マルチコア CPU 上での段階的一般化法の並列処理

Parallel Processing for Step-wised Generalization Method on a Multi-core CPU

八木　真平
Shinpei Yagi
広島市立大学情報科学研究科
Email: yagi@de.info.hiroshima-cu.ac.jp

田村　慶一
Keiichi Tamura
広島市立大学情報科学研究科
Email: ktamura@hiroshima-cu.ac.jp

北上 始
Hajime Kitakami
広島市立大学情報科学研究科
Email: kitakami@hiroshima-cu.ac.jp

Abstract— **An ambiguous query processing returns a large number of similar subsequences, called a mismatch cluster, to the user. It is difficult for the user to discover the characteristics from a mismatch cluster. Then, the generalization method is proposed as a technique for extracting the minimum generalization set that expresses the mismatch cluster. In this paper, we proposes the parallelization method for running the step-wised generalization method on a multi-core CPU.**

I.　はじめに

曖昧な問合せ処理では，非常に多くの類似部分文字列（ミスマッチクラスタ）が検索結果として得られる．ミスマッチクラスタをユーザが直接閲覧して規則性を把握することは困難である．そこで，ミスマッチクラスタを表現する最小汎化集合を抽出する手法として段階的一般化法が提案されている[1].

段階的一般化法ではミスマッチクラスタ *MIS* の冪集 2^{MIS}（$2^{|MIS|}$個の部分集合 *SubMIS* の集まり）に含まれる要素を，ボトムアップに小さいサイズの要素から順に列挙することで最小汎化集合を抽出する．段階的一般化法では，列挙木の枝刈りを行い，最小汎化集合を効率的に抽出することができる．しかしながら，ミスマッチクラスタを構成する部分文字列が増えると計算時間が膨大となる．

そこで，本論文では，段階的一般化法のマルチコア CPU 上での並列化手法を提案する．段階的一般化法の主な処理は列挙木の生成である．そこで，マルチコア CPU 上での列挙木の効率的な生成方法を提案する．提案手法の特徴は次の通りである．

- 分散型ワーカモデルを用い，ワーカをひとつのプロセスとせずにワーカをひとつのスレッドとして動作させる．また，タスクを効率的に管理するために，タスクプールを階層的に配置する階層的タスクプールを提案する．
- 段階的一般化法において生成される列挙木は各部分木の深さの偏りが大きい．そこで，2 種類の粒度のタスクを定義し，負荷の偏りの解消を極め細やかに行えるようにする．
- マルチコア CPU では，キャッシュとメモリ間のデータ転送量を減らすことが重要である．そこで，データ構造を工夫することでキャッシュとメモリ間のデータ転送量を削減する．

提案する並列化手法を用いて，段階的一般化法を実際に並列化し，*PROSITE*[2]から取得したデータセットに対して評価実験を行った．評価実験の結果として，理想的なスピードアップの実験結果が得られ，提案手法の有効性を示すことができた．

本論文の構成は以下の通りである．第 2 章で関連研究を述べ，第 3 章では段階的一般化法の説明や用語，記号の定義について説明する．第 4 章では提案手法，第 5 章では実験とその結果に対する評価を示す．第 6 章でまとめ，今後の課題について述べる．

II.　関連研究

段階的一般化法の主な処理は列挙木の生成であり，偏りが大きくなる可能性が高い．深さの偏りの大きな列挙木の並列生成手法として，計算機クラスタ上で頻出パターンを並列に抽出する並列 *Modified PrefixSpan* 法が提案されている[3]．頻出パターンを抽出する処理は列挙木の生成となる．提案されている手法では，マスタ・ワーカモデルを使用して，マスタとワーカに階層的にタスクプールを配置する方法を提案している．また，文献[3]で提案された手法は，同じように列挙木の生成となる分枝限定法の並列化にも適用されている[4].

また，シングルコアの計算機クラスタにおいて，分散型ワーカモデルが深さの偏りが極端に異なる列挙木の並列生成において効率的であることが示されている[5].

