_1), (S_0, S_2), (S_1, S_1), (S_1, S_2)\}$
$\mathtt{a}^{\mathrm{R}}$	$\{(S_1, S_1), (S_1, S_2), (S_2, S_2)\}$
$\mathtt{b}^{\mathrm{L}}$	$\{(S_0, S_1), (S_1, S_1)\}$
$\mathtt{b}^{\mathrm{R}}$	$\{(S_1, S_1), (S_2, S_2), (S_2, S_3)\}$
$\#^{\mathrm{L}}$	$\{(S_0, S_1), (S_1, S_1)\}$
$\#^{\mathrm{R}}$	$\{(S_1, S_1), (S_2, S_2)\}$

最後に，zip スケルトンを用いて dAcc の計算結果を元の木に付加し，ルートノードから親ノードまでの遷移とノードの値から，各ノードの遷移先状態を計算する．このとき，初期状態からの遷移先に受理状態を含むノードがクエリにマッチするノードである．マッチするノードのマーク付けを行う計算は map スケルトンを用いて

$$\begin{aligned} &\mathsf{map}\ \mathit{isMatch}_d\ \mathit{isMatch}_d \\ &\quad \textbf{where} \\ &\quad \mathit{isMatch}_d\ (t, n) = \\ &\qquad \textbf{if}\ (S_0, S_3) \in (t \oplus \mathit{maketable}\ n)\ \textbf{then}\ 1 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ 0 \end{aligned}$$

と表せる．マッチするノードは 1 を，マッチしないノードは 0 を，結果としている．

以上から，XPath 式に対応したオートマトンから必要な関数を生成し，dAcc，zip，map を用いた計算を行うことで，述語を含まない XPath クエリを処理することができる．

4.3 述語を含む XPath クエリ処理の並列化

次に，述語を含んだ XPath クエリ処理の並列化を示す．ここでは例として

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

という XPath 式を用いる．

XPath 式が述語を含む場合，まず，述語部分の条件を満たすノードのマーク付けを行う．例の場合，XPath 式中の

```
a[descendant::b],
```

```
c[following-sibling::d]
```

の部分の条件を満たすノード，すなわち，子孫にノード b を持つようなノード a，弟にノード d を持つようなノード c をそれぞれマーク付けする処理を行う．述語部分の条件を満たすノードのマーク付けを行った後に，結果を zip スケルトンを用いて元の木に付加することで，述語の条件をノード名の条件と同じように扱うことができる．すなわち，述語の条件を満たしている a を a’，述語の条件を満たしている c を c"とし，

```
/descendant::a'/child::c"
```

を前節の方法で処理する．本節では，抜き出された述語部分に対するクエリ処理について説明する．

述語条件がロケーションパスで与えられている場合，前節と同様，対応するオートマトンの遷移を考えることでクエリ処理を行うことができる．述語中のロケーションパスは，指定するノードに対しその子孫ノードに関する条件を与える．従って，各ノード以下の部分木に対して計算を行う uAcc を使用して述語部分のクエリ処理を行う．

例に用いた XPath 式の 1 つ目の述語部分，

```
a[descendant::b]
```

の場合，述語中のロケーションパスに対応する正規表現 “($\bullet^{\mathrm{L}}$ $\bullet^{\bullet}$* b$^{\bullet}$) | b$^{\mathrm{L}}$” の先頭に述語条件の対象となるノード名を加えた “a$^{\bullet}$ (($\bullet^{\mathrm{L}}$ $\bullet^{\bullet}$* b$^{\bullet}$) | b$^{\mathrm{L}}$)” を変換して得られるオートマトンが述語部分に対応するオートマトンである．このとき，各ノードを始点として子孫を辿った遷移先に受理状態を含むノードが述語の条件を満たしたノードとなる．従って，各ノードにおいて子孫を辿って遷移し得る状態集合が分かればよい．

