DEIM Forum 2016 D5-4

木構造データに対する類似部分木検索の並列化

小柳　涼介†　　天笠　俊之††　　北川　博之††

† 筑波大学システム情報工学研究科コンピュータサイエンス専攻　〒305–8573 茨城県つくば市天王台 1 丁目 1-1
†† 筑波大学システム情報系情報工学域　〒305–8573 茨城県つくば市天王台 1 丁目 1-1
E-mail: †rkoya@kde.cs.tsukuba.ac.jp, ††amagasa@cs.tsukuba.ac.jp, †††kitagawa@cs.tsukuba.co.jp

あらまし　木構造データに対する類似部分木検索問題として，与えられたクエリ木との木編集距離が T 以下であるという条件を満たすデータベース木の部分木を列挙する問題を考える．類似部分木検索問題は，XML を始めとする半構造データに対する類似検索の一手法として重要な問題である．近年に見られるビッグデータ化の傾向とともに，XML を始めとする木構造データのデータサイズも拡大の一途を辿っており，木編集距離に基づく類似部分木検索問題は多くの先行研究により高速化手法が考案されている．先行研究はどれも枝刈りをベースとした高速化であったが，本研究では既存の手法を効率的に並列処理化する手法を提案する．本手法は，(1) 与えられたデータベース木を効率良く複数ノードに割り当てるための分割方法と，(2) 複数ノード間にまたがる部分木への計算方法の二部から構成される．(1) ではノードをまたぐネットワークの通信量とノード間の計算対象の偏りを考慮した一般的な分割方法を考える．(2) では適切に木を順序付けることにより，ノード間の依存関係と処理順序から発生するオーバヘッドを抑える方法について考える．最後に，実験によって並列化効率を評価し，本手法が充分なスケーラビリティと拡張性を持つことを確認する．

キーワード　XML 類似度, 木編集距離, 類似検索, 並列処理

1. はじめに

近年，木構造データは XML や JSON を始めとして，Web において広く使われている．また，それら木構造データの広がりに伴い，効率よい処理方法に関する需要も高まっている．中でも，木構造データに対する類似検索は，単純なデータ検索に加え重複検出，コピー検出，データの統合など様々な目的に応用できる重要な問題である [1].

木構造データに対する類似検索において，二つの木の類似度を表す尺度は既存の多くの研究により様々なものが提案されてきた．その中の一つに木編集距離 [2] がある．木編集距離は，二つの木を同型にするために必要な操作（頂点の挿入・削除・値の変更）のコストの和として定義される．適用することができる操作の種類によって様々なバリエーションが存在し，求めるためにかかる計算量も多様である [3–8].

木構造データの類似検索に関する問題の中でも特に，与えられた木構造データ（クエリ木）に対して，予め用意しておいた木構造データ（データベース木）に含まれるすべての類似した部分木を列挙することを類似部分木検索と呼ぶ．類似部分木検索問題を対象とした研究として Augsten らの提案する TASM [9] や Cohen らの提案する Structure Search [10] や我々の先行研究である [11,12] がある．[9,10] はどちらも類似度として木編集距離を使用しており，計算手法に木編集距離の計算回数を低減する工夫を取り入れることで高速化を達成している．

TASM では，Top-k 類似部分木検索（類似部分木の上位 k 個を列挙）を対象とし，クエリ木の頂点数に基づき探索対象の部分木の頂点数に上限を設けることで処理の効率化を実現している．

Structure Search では，データベース木が持つ部分木全ての構造を調べ，予め同一の構造を持つ部分木を特定し，インデックス付けしておくことでクエリ処理時に計算の再利用を行うことで，さらなる高速化を実現した．

我々の先行研究である [11] では，類似尺度として木編集距離とテキストの類似度の 2 種類の尺度を採用し，それらの比重を考慮した類似度モデルについて提案した．また，テキスト類似度の計算にかかるコストが木編集距離計算に比べ軽いことを利用し，テキスト類似度による枝刈りを取り入れることで高速化を成し遂げた．

同じく我々の先行研究である [12] では，[11] において構造類似度の比重が大きい場合にテキスト類似度による枝刈りが適用できず，効率が落ちてしまう問題に対し，構造の局所性を用いて情報の再利用を行うことで木編集距離計算回数を削減し，高速化を成し遂げた．

本研究では，並列処理環境下での類似部分木検索について考える．また，本研究における類似部分木検索とは，クエリ木との木編集距離が θ 以下であるようなデータベース部分木を全て列挙する問題と定義する．計算効率のためにノード間通信の量とノード間の木編集距離計算量のバランスの二種類をコストとして考慮し，これらを環境に応じてどちらをより抑えるべきかを決定できるような類似部分木検索手法を提案する．そして最後に，実際に手法の実装を行い，巨大な XML ファイルに対して本手法を適用し並列化の性能を確認する．