文献[6]~文献[8]では，メモリやキャッシュを意識することにより，効率的な手法を提案している．本研究では，データ構造を工夫することでキャッシュとメモリ間のデータ転送量を削減している．

本研究では，これらの研究と異なり，マルチコア CPU 上での深さの偏りの大きな列挙木の並列生成手法を提案している．文献[3]～文献[5]の手法を取り入れ，マルチコア CPU 上で，効率的に列挙木を生成する手法となっている．

III. 段階的一般化法

本章では，用語の定義，最小汎化集合の定義と段階的一般化法の処理手順を示す．

A. 用語の定義

1) 曖昧な問い合わせとミスマッチクラスタ

部分文字列Kと許容誤差r ($\geqq 0$)が，問合せQとして与えられられ，ハミング距離$d(K, K') \leq r$を満たす部分文字列K'が配列データベースDBからすべて選択されるとき，検索結果として得られる長さkの部分文字列<*inst*>の集合をミスマッチクラスタMISと呼ぶ．

2) 曖昧性を表現する汎化配列パターン

アルファベットΣの部分集合をΣ_iとするとき，k個のΣ_iを並べたパターンをk - 汎化配列パターンと呼び，$<pat^k>=<\Sigma_1\Sigma_2 \ldots\Sigma_{k-1}\Sigma_k>$と表記する．ただし，$\Sigma_i$は括弧[ ]の中に$\Sigma_i$の全要素を列挙した形で表記をする．例えば，2 - 汎化配列パターン<[AB][CD]>では，Σ_1={A,B}，Σ_2={C,D}となる．

3) インスタンスを導出する関数

k - 汎化配列パターンから k - インスタンスのすべて，すなわち長さ k の部分文字列の集合を導出する関数をEVAL(<pat^k>) と表記する．例えば，*EVAL*(<[AB][CD]>) = {<AC>,<AD>,<BC>,<BD>}である．2つの k-汎化配列パターン<pat_1^k>と<pat_2^k>について，*EVAL*(<pat_1^k>) $\supseteq$ *EVAL*(<pat_2^k>) が成立するとき，<pat_2^k>は<pat_1^k>に冗長であるという．

B. 最小汎化集合の定義

ある集合$MGS = \{G_1, G_2, \ldots, G_m\}$が，$k$ -汎化配列パターン<pat^k>およびk -インスタンス<$inst^k$>から構成されているとする(1 $\leqq m \leqq$ |*MIS*|)．ただし，EVAL(<pat^k>)$\subseteq MIS$ かつ<$inst^k$>$\in MIS$を満たすものとする．この集合 *MGS* が次の性質を満たすとき，*MGS*を*MIS* に対する最小汎化集合と呼ぶ．

(1) *EVAL*(*MGS*)＝*MIS* が成立する．
(2) *MGS* の任意の2要素G_i ,G_j に対して，G_iとG_j の間には冗長な関係が存在しない(1$\leqq i \leqq j \leqq m$).
(3) *MGS* に含まれるどの要素G_iも極大である(1$\leqq i \leqq m$)．すなわち，さらに汎化すると*MIS*に存在しないインスタンスを含んでしまうことになる．
(4) 上記の(1) ～ (3)を満たす任意の *MGS'*に対して，|*MGS'*|$\leqq$|*MGS*|が成立する．

C. 処理手順

以下に段階的一般化法の処理手順を示す．以下の手順(1)と(2)は，列挙木を深さ優先に生成するための変数 *Stack* と候補解を格納するための変数 *Candidate* について，初期化を行う処理である．手順(2)(a)は，親ノードから子ノードを列挙する処理である．手順(3)(b)①は列挙木ノードの最汎パターンを計算する処理であり，手順(3)(b)②は列挙木ノードの枝刈り処理である．手順(3)(c)は候補解集合の構成要素を収集する処理であり，手順(4)は候補集合から冗長な葉ノードを除去するための処理である．