あるノードからの遷移は，そのノードから左の子へ辿った場合，右の子へ辿った場合，およびそのノードで辿るのをやめる場合，の 3 通りの和集合をとることによって求められる．各ノードからの遷移を求め，述語を満たすノードのマーク付けを行う計算は

$$
\begin{aligned}
&\mathsf{map}\ \mathit{isMatch}_u\ \mathit{isMatch}_u \circ \mathsf{uAcc}\ k_L\ k_N\\
&\quad \textbf{where}\ k_L\ n = \mathit{idtable}\\
&\qquad k_N\ n\ l\ r =\\
&\qquad\quad (\mathit{maketable}\ n \oplus (l \cup r)) \cup \mathit{idtable}\\
&\qquad \mathit{isMatch}_u\ t =\\
&\qquad\quad \textbf{if}\ (S_0, S_3) \in t\ \textbf{then}\ 1\ \textbf{else}\ 0
\end{aligned}
$$

となる．関数 $maketable$，$idtable$および演算子 $\oplus$ は 4.2 節と同じ定義である．上記の関数 k_N に対して uAcc の並列計算に必要な関数 ϕ，ψ_L，ψ_R，G は以下のように与えることができる．

$$
\begin{aligned}
&\phi\ n = (\mathit{maketable}\ n, \mathit{idtable})\\
&\psi_L\ (n_1, n_2)\ l\ (rn_1, rn_2) =\\
&\quad (n_1 \oplus rn_1, (n_1 \oplus (l \cup rn_2)) \cup n_2)\\
&\psi_R\ (n_1, n_2)\ r\ (ln_1, ln_2) =\\
&\quad (n_1 \oplus ln_1, (n_1 \oplus (ln_2 \cup r)) \cup n_2)\\
&G\ (n_1, n_2)\ l\ r = (n_1 \oplus (l \cup r)) \cup n_2
\end{aligned}
$$

また，本手法では self 軸のステップも直前のステップの述語とみなして処理を行う．例えば，

```
/descendant::*[child::e]
  /self::f[descendant::g]
```

という XPath 式では，最初のステップには,「子供に e を持つ」,「子孫に g を持つ f」という 2 つの述語が与えられていると考える．従って，

```
*[child::e],
f[descendant::g]
```

の両方に対してマーク付けがなされたノードを f’とし，

```
/descendat::f'
```

を処理することになる．なお，本論文ではネストした述語は扱わないため，self 軸が述語内に現れる場合も扱っていない．

実際の計算では，述語部分の数だけ uAcc を繰り返すのではなく，各述語部分の計算に用いる値を全て組にしたものを計算に用いることで，1 回の uAcc でまとめて計算を行う．従って，ネストした述語を持たない XPath 式は全て，uAcc と dAcc とを 1 回ずつ適用することでクエリ処理を行うことができる．

4.4 ノード集合関数を含む XPath クエリ処理の並列化

最後に，ノード集合関数 position，last の処理について述べる．例として

```
/child::a[position()=last()]/child::b
```

という XPath 式を用いる．

これらはまず，position，last を含んだ述語までのロケーションパス部分について先にクエリ処理を行

い，position 関数で数える対象となるノードをマークする．そのうえで，position に対してはマークされたノードに position 関数の結果を格納した木を返す処理を，last に対してはマークされたノードの総数を返す処理を，それぞれ行う．

例の場合，

```
/child::a
```

の部分の処理を行った後，position，last の計算を行う．得られた position の値を zip スケルトンで元の木に付加し，map スケルトンを用いることで，

```
/child::a[position()=last()]
```

にマッチするノード a' が得られる．そのうえで，

```
/child::a'/child::b
```

の処理を行えばよい．

関数 position の計算は，dAcc，uAcc を用いて，

$$\begin{aligned}&\mathsf{dAcc}\ (+)\ \pi_1\ \pi_2\ 1 \circ \mathsf{uAcc}\ k_L\ k_N\\ &\quad \textbf{where}\\ &\quad k_L\ n = (0,0,0)\\ &\quad k_N\ n\ l\ r = (n, n+\pi_3\ l, n+\pi_3\ l+\pi_3\ r)\end{aligned}$$

