階層的分割による並列連想計算

Hierarchical Table Structures for Parallel Association Computation

松田 一孝 [†] 西岡 真吾 [†] 胡 振江 [†‡] 武市 正人 [†]
Kazutaka Matsuda, Shingo Nishioka, Zhenjiang Hu, and 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
[‡] 科学技術振興機構 さきがけ研究 21
PRESTO 21, Japan Science and Technology Agency

文書群同士，単語群同士，または文書群と単語群間の類似性を計算する手法として，連想計算による検索エンジン GETA が実現されている．ここでは，連想計算により文書群と単語群の間から最も類似度の高い n 個の要素を得る問題を対象とする．連想計算の並列分散化は，ある確率以上で正しい結果を返す確率的な連想計算関数について GETA 上で既に行なわれているが，データである文書群と単語群のなす行列を，行か列の一方向にしか分割することができない．本研究では分割をデータ並列の視点から階層的に捉え，確率的な連想計算関数の再定式化を行なう．これにより行と列との両方向にデータを自由に分割できるようになる．また，データ並列の枠組を実現した並列スケルトンを用いて実装を行い，その有効性を評価する．

1 はじめに

文書群同士，単語群同士，また文書群と単語群間の類似度を計算する手法として，NII(国立情報学研究所) の高野らによって，連想計算 [6] による検索エンジン GETA(Generic Engine for Transposable Association)[7] が実現されている．

GETA で，コーパスを表すデータ構造は，WAM(Word Article Matrix) という形で，表現されている．WAM は疎行列表現であり，文書検索においては，表 1 の例のように，行に文書，列に単語を対応させ，その要素はある文書に含まれる単語の頻度を表す行列となる．

表 1: WAM の例

	単語 1	単語 2	単語 3	単語 4
文書 1	1	0	3	4
文書 2	0	2	1	2
文書 3	0	1	3	0

表 1 のデータは，文書 1 が，$(1, 0, 3, 4)$ というように文書は単語の頻度で表わされている．また列ベクトルを考え，単語 1 が $(1, 0, 0)$ というように文書の頻度で表されていると考えることもできる．GETA で扱う連想計算は，単語/文書の重みつき集合からなる query から，文書/単語の重みつき集合を返す計算である．返される文書/単語の重みは，その文書と query との類似度になる．表 1 の例で，単語 1 が 1，単語 2 が 2 という query に対し連想計算を行なうと，文書 1 が 0.125，文書 2 が 0.8，文書 3 が 0.5 などといった文書の重みつき集合が返る．さらに，返ったものを入力として用いることもできる．このような，行と列に対する双対性を GETA の連想計算の転置可能性 (transposability) という．以下，連想計算というと，この GETA の連想計算を指す．

1.1 GETA における確率的な連想計算

通常，文書の数，単語の数は非常に多く，また，結果としてもとめらるものは上位 n 個といった場合が多い．さらに，必要とされるのが上位 n 個である場合連想計算関数が類似度を計算した全ての結果を返すと，空間的にも効率が悪い．

現在の GETA は階層的でない平坦な分割を行っているが，分割をしていても，要求文書数が多い場合，分割したノードからのデータ転送が時間，空間ともに要求文書数と分割数との積のオーダーになってしまう．ここで現在の GETA では，ある確率 p 以上で正しい上位 n 個の結果をを要求する連想計算関数を用いることでこの問題を解決している．この連想計算関数により，128 分割において，要求文書数が 1000 個，誤り率を 1/1,000,000 とした場合に分割したノードに要求する文書の数を 28 個におさえることができ

る (通信は 3584 個) という結果を得ている [5].

1.2　本研究の目的

GETA の連想計算関数自体は既に並列化されている．しかし，その並列化の手法は，単語から文書方向では行方向のみ，文書から単語方向では列方向のみの一方向しか分割できない．さらに大きいデータを扱えるようにするために，WAM を両方向に分割できるようにする必要がある．

本研究では，ある方向に分割を繰り返し，そしてまた違う方向に分割を繰り返すという，階層的なデータの分割を用いて GETA の連想計算関数を再定式化し，また，それを並列スケルトンを用いた形に変形を行い，実装する．

本論文では，プログラム中の関数を表現するのに，関数型言語 haskell[3] の記法を用いる．

2　連想計算の並列のための定式化

本節では，列インデックスの重みつき集合を query として，行インデックスの重みつき集合を得る場合を考える．GETA の連想計算は，転置可能であるためこのような仮定をおいても，一般性を崩すことはない．