本論文の構成は以下のとおりである．第 2. 節で本研究の前提知識となる諸定義について述べる．第 3. 節で本研究の提案手法

について述べる．本手法における前提知識である木編集距離計算と， TASM を基にしたシングルプロセスアルゴリズムを説明し，次いでデータベース木の分割について論じる．その後，それらに基づくマルチプロセスアルゴリズムについて説明する．第 4. 節では評価実験について述べ，本手法の効率性の確認を行う．最後に，第 5. 節で本稿のまとめと今後の課題を述べる．

2. 諸 定 義

2.1 データ木

本研究ではデータベース木を，頂点が値を持つ根付き順序木 $D = (V(D), E(D), L(D), \lambda, r)$ として扱う．$V(D)$, $E(D)$, $L(D)$ は，それぞれ木に含まれる頂点，辺，値の集合である．$\lambda : V(D) \rightarrow L(D)$ は各頂点とラベルの対応を表す．任意のアルファベット列から成るラベル集合を Σ とした時，全ての $l \in L(D)$ は $l \in \Sigma$ を満たす．また，$r \in V(D)$ を 木 D の根にあたる頂点とする．

木 D に関して，D に含まれる頂点数を $|D|$ で表す．また，木 D に含まれる頂点 $V(D)$ には後行順に基づく全順序を定義する．後行順において i 番目に出現する頂点を d_i とし，頂点 d_i の親に当たる頂点の後行順番号を $par(d_i)$，頂点 d_i を根とする部分木を D_i とする．部分木 D_i に含まれる頂点において後行順番号が最も小さい頂点の後行順番号を $l(d_i)$ と定義する．任意の部分木において $|D_i| = i - l(d_i) + 1$ が成り立ち，また d_i が葉である時 $i = l(d_i)$ が成り立つ．また，この定義において 部分木 D_i について $V(D_i) = \{d_j \mid l(d_i) \leqq j \leqq i\}$ が成り立つ．すなわち，任意の部分木は後行順において連続する頂点集合から構成される．

最後に，クエリ木 を Q とし，D と同様に各種記号を定義する．

2.2 木編集距離

二つの木 D, Q について，木編集距離とは，二つの木を同型にするために必要な操作のコストの総和の最小値で定義される．本研究では,「頂点の削除」,「頂点の挿入」,「頂点の値の変更」の 3 つの基本的操作を許可する．

$delete(v, l_1)$ は 木 D から $\lambda(v) = l_1$ である頂点 v の削除を表す．この時，v が親頂点を持っていた場合， v の全ての子はその親頂点の子となる．そうでない場合，つまり v が根にあたる場合， v の全ての子はそれぞれが非連結になり、 木 D は森となる．なお，この森についても各頂点の順序は分割される前の順序が保たれる．$insert(v, l_1)$ は 木 D への $\lambda(v) = l_1$ である頂点 v の挿入を表す．削除時と対照をなすように，$insert(v, l_1)$ にも 2 つの場合が存在する．v が 頂点 u の子として挿入される場合，u の子となる順序付き頂点集合について，いずれかの連続する部分頂点列（空にもなり得る）は v の子として移動される．そうでない場合，つまり v が新しく根として挿入される場合， 森（木） D に含まれる任意の連続する根頂点が v の子として移動される．$rename(v, l_1, l_2)$ は 木 D に含まれる $\lambda(v) = l_1$ である頂点に対する， $\lambda(v) = l_2$ となるよう値の書き換えを表す．以上の全ての操作において，操作の前後で任意の頂点間の順序が前後することはない．

次に，二つのラベル $l_1, l_2 \in (\Sigma_\epsilon = \Sigma \cup \{\epsilon\})$ についてそのコスト関数を $\gamma(l_1, l_2)$ と定義する．3 つの操作はこのコスト関数に紐付けられる．$delete(v, l_1)$ のコストは $\gamma(l_1, \epsilon)$ で表され，$insert(v, l_2)$ のコストは $\gamma(\epsilon, l_2)$ で表され，$rename(v, l_1, l_2)$ のコストは $\gamma(l_1, l_2)$ で表される．

コスト関数 γ は以下の性質を満たす距離関数でなくてはならない:

- $\gamma(l_1, l_2) \geqq 0$ with equality if $l_1 = l_2$,
- $\gamma(l_1, l_2) = \gamma(l_2, l_1)$
- $\gamma(l_1, l_3) \leqq \gamma(l_1, l_2) + \gamma(l_2, l_3)$.