と表せる．最初の uAcc は，各ノードから左に辿ったときと，右に辿ったときで，position の値がいくつ増えるかの計算を行う．対象とする木は 0 と 1 でマーク付けされているものとする．計算には補助として，各ノード以下のマーク付けされたノードの総数も組にした 3 つ組を用いている．関数 π_i は，3 つ組の i 番目の値を返す関数である．関数 k_N に対する並列計算用の関数は以下のように与えることができる．

$$\begin{aligned}&\phi\ n = (n, 1, 0, 1, 0, 0, 0)\\ &\psi_L\ (v,a,b,c,d,e,f)\ l\ (rv,ra,rb,rc,rd,re,rf) =\\ &\quad (rv, 0, a\times v+b, 0, d, d\times rf + c\times(\pi_3\ l + v)\\ &\qquad + d\times(\pi_3\ l + v) + e, rf + \pi_3\ l + v + f)\\ &\psi_R\ (v,a,b,c,d,e,f)\ r\ (lv,la,lb,lc,ld,le,lf) =\\ &\quad (lv, 0, a\times v+b, 0, c+d, (c+d)\times lf + c\times v\\ &\qquad + d\times(\pi_3\ r + v) + e, lf + \pi_3\ r + v + f)\\ &G\ (v,a,b,c,d,e,f)\ l\ r =\\ &\quad (a\times v+b, c\times(v+\pi_3\ l) + d\times(v+\pi_3\ l\\ &\qquad + \pi_3\ r) + e, v + \pi_3\ l + \pi_3\ r + f)\end{aligned}$$

そして，次の dAcc で，uAcc によって計算された値を用い，各ノードの position の値を求めている．計算の結果，マーク付けされていないノードにも値が格納されるが，これらは意味をなさないものとする．

また，関数 last の計算は，reduce を用いて，

$$\begin{aligned}&\mathsf{reduce}\ k_L\ k_N\\ &\quad \textbf{where}\ k_L\ n = 0\\ &\qquad k_N\ n\ l\ r = n + l + r\end{aligned}$$

と表せる．この計算は，position の計算の uAcc で用いる 3 つ組の 3 番目の値の計算に相当する．

5 評価実験

本節では，前節で示した XPath クエリ処理の手法を実装し，評価実験によって並列化の効果を確認する．

5.1 実装

実装したプログラムは入力として XML 木および XPath 式を受け取る．ただし，前節で説明した前処理は既に施されているものとする．すなわち，XML 木として二分木表現へ変換されたものを受け取り，XPath 式として逆方向軸を含まないものを受け取る．

プログラムは以下のような動作を行う．まず最初に，与えられた XPath 式から対応するオートマトンを生成する．正規表現からオートマトンへの変換は，XPath 式に対応する正規表現のみを対象とするため，各軸に対応するオートマトンを直接与えることで行う．次に，オートマトンから得られた関数を引数として木スケルトンを呼び出し，クエリ対象の XML 木の二分木表現に対して並列に計算を行う．なお，木スケルトンの実装には並列スケルトンライブラリ「助っ人」 [14] を用いた．

5.2 評価実験とその結果

実験の並列計算機環境には 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 Intel® Xeon® 2.80GHz の CPU，2GB のメモリを持ち，それぞれギガビットイーサネットで接続されている．OS は Linux 2.4.21，コンパイラは gcc 2.96，MPI ライブラリは mpich 1.2.7 を用いた．

実験では，表 3 に示す XML データに対して表 4 の XPath 式のクエリ処理を並列実行し，その実行時間を測定した．ただし，データの分散にかかる時間や結果の収集にかかる時間は実行時間に含めていない．

表 3　実験に用いた XML データ

名前	ノード数	特性
random.xml	100000	ランダムに選んだノードに子を挿入していくことによって生成されている.
mono.xml	100000	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	100000	各ノードの子の数が多く平坦な形であり，木の高さが高々10 である.