(1) *Stack* ：＝ { {1}, {2}, ・ ・ ・ ,{n}}; ただし，{ *i* }$\in$*Stack* の *i* はミスマッチクラスタ *MIS* に存在する要素の識別番号とし，*Stack* の要素は辞書式順に取り出せるように管理する．
(2) *Candidate* := { }; ただし，*Candidate* を候補解を格納する領域とする．
(3) ***while*** (*Stack* $\neq \phi$)
  for each *S*$\in$ *Stack*
  (a) *Next* := *S* を親ノードとして列挙された子ノード *C* の集合;
  (b) ***for each*** *C*$\in$ *Next*
    ① <*pat*>:= *MGP*(*C* に対する *SubMIS*);
    ② ***if*** *EVAL*(<*pat*>)－*MIS* $\neq \phi$ ***then*** *C* を捨てる;
    else *Stack* : = *Stack* $\cup$ {*C*};
  (c) ***if*** *S* が葉ノードの条件を満たす ***then*** *Candidate* ： = *Candidate* ∪ *MGP*(*S* に対する *SubMIS*);
(4) *Candidate* から冗長な要素を除去;

IV. 提案手法

本章では，提案する段階的一般化法の並列化手法について説明する．

A. 並列化モデル

分散型ワーカモデルをマルチコア CPU 用に改良した並列化モデルを用いる．分散型ワーカモデルではすべてのワーカがひとつのプロセスであるが，マルチコア CPU 上では複数のプロセスに分ける必要がないため，すべてのワーカをひとつのスレッド（ワーカースレッドと呼ぶ）として配置する（図 1）．

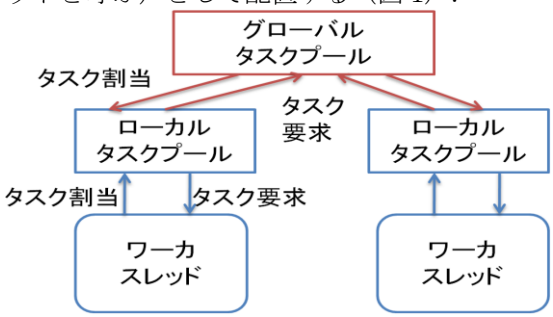

図 1：階層的タスクプール

また，タスクプールを階層的に配置する階層的タスクプールを提案する．階層的タスクプールでは，ワーカは 2 種類のタスクプールを持つ．1 つ目がグローバルタスクプール（以下 GTP），2 つ目がローカルタスクプール（以下 LTP）である．GTP はワーカで 1 つ設置する．LTP はワーカスレッドごとに 1 つ設置する．ワーカスレッドは，まずタスクを GTP から LTP に取り出す．次に，LTP からタスクを取り出し，タスクを実行する．タスクを実行することで生成されたタスクは LTP に戻す．ワーカスレッドは，タスクの実行を LTP にタスクがなくなるまで行う．LTP にタスクがなくなった場合のみ，GTP からタスクを取り出す．ワーカス

レッドごとに LTP を持つことで，タスク処理に関わる競合がワーカスレッド間で発生することを回避できる．

B. タスクモデル

段階的一般化法において，列挙木の部分木生成は他の部分木生成とは独立に行うことができる，そこで，部分木を同時に生成することで並列に列挙木を生成することが可能である．

しかし，列挙木の部分木の深さは偏りが大きいため，部分木生成をタスクとして定義して処理を行うと負荷の偏りを効率的に解消することができない．そこで，次に示すように異なる 2 つの粒度のタスクを定義する（図 2）．そのため，メインスレッドとワーカスレッドに別々の処理を行わせ，ワーカスレッドでタスク処理を効率的に並列化を行い，負荷の解消を実現した．