2.1　階層的な分割

階層的な分割を表すデータ構造として，二分木

$$\begin{aligned} \mathit{BTree}\ \alpha\ \beta &= \mathit{Node}\ \alpha\ (\mathit{BTree}\ \alpha\ \beta, \mathit{BTree}\ \alpha\ \beta) \\ &\quad \mid \mathit{Leaf}\ \beta \end{aligned}$$

を考える．

データ構造の上での重要な概念の一つとして，catamorphism がある．二分木の上の catamorphism は，$(\!|f_N, f_L|\!)$ と書き，

$$\begin{aligned} &(\!|f_N, f_L|\!)\ (\mathit{Node}\ n\ l\ r) = \\ &\qquad f_N\ n\ ((\!|f_N, f_L|\!)\ l)\ ((\!|f_N, f_L|\!)\ r) \\ &(\!|f_N, f_L|\!)\ \mathit{Leaf}\ n = f_L\ n \end{aligned}$$

とう性質を表わす．

本研究では，この二分木を用いて WAM を分割することを考える．二分木を選んだ理由は，既にその並列スケルトンに関する研究が，行われていること，また，行列上の並列スケルトンに関して，いまだそれほど研究が行われていないこと，さらに，再帰的に分割を考えることができるのことなどが挙げられる．

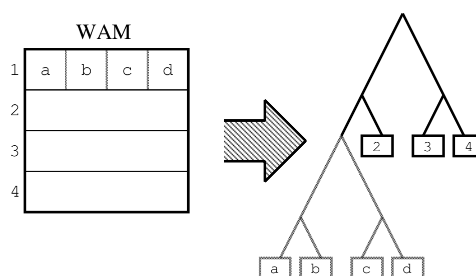

図 1: WAM の階層的分割

以下，行方向分割においては，既存の GETA のアルゴリズムを二分木というデータ構造を用いて定式化し，列方向分割については，行方向分割において末端に割り振られるデータをさらに分割していくことを考える．つまり，最終的に扱うデータ構造は図 1 のようになる．

2.2　行方向分割

2.2.1　確率的な連想計算

GETA における確率的な連想計算は平坦な分割に関してのものであったので，ここではそれを二分木を用いたものへと定式化しなおす．

あるノードでほぼ均等に WAM が分割されている仮定して，子が確率 p' で正しい行を m' 個返すとき，自分自身が確率 p 以上で正しい行を n 個返すための条件を求める．左の子に正しい行が k 個割当てられる確率は，WAM の行数が要求文書数より十分に大きいため，超幾何分布を二項分布で近似して

$$\begin{pmatrix} m \\ k \end{pmatrix} q^m (1-q)^{m-k}$$

となる．ただし

$$q = \frac{\text{左の子に割り振る WAM の行数}}{\text{子に割り振る WAM の行数の和}}$$

である．このとき，左の子から得た m' 個および，右の子から得た m' 個が正しい上で得られた $2m'$ に正解が入っていないという事象の条件付き確率確率 e は，右の子に k より多い ≡ 左の子に $m-k$ 未満 であるから

$$e(m') = \sum_{m'<k, k<m-m'} \begin{pmatrix} m \\ k \end{pmatrix} q^n (1-q)^{m-k}$$

となる．ここで，左の子から得た m' 個が正しく，右の子から得た m' 個が正しく，得られた $2m'$ 個に正

解が入っている確率が p 以上であればよい，つまり

$$p'^2(1-e(m')) \geq p$$

となればよい．これを満すような m', p' を用いて，確率 p' で正しい行を m' 個子ノードに要求すれば，このノードで確率 p 以上で正しい m 個の行を返すことができる．

ある超幾何分布と，それを近似する二項分布では，近似する二項分布のほうが分散が大きいため，超幾何分布を二項分布で近似した結果，p をよりきつく見積っていることになる．

m' を，p' が前述の条件を満たす最初が m' とする場合，WAM を 128 分割 (つまり 7 階層)，要求文書数 1000，誤り率 1/1,000,000 以下とした場合，上から順に 1000，577，354，231，162，121，96，81 と要求していく．うけとるべき，2 つ文書は通常そのうち 1 つを自分がもっているため総通信量は $577+354+231+162+121+96+81=1622$ 個の文書送信分である．これは，従来の GETA の総通信量が 3584 個分であるのに比べ小さくなっている．