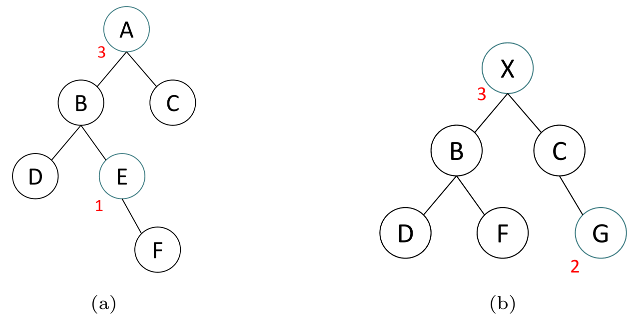

図 1: 木編集距離の計算例

例として図 1 の木 (a), 木 (b) 間の木編集距離を考える．また，木への操作として頂点の追加，頂点の削除，頂点の値の変更の 3 種類の操作に対し，それぞれ 1 のコストとする．木 (a) を 木 (b) に変形させるためには

（1） 頂点 E を削除
（2） 頂点 G を追加
（3） 頂点 A を頂点 X に改名

の 3 つの操作が必要となる．これらの操作のコストは 3 であり，コストが 3 未満になるような操作は存在しないため，木 (a), 木 (b) 間の木編集距離は 3 となる．

木編集距離の計算手法については様々な研究が存在するが，Zhang らの提案している手法 [4] は動的計画法を用いて 時間計算量 $O(n^2m^2)$ ，空間計算量 $O(nm)$ （ここでそれぞれ n, m は比較対象となるそれぞれの木の頂点数）を達成している．本研究と同じく木編集距離に基づく類似部分木検索を対象としている TASM, StructureSearch もこの手法により木編集距離計算を行っている．

[4] に基づく木編集距離計算アルゴリズムを Algorithm1 に示す．このアルゴリズムはデータベース部分木 D_I とクエリ部分木 Q_J が与えられた時に，それぞれの部分木に含まれる**兄を持たない頂点**同士の木編集距離を計算する．なお，このアルゴリズムが実行される際には，それぞれの部分木に含まれる**兄を持つ頂点**同士の木編集距離は既に求まっている必要がある．

3. 提 案 手 法

3.1 シングルプロセスアルゴリズム

シングルプロセスでの類似部分木検索アルゴリズムを Algorithm 2 に示す．図 2 のデータベース木と図 3 のクエリ木に

Algorithm 1 *TEDCalculation*

Require: $D_I, Q_J, \{d_i.TED[j] \mid left(d_i) \neq left(d_I)$ and $left(q_j) \neq left(q_J)\}$
Ensure: $\{d_i.TED[j] \mid left(d_i) = left(d_I)$ and $left(q_j) = left(q_J)\}$

1: $ForestDist[left(d_I)-1][left(q_J)-1] = 0$
2: **for** i : $left(d_I)$ to I **do**
3: $\quad ForestDist[i][left(q_J)-1] = ForestDist[i-1][left(q_J)-1] + \gamma(d_i, \wedge)$
4: **end for**
5: **for** j : $left(q_J)$ to J **do**
6: $\quad ForestDist[left(d_I)-1][j] = ForestDist[left(d_I)-1][j-1] + \gamma(\wedge, q_j)$
7: **end for**
8: **for** i : $left(d_I)$ to I **do**
9: $\quad$ **for** j : $left(q_J)$ to J **do**
10: $\quad\quad$ **if** $left(d_i) = left(d_I)$ and $left(q_j) = left(q_J)$ **then**
11: $\quad\quad\quad d_i.TED[j] = ForestDist[i][j] = min($
$\quad\quad\quad\quad ForestDist[i-1][j] + \gamma(d_i, \wedge),$
$\quad\quad\quad\quad ForestDist[i][j-1] + \gamma(\wedge, q_j),$
$\quad\quad\quad\quad ForestDist[i-1][j-1] + \gamma(d_i, q_j)$
$\quad\quad\quad)$
12: $\quad\quad$ **else**
13: $\quad\quad\quad ForestDist[i][j] = min($
$\quad\quad\quad\quad ForestDist[i-1][j] + \gamma(d_i, \wedge),$
$\quad\quad\quad\quad ForestDist[i][j-1] + \gamma(\wedge, q_j),$
$\quad\quad\quad\quad ForestDist[left(d_i)-1][left(q_j)-1] + d_i.TED[j]$
$\quad\quad\quad)$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end for**