表 4　実験に用いた XPath 式

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 5　XPath クエリ処理の並列実行時間測定結果 (P はプロセッサ数，時間の単位は秒)

		$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
XPath 式	XML 木	時間	比	時間	比	時間	比	時間	比	時間	比
XPath-small	random.xml	0.74	1.00	0.41	1.80	0.22	3.34	0.10	7.59	0.05	13.82
	mono.xml	1.29	1.00	0.68	1.90	0.37	3.44	0.19	6.66	0.13	9.86
	flat.xml	0.75	1.00	0.48	1.55	0.25	2.93	0.13	5.55	0.07	10.72
XPath-large	random.xml	2.05	1.00	1.07	1.90	0.56	3.65	0.27	7.64	0.14	14.32
	mono.xml	3.29	1.00	1.72	1.92	0.93	3.55	0.48	6.79	0.33	10.06
	flat.xml	2.13	1.00	1.15	1.85	0.61	3.52	0.31	6.98	0.16	13.26

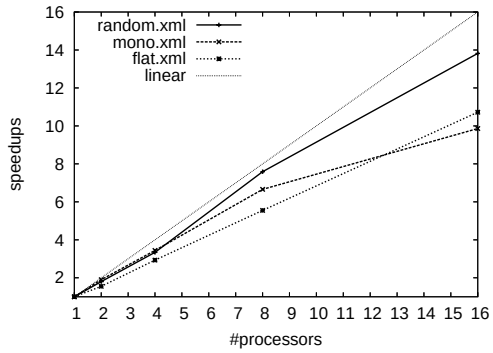

図 9　XPath-small に対する処理の速度向上

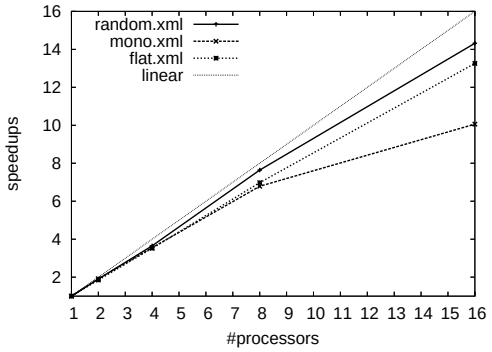

図 10　XPath-large に対する処理の速度向上

実験に用いる XML 木は，ランダムに選んだノードに子ノードを挿入していくことによって生成された “random.xml”，極端に高い XML 木として各ノードがちょうどひとつの子を持つように生成された “mono.xml”，極端に幅の広い XML 木として木の高さが高々10 であるように生成された “flat.xml” という 3 種類のデータである．また，各々のノード数は 100000 である．

測定された実験結果を表 5 に示す．表中の比は 1 台のプロセッサでの並列プログラム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 9，図 10

に示す．図 9，図 10 において，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．

ランダム生成された “random.xml” に関しては，使用する台数にほぼ比例する形で速度向上が得られ，並列化による台数効果が確認できた．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た．いずれの結果も台数効果がはっきりと現れており，提案手法による並列化が有効であることが示された．

6 関連研究

はじめに述べたように，XPath クエリ処理の並列化に関する研究はまだあまりない．我々以外のアプローチでは Lü ら [13] がリレーショナルデータベース上で XPath クエリ処理を実現する手法を提案している．また，本手法の基となった Skillicorn の手法 [22] は，スケルトン並列プログラミング [5][20] の枠組みに基づき，親子関係によるクエリを dAcc スケルトンを用いて実現した．

Lü らの手法では，部分木の子の数によりデータ分散を行うなど，アドホックで複雑な並列計算を行っている．一方，本手法の並列計算は木スケルトン [16][21] に隠蔽されており，クエリ処理とは分離されている．本手法で用いた木スケルトンは，木の形状によらず効率の保証された並列計算を行う tree contraction アルゴリズム [1][17] に基づいて実装されている．

Skillicorn の手法では，親子関係のみの簡単なクエリしか扱われていない．これに対し我々は，全ての軸および述語を用いた XPath 式を扱う方法を与えた．逆方向軸を扱うための前処理には，Olteanu らの rare アルゴリズム [18] を用いた．これは，与えられた XPath 式に対し，変換規則の適用を収束するまで繰り返すことによって，逆方向軸を含まない等価な XPath 式を与えるアルゴリズムである．また，Skillicorn の手法では決定性オートマトンが用いられていたが，uAcc スケルトンによる述語処理の際には子孫ノードの辿り方に対して非決定的に状態遷移を扱う必要があることから，本手法では非決定性のオートマトンを用いている．非決定性オートマトンの使用は，状態数が指数関数的に増加する決定性オートマトンへの変換を避けることで，効率の改善にもつながっている．