2.2.2 確率，要求文書数の分配と集約の定式化

このような，確率，要求文書数を各ノードに分配し，葉での連想計算の結果を集約していくという確率的な連想計算の操作を定式化すると次のようになる：

$$\begin{aligned}
&\mathit{assoc}\ \mathit{red}\ \mathit{ext}\ \mathit{dist}\ \mathit{base}\ d\ q\ (\mathit{Node}\ n\ l\ r) = \\
&\qquad \mathit{red}\ d\ q\ n \\
&\qquad\quad (\mathit{assoc}\ \mathit{red}\ \mathit{ext}\ \mathit{dist}\ \mathit{base}\ (\pi_1\ d')\ q\ l) \\
&\qquad\quad (\mathit{assoc}\ \mathit{red}\ \mathit{ext}\ \mathit{dist}\ \mathit{base}\ (\pi_2\ d')\ q\ r) \\
&\qquad\qquad \textbf{where}\ d' = \mathit{dist}\ n\ (\mathit{ext}\ l)\ (\mathit{ext}\ r)\ d \\
&\mathit{assoc}\ \mathit{red}\ \mathit{ext}\ \mathit{dist}\ \mathit{base}\ d\ q\ (\mathit{Leaf}\ n) = \mathit{base}\ d\ q\ n \\
&\mathit{ext} = (\![\mathit{extNode}, \mathit{extLeaf}]\!)
\end{aligned}$$

ext，*dist*，*red* は，図 2 のように振舞う関数である．*ext* により，分割された WAM から情報を引き出し，*dist* で，要求文章数などの情報を，*ext* で得られた情報を用いて，分配し，*red* によりそれらをまとめあげる．

関数 *assoc* をに適当な引数を渡すことで確率的な連想計算を実現することができる．具体的な例をあげると，*ext* は WAM の行数の抽出，*dist* は要求文書数と確率の分配，*base* は分割された WAM に対する連想計算であり，*red* はそれをマージする関数になる．

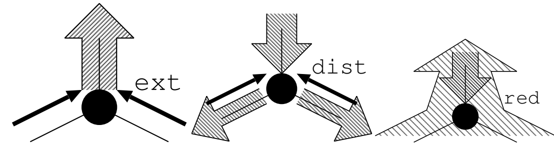

図 2: *ext*，*dist* および *red* の働き

2.3 列方向分割

行方向分割で，GETA の計算を二分木上で考えることによって，並列スケルトンで記述できることが示された．本節では，行方向分割における葉の部分での処理を，同じように分割できないかを議論する．これは，各行についての query との類似度の計算の並列化に関する議論である．ここで，GETA では，類似度の計算に query, 行の要素の写像を行ない，さらにその内積をとり，また正規化するという処理をとっている．必要な情報をあらかじめ各ノードに分配しておけば，このような計算の多くは，行方向分割と同じ方法で記述することができる．

これらの理由から列方向分割を関数 *assoc* を用いてあらわせると仮定し，

$$\begin{aligned}
&\mathit{assocL}\ \mathit{redL}\ \mathit{extL}\ \mathit{distL}\ \mathit{baseL}\ d\ q\ t = \\
&\qquad \mathit{trL}\ (\mathit{assoc}\ \mathit{redL}\ \mathit{extL}\ \mathit{distL}\ \mathit{baseL}\ d\ q\ t
\end{aligned}$$

という関数を考える．但し $\mathit{extL} = (\![\mathit{extLN}, \mathit{extLL}]\!)$ である．

2.4 両方向分割

これまでは，どのようにデータを分割するか，とそこで計算に用いる関数がどのようなものになるかを議論してきた．しかし，まだ全体に関してどのようにデータが流れているかを規定していない．

$$\mathit{BTree}\ \alpha\ (\mathit{BTree}\ \beta\ \gamma)$$

のようなデータ構造を考える．最外の二分木が行方向の分割を表し，また最外の二分木の葉での二分木が列方向の分割を表すとする．

そして，分割の仕方の違う 2 つの *assoc* を

$$\begin{aligned}
&\mathit{assocComp}\ \overline{\mathrm{fs}}\ d\ q\ t\ = \\
&\quad \mathit{assoc}\ \mathit{redH}\ (\![\mathit{extHN}, \mathit{extHL}\cdot\mathit{trE}\cdot\mathit{extL}]\!)\ \mathit{distH} \\
&\quad\ (\lambda d' q' n'.(\mathit{trHL}\ d'\cdot\mathit{trL} \\
&\qquad \cdot(\mathit{assoc}\ \mathit{redL}\ \mathit{extL}\ \mathit{distL}\ \mathit{baseL} \\
&\qquad\quad (\mathit{trD}\ d'\ (\mathit{extL}\ n'))\ q'))\ n') \\
&\qquad\quad\ (\mathit{top}\ d \\
&\qquad\qquad ((\![\mathit{extHN}, \mathit{extHL}\cdot\mathit{trE}\cdot\mathit{extL}]\!)\ t))\ q\ t
\end{aligned}$$