対して $\theta = 3$ でこのアルゴリズムを適用した場合について考える．なお，図における各頂点の左の数字は後行順番号を表し，右の文字は値を表す．1 行目の $max|D|$ は計算対象となる部分木のサイズの上限を表す．例では，$|Q| = 6, \theta = 3$ のため，$max|D| = 9$ となる．D_{20}, D_{21} は頂点数が $max|D|$ を超えるため，無視される（4 行目）．また，ある部分木とクエリとの木編集距離を全て埋める為には，その部分木に含まれる全ての兄を持つ頂点と根に対して子から順に Algorithm 1 による木編集距離計算アルゴリズムを適用する必要がある（7-13 行目）．今見ている頂点の親を根とする部分木の頂点数が $max|D|$ を超えている場合，その頂点を根とする部分木が計算対象となる極大な部分木となる．よって，そのような頂点を根とする部分木に対しクエリ木との木編集距離を列挙し，θ 以下のものを検索結果として格納する（14-21 行）．

TASM [9] でも触れられているように，極大部分木を処理した後はそれらの頂点情報は必要なくなるため，20 行目時点で過去の頂点情報を全て破棄することができる．これにより，空間計算量は $O((|Q|+\theta)|Q|)$ に抑えることができる．

3.2 データベース木の分割

データベース木を分割し，複数のプロセスに木編集距離計算を割り当てることを考える．本手法では，木を先行順に並べた時にいくつかの連続する部分列になるようデータベース木を分割する．表 1 に，図 2 のデータベース木を先行順に並べた列を示す．

図 4，図 5 に 3 個のプロセスに割り当てるための 2 種類の分割例を示す．分割例 1 は先行順番号 $[0,9]$, $[10,17]$, $[18,21]$ で分割されている．分割例 2 は先行順番号 $[0,6]$, $[7,13]$, $[14,21]$ で分割されている．各図において，青い点線で示される辺は異なるプロセス間での頂点情報の通信が発生する可能性があること

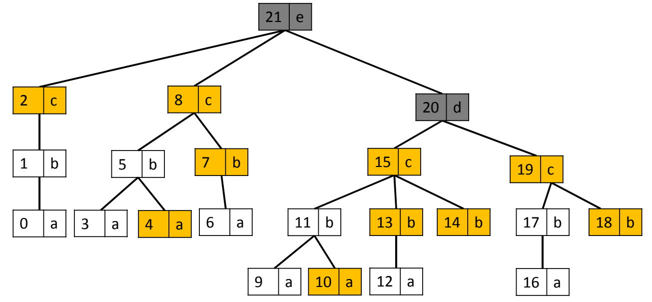

図 2: データベース木 D の一例

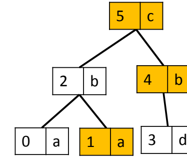

図 3: クエリ木 Q の一例

Algorithm 2 *SubtreeMatchingSingleProcess*

Require: D, Q, θ
Ensure: $Result = \{i \mid TED(D_i, Q) \leqq \theta\}$

1: $max|D| = |Q| + \theta$
2: **for** d_i : D by post-order **do**
3: $\quad$ **if** $|D_i| > max|D|$ **then**
4: $\quad\quad$ goto NEXTFOR
5: $\quad$ **end if**
6: $\quad p = par(d_i)$
7: $\quad$ **if** d_i は兄を持つ or d_i は根 or $|D_p| > max|D|$ **then**
8: $\quad\quad$ **for** j : 0 to $|Q|-1$ **do**
9: $\quad\quad\quad$ **if** q_j は兄を持つ or q_j は根 **then**
10: $\quad\quad\quad\quad TEDCalulation(D_i, Q_j)$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end for**
13: $\quad$ **end if**
14: $\quad$ **if** d_i は根 or $|D_p| > max|D|$ **then**
15: $\quad\quad$ **for** k : $left(d_i)$ to i **do**
16: $\quad\quad\quad$ **if** $d_k.TED[|Q|-1] \leqq \theta$ **then**
17: $\quad\quad\quad\quad$ add k to $Result$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end for**
20: $\quad\quad$ // we can release buffer
21: $\quad$ **end if**
22: $\quad$ NEXTFOR:
23: **end for**
24: return $Result$

を示す．分割例 1 において，D_{20} が計算対象部分木である場合，D_{20} の木編集距離の計算に $D_{16}...D_{19}$ の頂点情報（$left(d_i)$，値，木編集距離）が必要となるため，プロセス 3 からプロセス 2 へ頂点情報および計算結果を送信しなければならない．このような通信が発生する条件は，プロセスをまたぐ辺の親側の頂点を根とする部分木の頂点数（境界頂点数とする）が $max|D|$ を超えているかどうかで決まる．各プロセスにおける境界頂点数は，そのプロセスに与えられている先行順頂点列の先頭の頂点の親部分木の頂点数となる．分割例 1 において，プロセス 3 の境界頂点数，すなわちプロセス 3 からプロセス 2 への通信が起こる $max|D|$ の下限は，先行順番号 18 である頂点の親である頂点を根とする部分木の頂点数 $|D_{20}| = 12$ となる．