1) 部分木生成タスク

部分木生成タスクとは，列挙木のある親ノードから生成された子ノード集合から，子ノード集合の子孫ノードをすべて生成する部分木生成のことである．ただし，ルートノードを親ノードとする子集合ノードを対象とするタスクだけは，初期タスクとして子ノード集合の中の 1 つの子ノード以下の部分木を生成する部分木生成タスクとして定義する．

2) 子ノード生成タスク

子ノード生成タスクとは，列挙木のある親ノードから生成された子ノード集合から，その子ノード集合以降の子孫ノードを n 段下まで，ノードを生成する処理である．また，新たに生成された子ノードタスクとして，n 段下の子ノードを対象としたタスクがある．さらに，2 段下，3 段下，n 段下までノードを生成することができる．また，n はパラメータとして実行時に実行側が決めることができる．

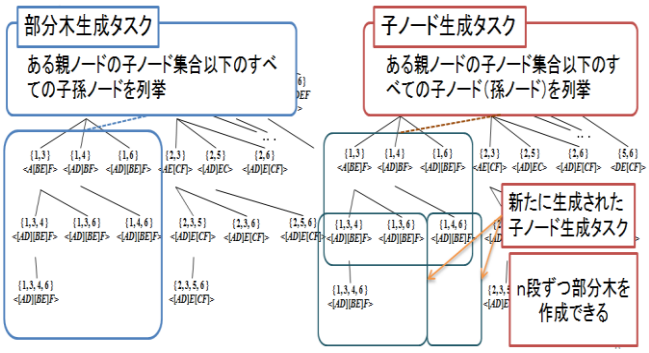

図 2 : タスクモデル

C. キャッシュコンシャスなデータ構造

キャッシュコンシャスとは，キャッシュの効果的な利用を考慮して効率の向上を目指すことである．キャッシュの動作を考えることは処理の高速化を考える上で重要な課題である．段階的一般化法の計算時間の短縮を図る上では，データ転送におけるデータ量を減らす必要がある．そこで，段階的一般化法では，列挙木のノードのデータ構造をビットマップで表現している．変更前では 1 回のデータ転送量が 8 ビットであったが，変更後では 1 回のデータ転送量を 32 ビットとした．

D. 処理手順

ここでは，メインスレッドとワーカスレッドの処理手順を示す．ワーカスレッドは，コア数に応じて起動する．ワーカスレッドを w とする．ここで，ワーカのグローバルタスクプールを *GTP*，ワーカスレッドのローカルタスクプールを *LTP* と表記する．ワーカは解の候補を格納するための集合として，*W_Candidate* とワーカスレッドが解の候補を格納するための集合として *WH_Candidate* を持つ．

メインスレッド

(1) 検索結果からミスマッチクラスタ *MIS* を生成する．
(2) 初期タスクを *GTP* に挿入する．
(3) ワーカスレッドを w 個起動する．
(4) w 個のワーカスレッドが終了するのを待つ．
(5) 各 *WH_Candidate* を *W_Candidate* にまとめる，
(6) *W_Candidate* から冗長な要素を除去する．
(7) 結果を出力する．

ワーカスレッド

(1) *GTP* から部分木生成タスクを取り出し，子ノード探索タスクとしてローカルタスクプールを *LTP* に挿入する．もし， *GTP* が空の場合，他の *LTP* からタスクを取り出す．
(2) ローカルタスクプールを *LTP* から子ノード生成タスク *T* を 1 つ取り出す．
(3) 子ノード生成タスク *T* に登録されている子ノードの集合を *TC* とする．各子ノード $S \in TC$ について次の処理を行う．
  (a) 子ノード *S* を親ノードとして，*S* のすべての子ノード *C* を列挙する．列挙した子ノードの集合を *Next* とする．生成された各子ノード $C \in Next$ について以下の処理を行う．
    (i) *C* に対応するミスマッチクラスタの部分集合 *SubMIS* の最汎パターン *MGP*(*SubMIS*)を求める．
    (ii) $EVAL(MGP(SubMIS)) - MIS \neq \phi$を満たすならば，*Next* から *C* を取り除く．
  (b) *Next* が空でなければ，*S* を親ノード，*Next* を子ノード集合とする子ノード生成タスク T_{new} を，ローカルタスクプールを *LTP* に挿入する．
  (c) 木の展開数 n に従い，(a)の親ノードの子ノード(最初の場合は，子ノード *C*)を親ノードとして，(*a*)，(*b*)の処理を $n-1$ 回行う．$n-1$ 回目の最後の処理において，*Next* が空ならば，*S* は解の候補であるため，*S* を *WT_Candidate* に挿入する．
(4) ローカルタスクプールを *LTP* が空でなければ(2)へ戻る．空ならば，(1)へ戻る．