のように連結させる．$\overline{\mathrm{fs}}$ とは，*extLN*，*extLL*，*trE*，*extHN*，*extHL*，*distH*，*top*，*trD*，*distL*，*baseL*，*redL*，*trL*，*trHL*，*redH* という 14 個の関数を表す．

この関数 *assocComp* は，木を葉に持つ木に対し，情報を抽出し，分配し，まとめあげるという操作を上部の木と下部の木でつなげたものである．

2.5 記述性

ここでは，詳しくは述べないが，この枠組のもとで，あらかじめ必要なデータを分配しておけば，GETA のチュートリアル [7] にある，ヒットする検索単語数を用いる方法，tf*idf 法，Singhal 法の 3 つ連想計算関数を記述することができる．

3 並列スケルトンによる *assocComp* の実装

assocComp はそのままの形では，並列化の議論をしづらい．そこで，この関数を並列スケルトンで記述することにより，この関数が並列化可能であることを示す．

3.1 二分木上の並列スケルトン

並列スケルトンは，スケルトン並列プログラミング [1] で用いられる，あらかじめ用意されたプログラムの構成部品である．これを用いることにより，その部分での計算が並列に実行されることが保証される．つまり，あるプログラムを並列スケルトンの組合せで書くことができると，そのプログラムを並列に実行することができる．個々のスケルトンは高階関数であり，二分木上の基本的なスケルトン [4] は map，zip，reduce，upwards accumulate そして downwards accumulate である．ここでは，松崎ら [2] による，スケルトンの定義を用いる．

3.2 *assocComp* の実装

関数 *assocComp* を，図 3 のように，二分木に対する並列スケルトンの組合せで記述することができる．並列スケルトンの部分は並列計算可能であることが，保証されているので，これにより関数 *assocComp* を並列化することができた．

表 2: 0 番プロセスの maximum resident set size

	列 1 段階分割	列 2 段階分割
行分割なし	3800	4200
行 1 段階分割	2600	2700
行 2 段階分割	2200	2003

但し，図 3 では

$$\begin{aligned}
&\mathit{myAcc}\ (f_N, f_L) = \\
&\quad \mathit{uAcc}(\lambda n, a, b.(f_N\ n\ (\pi_1\ a)\ (\pi_1\ b), \\
&\qquad\qquad ((\pi_1\ a), (\pi_1\ b))), \\
&\qquad \lambda n.(f_L\ n, (f_L\ n, f_L\ n))) \\
&\mathit{zipN}\ t_1\ t_2 = \mathit{map}(\mathrm{id}, \pi_1)\ \mathit{zip}(t_1, t_2) \\
&\mathit{zipL}\ t_1\ t_2 = \mathit{map}(\pi_1, \mathrm{id})\ \mathit{zip}(t_1, t_2) \\
&\mathit{mapN}\ f = \mathit{map}(f, \mathrm{id}) \\
&\mathit{mapL}\ f = \mathit{map}(\mathrm{id}, f)
\end{aligned}$$

という関数を用いている．これらの関数は，すべて並列スケルトンの組み合わせで書かれている．

4 実験

実装は，C 言語で MPI を用いて行った．データとしては，OpenGL Mailing List(日本語)[8] に投稿された 1996 年から 1999 年のメールから，形態素解析器の都合のためアルファベット，数字を取り除いたものを用いた．このとき WAM のサイズは約 7200× 約 12700 となる．今回の実装の目的は使用メモリ量の削減である. そのための指標として，システムコール `getrusage` で取得できる，maximum resident set size を用いた．これが小さいほど，利用しているメモリのサイズが小さいことが言える．今回の実装では，0 番プロセスには根から最左の葉までの全てのノードが割当てられることになる. そのため，0 番プロセスが一番メモリを必要とする. また，分割は，ある方向の分割数 p とすると，インデックスを 2^p で割った余りを用いて分割を行った．