分割例 1 は $max|D|$ が 12 未満の場合に通信が発生しないが，分割例 2 は 3 以上の場合に通信が発生してしまう．しかし，各プロセスに割り当てられる頂点数は分割例 1 よりも分割例 2 の方が均等に割り振られており，木編集距離計算回数の偏りは少ないと考えられる．一般に，プロセス間通信のコストが大きい場合は分割例 1 のように境界頂点数が大きい方が効率がよく，プロセス間通信のコストが小さい場合は分割例 2 のように割り当てられる頂点数が均等である方が効率が良いと考えられる．

表 1: 図 2 の木を先行順に並べた時の頂点列

先行順	0	1	2	3	4	5	6	7	8	9	10
後行順	21	2	1	0	8	5	3	4	7	6	20
先行順	11	12	13	14	15	16	17	18	19	20	21
後行順	15	11	9	10	13	12	14	19	17	16	18

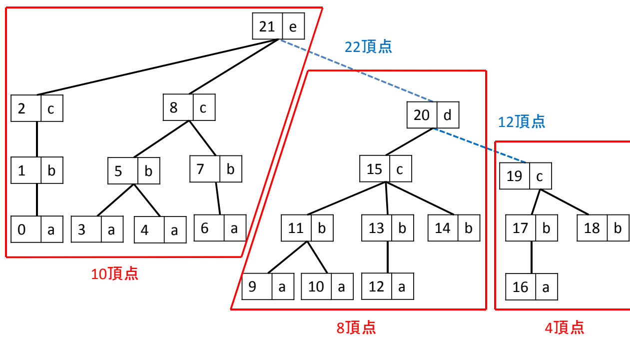

図 4: データベース木の分割例 1

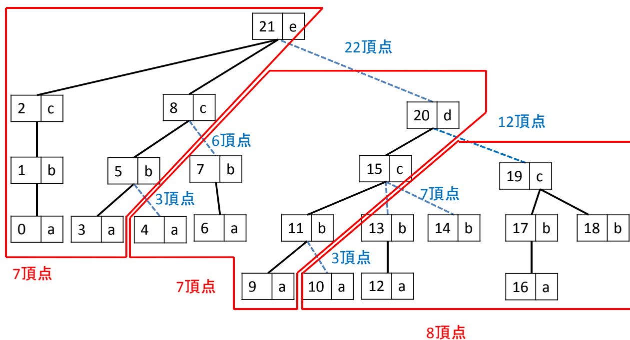

図 5: データベース木の分割例 2

Algorithm 3 *DivideTree*

Require: D, t, k
Ensure: $Assign[|D|]$
1: $l \Leftarrow 1, r \Leftarrow |D|$
2: **while** **do**
3: $\quad x \Leftarrow (l + r)/2$
4: $\quad j \Leftarrow 1$
5: $\quad$ **for** d_i : D by preorder **do**
6: $\quad\quad Assign[i] = j$
7: $\quad\quad$ **if** $j < k$ and d_{i+1} で分割可能 and プロセス j の頂点数が x 以上 **then**
8: $\quad\quad\quad j \Leftarrow j + 1$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad$ **if** $l = x$ **then**
12: $\quad\quad$ goto BREAKWHILE
13: $\quad$ **end if**
14: $\quad$ **if** $j = k$ **then**
15: $\quad\quad l \Leftarrow mid$
16: $\quad$ **else**
17: $\quad\quad r \Leftarrow mid$
18: $\quad$ **end if**
19: **end while**
20: BREAKWHILE:
21: return $Assign$

本手法では，プロセス数 k と各プロセスの境界頂点数の下限 t をユーザが指定し，その条件下で各プロセスに割り当てられる頂点数の最大値を最小化するような分割を行う．Algorithm 3 に分割アルゴリズムを示す．なお，このアルゴリズムにおいてのみ d_i は 木 D の先行順番号が i である頂点を示す．

このアルゴリズムでは，頂点数の最大値を二分探索することで,「木 D を，与えられた条件下で，x 頂点以上の k 個の連続部分列に分解できるか」という問題に帰着する．この問題は，先頭から順に見ていき，切ることができる部分が来た時に，現在の頂点数が x 以上であるならば貪欲に切る，という方法を取り，実際に k 個に切り分けられたかどうかで判定することができる．また，ここでの d_i と d_{i+1} の間で切ることができるための条件とは「d_i が葉である」かつ「d_{i+1} の親部分木の頂点数が t より大きい」である．