V. 評価実験

評価実験では，*Kringle*，*Homeobox*，*PTS_EIIA*，*HTH_DEOR*，*HTH_ASNC* のデータセットを用い，提案手法を評価した．利用した計算機環境は，Intel(R)Core(TM)2 Quad CPU Q6700 @ 2.66GHz，メモリ：2.0GB，OS：ubuntu10.04 である．

（実験 1）スレッド数を 1～4 まで変更し，それぞれ 3 回ずつ実行することでスピードアップの評価した．結果として，どのデータセットもほぼ理想線形加速を得えることができた．図 3 に *HTH_ASNC* の実験結果を示す．

（実験 2）データ構造の変更前後で比較する．測定方法は（実験 1）と同じである．結果としては，計算時間が *Kringle* と *PTS_EIIA* は約 24%減，*Homeobox* と *HTH_DEOR* は約 9%減，*HTH_ASNC* は約 6%減した．表 1 に *HTH_ASNC* の実計算時間比較を示す．

（実験 3）データ構造の変更前後でキャッシュミスの回数を比較する．測定方法は（実験 1）と同じである．L1 でのキャッシュミス，L2 でのキャッシュミス，総サイクル数，待ちサイクル数を計測した．表 2 に変更前後での *L1* キャッシュミス回数と変更前後での *L2* キャッシュミス回数の比率を示す．図 4, 図 5 に変更前後の総サイクル数と待ちサイクル数を示す．

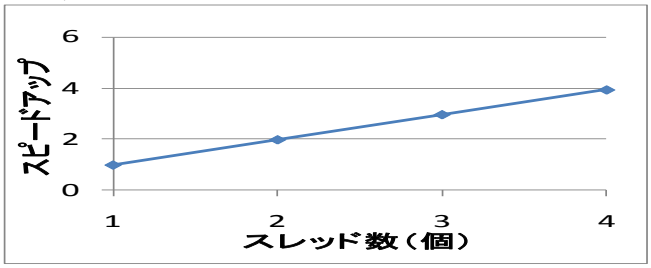

図 3:*HTH_ASNC* のスピードアップ

表 1: *HTH_ASNC* の実計算時間比較

（スレッド数）	変更前（秒）	変更後（秒）
1	1322.959	1246.789
2	664.832	627.121
3	445.614	419.770
4	336.517	316.559

表 2: 変更前後での *L1* キャッシュミス回数と *L2* キャッシュミス回数の比率（変更後÷変更前）

（データ）	L1 キャッシュミス回数比率	L2 キャッシュミス回数比率
Kringle	0.842	0.630
Homeobox	0.865	0.794
PTS_EIIA	0.871	0.836
HTH_DEOR	0.863	0.805
HTH_ASNC	0.862	0.961