表 2 は，0 番プロセスの maximum resident set size を Kbyte 単位で示したものである．行 p 段階分割列 q 段階分割というのは，上部の木として深さ p 下部の木として深さ q のものを用いたということを表わす．これを見ると，分割を行った方が使用メモリが少くなっていることわかる．列方向に分割する場合にあまりメモリ効率がよくなっていないのは，*redL* が，行数に比例したデータを送信するため及び，列

$$
\begin{aligned}
&assocComp\ \overline{\mathrm{fs}}\ d\ q\ t = \\
&\quad \mathbf{let}\quad t_1 = mapL\ (myAcc\ (extLN, extLL))\ t \\
&\quad\quad\quad t_2 = (myAcc\ (extHN, extHL) \cdot mapL(trE.\pi_1, root))\ t_1 \\
&\quad\quad\quad t_3 = dAcc\ distH'\ (top\ d\ (\pi_1 \cdot (root\ t_2)))\ (zipN\ (t, (mapN\ \pi_2\ t_2))) \\
&\quad\quad\quad t_4 = mapL\ (\lambda(a,b).trD\ a\ b)\ (zipL\ (mapN\ K_{dum}\ t_3, (mapL\ (\pi_1 \cdot root)\ t_1))) \\
&\quad\quad\quad t_5 = mapL\ (\lambda(a,(b,c)).dAcc\ distL'\ a\ (zip\ (b,c)))\ (zipL\ (t_4, (zipL\ (t, mapL\ (mapN\ \pi_2))t_1)))) \\
&\quad\quad\quad t_6 = mapL\ (trL \cdot reduce(redL', baseL') \cdot zip)\ (zipL\ (t, t_5)) \\
&\quad\quad\quad t_7 = mapL\ (\lambda(a,b).trHL\ b\ a)\ (zipL\ (t_6, t_3)) \\
&\quad \mathbf{in} \\
&\quad reduce\ (redH', \mathrm{id})\ (zipN\ (t_7, t_2)) \\
&\quad \mathbf{where}\quad distH' = (\lambda(n,(a,b))\ c.\pi_1\ (distH\ n\ a\ b\ c), \lambda(n,(a,b))\ c.\pi_2\ (distH\ n\ a\ b\ c)) \\
&\quad\quad\quad distL' = (\lambda(n,(a,b))\ c.\pi_2\ (distL\ n\ a\ b\ c), \lambda(n,(a,b))\ c.\pi_2\ (distL\ n\ a\ b\ c)) \\
&\quad\quad\quad baseL'_{list}\ (n,d) = baseL\ d\ q\ n \\
&\quad\quad\quad redL'_{list}\ (n,d)\ l\ r\ = redL\ d\ q\ n\ l\ r \\
&\quad\quad\quad redH'\ (n,d)\ l\ r\ = redH\ d\ q\ n\ l\ r \\
&\quad\quad\quad K_a\ b = a
\end{aligned}
$$

図 3: *assocComp* の並列スケルトンを用いた定義

方向に 2 つ分割した際に WAM のサイズに大きな偏りが生じたことが原因であろう．

今回は，簡単な実装であったため，通信のコストが大きくなったが，*redL*，*baseL* をもっと効率よく実装すると，もっと通信に関する空間コストをおさえられ，分割された WAM のサイズが使用メモリの主原因になってくることが予想される．

5 結論と今後の課題

GETA での連想計算関数を木に関する並列スケルトンを用いて書くことができた．これにより，行列を行方向，列方向ともに分割できるようになり，また，既存の並列スケルトンに関する議論，研究をこの問題に対し適応することができるようになる．

今回は WAM が与えられた上で議論をしてきたが，その WAM を分配する場合に多くの資源を必要とする．この関数をどう定式化するかは一つの課題である．

また，本来なら行列とベクトルに関する演算を今回は階層的な分割を用いて定式化し，階層的な分割にあわせた関数を定義することで計算した．そのため，14 個の関数を定義することになり，非常に連想計算関数の定義が繁雑になってしまった．これを自動的に導出することができるかどうか検証する．

参考文献

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

[2] K. Matsuzaki, Z. Hu, and M. Takeichi, Parrallelization with Tree Skeletons, *Technical Report METR 2003-21*, Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2003.

[3] S. Peyton Jones and J. Hughes, editors. *Haskell98: A Non-strict, Purely Functional Language.* Available online:`http://www.haskell.org`, February 1999.

[4] D.B. Skillicorn, Parallel Implementation of Tree Skeletons.*Journal of Parallel and Distributed Computing*,vol 39, no. 2, pp. 115–125,1996.

[5] 高野 明彦 et al., 汎用連想計算エンジンの開発と大規模文書分析への応用, `http://geta.ex.nii.ac.jp/pdf/itx2002.pdf`.

[6] A. Takano, S. Nishioka, M. Iwayama, T. Hisamitsu, O. Imaichi, and H. Sakurai, Information Access Based on Associative Calculation, In *Proceedings of Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 187-201, Milovy, Czech Republic, 2000. Springer-Verlag.

[7] 汎用連想計算エンジン GETA `http://geta.ex.nii.ac.jp/`.

[8] OpenGL Mailing List `http://www.gimlay.org/~andoh/opengl/ml/`.