3.3 マルチプロセスアルゴリズム

マルチプロセスでの類似部分木検索アルゴリズムを Algorithm 4 に示す．なお，このアルゴリズムは SPMD(Single Program Multiple Data) であり，r を現在実行しているプロセスのランクとする．D^r は r 番目のプロセスに割り当てられている頂点を後行順で格納した頂点配列を表す．D^r は，各頂点の後行順番号，頂点の値と $left(d_i)$ に加え，親部分木の頂点数を表す $parSize$ と，親の頂点が属するプロセスのランクを表す $parRank$，d_{i+1} が属するプロセスのランクを表す $nextRank$ を持つ．

$r = 2$ 番目のプロセスにおいて，図 5 に従い分割されたデータベース木と図 3 のクエリ木に対して $\theta = 3$ でこのアルゴリ

ズムを適用した場合について考える．表 2 に，D^2 に含まれる情報を示す．Algorithm 2 との違いは主に受信部分である 7-13 行目と，送信部分である 29-32 行目である．d_4 を処理した時点で $parRank$ が 1 を示しており，$parSize$ が $max|D|$ 以下であるため，プロセス 3 に頂点情報を送信しなければならない（29-30 行目）．同様に，d_6, d_7 も d_7 の処理後にプロセス 1 に送信される．次に，d_{11} を処理する際に $next$ は 3 を示しているため，プロセス 3 から頂点情報を受信しなければならない（9 行目）．d_{10} をプロセス 3 から受信した時点で $next$ が 2 になり，処理が続行される（13 行目）．同様にして $d_{12}..d_{14}$ も受信する．最後に，d_{20} を処理するが，$|D_{20}| > max|D|$ であるため，全ての処理は無視される（5 行目）．

このアルゴリズムの重要な特徴として，先行順で木を分割し後行順で処理を行うことで，ほとんどの場合に各頂点情報の送信タイミングが受信タイミングより早くなることが期待できるという点がある．これにより，プロセスをまたぐ頂点の処理の依存関係による通信待ちを防ぐことが期待できる．

表 2: 頂点配列 D^2

後行順	値	$left(d_i)$	parSize	parRank	nextRank
4	a	4	3	1	1
6	a	6	2	2	2
7	b	6	6	1	1
9	a	9	3	2	3
11	b	9	7	2	3
15	c	9	12	2	3
20	d	9	22	1	1

4. 評価実験

4.1 実験の目的および実験環境

本節では，実際に巨大な XML ファイルを用いて並列計算を行うことで，手法の性能を評価する．実験は CPU: Xeon E5-2650 v3 2.3GHz 10 cores (hyper-threading: 20 cores)，Disk: Intel Solid-State Drive 730 Series 240 GB を搭載したマシンを 9 ノード用意し，最大 180 コアの環境で行った．

実験では，データベース木 D として実データである dblp [13], SwissProt [14]，ProteinSequenceDatabase (PSD) [15] を使用した．dblp, SwissProt，PSD の XML ファイルは XML Data Repository [16] の物を利用した．また，Q にはそれぞれ D から抽出した適当なサイズの部分木を使用した．

本実験では，木編集距離の計算時のコストはどの操作も一律 1 として計算する．XML ファイルは予めパースし，プロセスの数に合わせて分割し，バイナリファイルに保存しておくものとする．また，各頂点が持つ値は XML ファイルの各頂点が持つラベルのハッシュを取り整数化したものを使用する．

4.2 並列化効率の評価

3 種の XML データに対し，1 プロセスから 180 プロセスで類似部分木検索を行い実行時間を計測し，並列化効率を評価する．なお，r プロセスでの実行時間が T_r とした時の，$\frac{T_1}{rT_r}$ を

Algorithm 4 *SubtreeMatchingMultiProcess* for Node r

```
Require: D^r, Q, θ, r
Ensure: Result = {i | TED(D_i^r, Q) ≦ θ}
 1: max|D| ⇐ |Q| + θ
 2: next ⇐ r
 3: for d_i : D^r by post-order do
 4:   if |D_i| > max|D| then
 5:     goto NEXTFOR
 6:   end if
 7:   while next ≠ r do
 8:     if next > r then
 9:       receive d_j from next node
10:     else
11:       next ⇐ r
12:     end if
13:   end while
14:   p = par(d_i)
15:   if d_i は兄を持つ or d_i は根 or |D_p| > max|D|  then
16:     for j : 0 to |Q| − 1 do
17:       if q_j は兄を持つ or q_j は根 then
18:         TEDCalulation(D_i, Q_j)
19:       end if
20:     end for
21:   end if
22:   if d_i は根 or |D_p| > max|D| then
23:     for k : left(d_i) to i do
24:       if d_k.TED[|Q| − 1] ≦ θ then
25:         add k to Result
26:       end if
27:     end for
28:     // we can release buffer
29:   else if d_i.parRank ≠ r then
30:     send d_i to d_i.parRank node
31:     // we can release buffer
32:   end if
33:   NEXTFOR:
34:   next = d_i.nextRank
35: end for
36: return Result
```