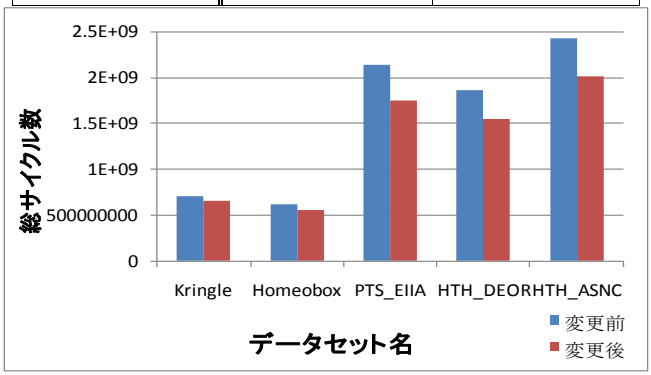

図 4: 各データセットの総サイクル数

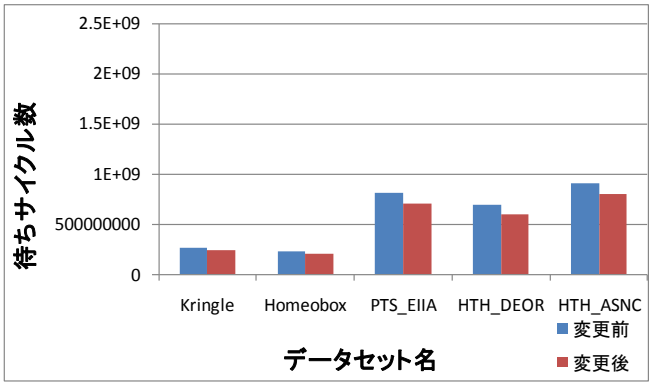

図 5: 各データセットの待ちサイクル数

VI. おわりに

本論文では，マルチコア CPU 上での段階的一般化法の並列処理を提案した．評価実験より，効率的に並列化が行えることが分かった．また，データ構造を工夫することでキャッシュミスを軽減することができることも確認できた．今後の課題として，キャッシュコンシャスなデータ構造を用いてメモリとキャッシュ間のデータ転送量をさらに減らすことが挙げられる．

謝辞

本研究の一部は，日本学術振興会・科学研究費補助金(基盤研究(C)，課題番号：20500137)，文部科学省・科学研究費補助金（若手研究(B)，課題番号：23700124）の支援により行われた．

参考文献

[1]田村慶一，木村浩明，荒木廉太郎，北上始：段階的一般化法によるミスマッチクラスタを表現する最小汎化集合の効率的抽出，電子情報通信学会論文誌 D , Vol. J93-D, No. 3, pp.189-202, 2010.

[2] http://kr.expasy.org/prosite/

[3]高木允，田村慶一，周藤俊秀，北上始：並列 modified prefixspan 法の並列化と動的負荷分散手法，情報処理学会論文誌　数理モデル化と応用，Vol. 46, No.SIG10, pp.138-152, 2005.

[4]田村慶一，岩木稔，高木允，北上始：PC クラスタにおける混合整数計画問題の並列処理とその性能評価．情報処理学会論文誌　数理モデル化と応用，Vol.46, No.SIG17, pp.56-69, 2005.

[5]Makoto Takaki, Keiichi Tamura, Toshihide Sutou, and Hajime Kitakami: A new dynamic load balancing technique for parallel modified prefixspan with distributed worker paradigm and its performance evaluation, In *Proceedings of ISHPC2005*, pp.227-237 , 2005.

[6] Shirish Tatikonda and Srinivasan Parthasarathy: Mining tree-structured data on multicore systems, *Proc*. VLDB *Endow*. ,Vol.2, No.1 , pp.694-705 , 2009.

[7] 高見澤　秀久，有次　正義：配列を用いたキャッシュコンシャスな索引木の提案，DBSJ Letters Vol.1, No.1 pp.11-14,2002

[8] Hidehisa Takamizawa, Kazuyuki Nakajima and Masayoshi Aritsugi: “Array-based Cache Conscious Trees”, *Information and Media Technologies*, Vol. 1, No. 2, pp.748-761, 2006

問い合わせ先

〒123-4567

広島市安佐南区大塚東 3 丁目 4 番 1 号

広島市立大学　情報科学研究科　知能工学専攻

八木　真平