7 まとめと今後の課題

本論文では，木の構造に関する XPath クエリ処理を木スケルトンによって実現する新しい手法を提案した．

本手法では，まず，前処理として rare アルゴリズムによって XPath 式から逆方向軸を除去し，XML 木を二分木表現へと変換する．続いて，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，子孫に関する条件については uAcc スケルトンを用いて計算を行う．木スケルトンで使用する関数は，XPath 式に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．木スケルトンによって効率の保証された低レベルな実装が与えられるため，実装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まず，扱う XPath の拡大が挙げられる．本論文では木の構造に関するクエリに限定し，ロケーションパスとノード集合関数だけを扱った．しかし，XPath のその他の表現のうち，各ノードで独立に計算を行うものに関しては，map スケルトンを用いて比較的簡単に組み込むことができると考えている．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，逐次の動作のさらなる効率化を行うことも重要な今後の課題である．

参 考 文 献

[1] Abrahamson, K., Dadoun, N., Kirkpatrik, D. G. and Przytycka, T.: A Simple Parallel Tree Contraction Algorithm, *Journal of Algorithms*, Vol. 10, No. 2 (1989), pp. 287–302.

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

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

[4] Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J. and Simeon, J. (eds.): *XQuery 1.0: An XML Query Language*, W3C

Working Draft 29, 2004. Available from `http://www.w3.org/TR/xquery/`.

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

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: *Introduction to Algorithms*, MIT Press, second edition, 2001.

[7] Gottlob, G., Koch, C. and Pichler, R.: Efficient Algorithms for Processing XPath Queries, in *Proceedings of the International Conference on Very Large Data Bases* (*VLDB '02*), 2002, pp. 95–106.

[8] Gottlob, G., Koch, C. and Pichler, R.: XPath Query Evaluation: Improving Time and Space Effciency, in *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE '03*), 2003, pp. 379–390.

[9] Grust, T.: Accelerating XPath Location Steps, in *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD '02*), 2002, pp. 109–120.

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

[11] Helmer, S., Kanne, C.-C. and Moerkotte, G.: 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*), 2002, pp. 215–224.

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

[13] Lü, K., Zhu, Y. and Sun, W.: Parallel Processing XML Documents, in *Proceedings of the International Database Engineering & Applications Symposium* (*IDEAS '02*), 2002, pp. 96–105.

[14] Matsuzaki, K., Emoto, K., Iwasaki, H. and Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming, in *Proceedings of the First International Conference on Scalable Information Systems* (*InfoScale '06*), 2006.

[15] Matsuzaki, K., Hu, Z. and Takeichi, M.: Implementation of parallel tree skeletons on distributed systems, in *Proceedings of the Third Asian Workshop on Programming Languages And Systems*, 2002, pp. 258–271.

[16] Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallelization with Tree Skeletons, in *Proceedings of the Annual European Conference on Parallel Processing* (*EuroPar '03*), LNCS 2790, Springer-Verlag, 2003, pp. 789–798.

[17] Miller, G. L. and Reif, J. H.: Parallel Tree Contraction and its Application, in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 1985, pp. 478–489.

[18] Olteanu, D., Meuss, H., Furche, T. and Bry, F.: XPath: Looking Forward, in *Proceedings of the XML-Based Data Management and Multimedia Engineering* (*EDBT '02*), LNCS 2490, Springer-Verlag, 2002, pp. 109–127.

[19] Peng, F. and Chawathe, S. S.: XPath Queries on Streaming Data, in *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD '03*), 2003, pp. 431–442.

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

[21] Skillicorn, D. B.: Parallel Implementation of Tree Skeletons, *Journal of Parallel and Distributed Computing*, Vol. 39, No. 2 (1996), pp. 115–125.

[22] Skillicorn, D. B.: Structured Parallel Computation in Structured Documents, *Journal of Universal Computer Science*, Vol. 3, No. 1 (1997), pp. 42–68.