並列化効率とする．$|Q| = 64, \theta = 20$ とし，分割時の t は通信が発生しない程度に大きな値である 100 を使用した．

図 6 にプロセス数を変化させた時の実行時間との並列化効率を表す．結果では，プロセス数 9 から 18 にかけて大きく性能が減少しているが，これは実験環境が 9 ノード × 20 コアであるため，ディスクのシークにかかる時間が発生しオーバヘッドが生じていると考えられる．$r = 1...9$ および $r = 18...180$ にかけて，並列化効率を維持していることが確認できるため，本手法は充分なスケーラビリティを持つと考えられる．

4.3 境界頂点数による通信量と実行時間のばらつきのトレードオフ

データベース木 D の分割時に指定する t の値を変化させた時にデータ通信量と実行時間がどのように変化するかを調べる．$|Q| = 512, \theta = 20$ とし，$t = 1, 10, 100, 1000$ の 4 種類の分割

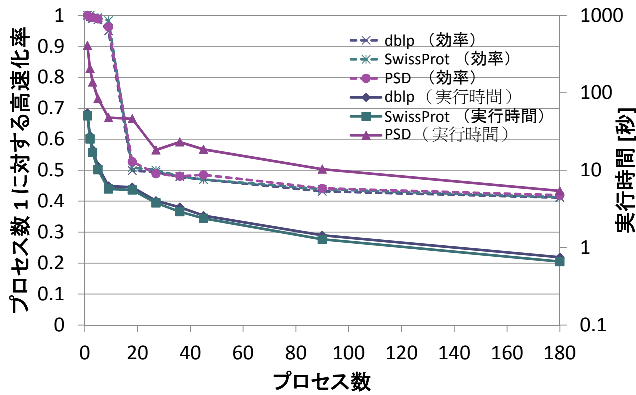

図 6: 並列化効率の評価

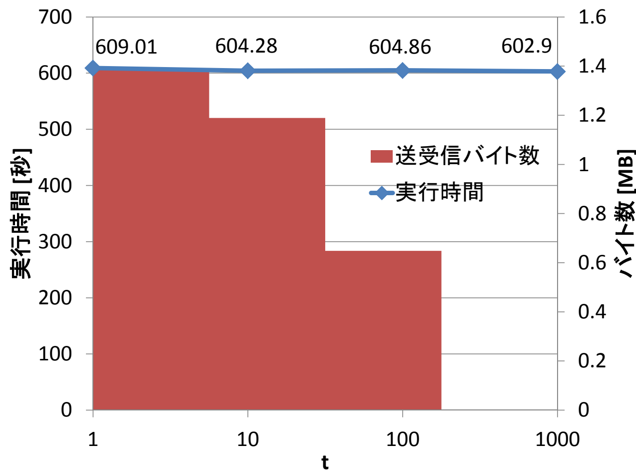

図 7: 境界頂点数による実験 (9 プロセス)

方法で実験した．

図 7 に 9 プロセスでの実験結果を，図 8 に 180 プロセスでの実験結果を示す．実験結果では想定通り t が大きくなるほどデータ通信量が減り，わずかながら実行時間も減っていることが確認できる．また，今回の設定では $max|D| = 532$ であるため， $t = 1000$ の場合にはデータ通信が発生しない．今回の実験環境はローカルネットワーク内で構成されたクラスタであるため，プロセス間の通信コストが比較的小さい．通信コストがさらに大きい環境で実行する場合は通信量の差による実行時間の変化もより大きくなると考えられる．

また，さらに t の値を増やしていくと分割された木の頂点数に偏りが生じ実行時間が伸びていくと考えられる．しかし，今回使用したデータセットは木の頂点数に比べ木の高さが低く，根から出る辺のみで偏りの生じない分割が可能となり，$t = |D|$ のような極端なパラメータ設定でも，180 プロセス程度であればある程度偏りの無い分割が可能となり，差が生じなかった．一般的に XML データはこのような理由から分割の自由度が高いと考えられるが，もっと分割の自由度が低いような木構造データを対象とする場合は t を値を増やすことにより性能が低下する可能性があると考えられる．

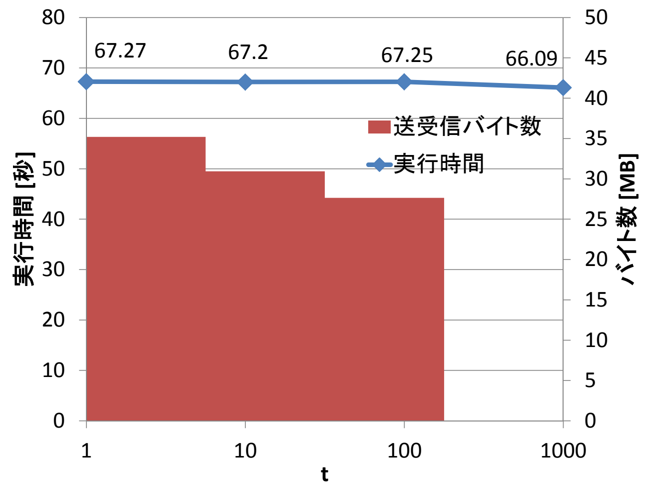

図 8: 境界頂点数による実験 (180 プロセス)

5. ま と め

本稿では木構造データに対する類似部分木検索という問題を対象とし，複数ノードによる並列処理による手法を提案した．同じ問題に対する既存の研究では枝刈りによる高速化が主だったが，本研究では並列化という方法により高速化を成し遂げた．実験による並列化性能の評価では，充分なスケーラビリティを確認できた．また，本手法は実験環境の通信コストの大小や想定されるクエリや閾値の大きさから，状況に適したデータの割り当てを行う方法を提案した．これは処理性能の良し悪しを左右することが予想されるが，実験では用いたデータの特徴から，通信量の違いによる時間の差しか確認することができなかった．より局所性の高い木構造データを用いることで分割データの偏りによる性能への影響が確認できることが期待できるが，これは今後の課題とする．また，関連研究である [10] や [12] では構造の局所性を利用し高速化を行っている．本手法にもこの工夫を取り入れることでさらなる実行時間の改善が期待できる．しかしながら，並列化効率を維持したまま取り入れることは難しいと考えられるため，これも今後の課題とする．

謝 辞

本研究の一部は JSPS 科研費 25330124 の助成を受けたものである．

文 献

[1] Joe Tekli, Richard Chbeir, and Kokou Yetongnon. Survey: An overview on XML similarity: Background, current trends and future directions. *Comput. Sci. Rev.*, Vol. 3, No. 3, pp. 151–173, aug 2009.

[2] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, Vol. 337, No. 1-3, pp. 217–239, jun 2005.

[3] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, Vol. 26, No. 3, pp. 422–433, July 1979.

[4] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, Vol. 18, No. 6, pp. 1245–1262, December 1989.

[5] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual Eu-*

ropean Symposium on Algorithms, ESA '98, pp. 91–102, London, UK, UK, 1998.

[6] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, Vol. 40, No. 2, pp. 135–158, August 2001.

[7] Joe Tekli, Richard Chbeir, and Kokou Yétongnon. Efficient XML structural similarity detection using sub-tree commonalities. In *XXII Simpósio Brasileiro de Banco de Dados, 15-19 de Outubro, João Pessoa, Paraíba, Brasil, Anais*, pp. 116–130, 2007.

[8] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pp. 90–101, San Francisco, CA, USA, 1999.

[9] Nikolaus Augsten, Denilson Barbosa, Michael M. Bohlen, and Themis Palpanas. Efficient top-k approximate subtree matching in small memory. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 23, No. 8, pp. 1123–1137, 2011.

[10] Sara Cohen. Indexing for subtree similarity-search using edit distance. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 49–60, New York, NY, USA, 2013.

[11] 小柳涼介, 天笠俊之, 北川博之. テキストおよび構造の類似度に基づいた XML データに対する効率的な類似検索. In *DEIM Forum 2014 D7-5*, 2014.

[12] 小柳涼介, 天笠俊之, 北川博之. 構造情報の再利用による XML データに対する類似検索の高速化. 情報科学技術フォーラム講演論文集, Vol. 13, No. 2, pp. 71–72, aug 2014.

[13] dblp computer science bibliography. `http://dblp.uni-trier.de/db/`.

[14] Uniprotkb/swiss-prot. `http://web.expasy.org/docs/swiss-prot_guideline.html`.

[15] Protein information resource. `http://pir.georgetown.edu/`.

[16] XML data repository. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.