2003－DBS－131　(43)
2003／7／17

FP-growth 並列化による頻出パターン抽出高速化

岩橋　永悟†　山名　早人‡

†早稲田大学大学院理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1
‡早稲田大学理工学部 〒169-8555 東京都新宿区大久保 3-4-1
E-mail: †eigo@yama.info.waseda.ac.jp, ‡yamana@waseda.jp

概要　データマイニング分野で重要な問題の一つに頻出パターン抽出問題がある。頻出パターン抽出手法では、多くの拡張手法を生んだ Apriori が有名である。2000 年になると Apriori よりも高速な手法として、FP-growth が提案されたが、従来の並列化手法の多くは、依然として Apriori に基づいている。本稿では、並列にディスクアクセスを行い、FP-tree をローカルに構築することによって、FP-growth を並列化する。本手法を 32 ノードクラスタ上で実験した結果、最小サポートを 0.25%とした場合に約 2 倍の速度向上を得ることができた。また、最小サポートを 2%とした場合、約 130 倍の速度向上を得ることができた。

キーワード　データマイニング, 頻出パターン, 並列処理, FP-growth, PC クラスタ

Parallel FP-growth Algorithm for Frequent Pattern Mining

Eigo IWAHASHI†　Hayato YAMANA‡

†Graduate School of Science and Engineering, Waseda University　3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
‡Faculty of Science and Engineering, Waseda University　3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
E-mail: †eigo@yama.info.waseda.ac.jp, ‡yamana@waseda.jp

Abstract

Frequent patterns mining is one of the important problem in data mining research. The Apriori is a prominent algorithm followed by many variants. In 2000, the FP-growth, which is reported to be faster than the Apriori, was proposed. However, many parallel algorithms of frequent pattern mining are still based on the Apriori. In this paper, we propose a parallelized version of the FP-growth, which accesses disks in parallel and constructs local FP-trees on each local memory. As a result of the evaluation using 32 node PC cluster, our method is approximately 2 and 130 times faster than sequential FP-growth, when minimum support is 0.25% and 2%, respectively.

Keyword　Data Mining, Frequent Pattern, Parallel Processing, FP-growth, PC Cluster

1. はじめに

近年、ネットワーク環境の整備、記憶装置の低価格化が進むにつれて、大量のデータを蓄積することが可能となった。しかし、データは記号の列にすぎず、データから情報を見出すのは本来ユーザの役割である。無秩序に集められた大規模なデータから、有用な知識を抽出するデータマイニング技術が注目されている。

データマイニング技術における重要な手法の一つとして、頻出パターン抽出がある。頻出パターンは、購買データから併売パターンを抽出するバスケット解析、Web ログからユーザの行動パターンを抽出する Web ログ解析、DNA 解析などに利用される。頻出パターンを抽出するためには、大規模なデータに対して処理を行うため、効率よく頻出パターンを抽出する手法の研究が行われている。

従来の頻出パターン抽出手法の多くは、Apriori アルゴリズム[1]をベースとする、候補パターンを生成する手法であった。近年では、Apriori とは異なるアプローチで頻出パターンを抽出する手法の研究も行われている。また、頻出パターン抽出処理は、並列性が高く、PC クラスタなどをターゲットとした、並列化手法の研究も行われている。しかし、従来の並列化手法は、なおも多くが Apriori に基づいている。そこで本稿では、Apriori に基づかない FP-growth アルゴリズム[2]の並列化手法を提案する。

FP-growth は、後に拡張手法[6]を生む重要なアルゴリズムであり、並列化を行えば、それらのアルゴリズムの高速化にも貢献できると考えられる。

本稿では、第 2 節で従来の頻出パターン抽出手法について、逐次アルゴリズムと並列アルゴリズムに分類して述べる。第 3 節では、FP-growth アルゴリズムの並列化手法について述べる。第 4 節では、第 3 節で述べた手法の性能評価について述べる。第 5 節で、まとめをおこなう。

2. 従来手法

第 2 節では、頻出パターン抽出問題について概要を述べ、従来行われてきた頻出パターン抽出手法について述べる。

2.1 頻出パターン抽出問題

頻出パターン抽出問題は、以下のように定義される。アイテムの集合を $I=\{i_1, i_2, \ldots, i_m\}$、トランザクションデータベースを $T=\{t_1, t_2, \ldots, t_n \mid t_i \subseteq I\}$ とする。T の各要素 t_i をトランザクションとする。アイテム集合 X のサポート support(X) は、T 全体に対して X を含むトランザクションの割合を表す。k 個のアイテムからなるアイテム集合を k-itemset とする。頻出パターンとは、ユーザが与えた最小サポートを満たすアイテム集合である。頻出パターン抽出問題は、ユーザが与えた最小サポートを満たす全ての頻出パターンを、T から抽出することである。

2.2 従来の頻出パターン抽出アルゴリズム

本節では、これまでに多くの手法で引用されている Apriori ベースのアルゴリズムについて述べる。その後、近年提案されている、トランザクションデータベースを別のデータ構造に変換し、頻出パターンを抽出する手法について述べる。

2.2.1 Apriori アルゴリズム

効率的に頻出パターンを抽出する手法として、1994 年に提案されたアルゴリズムが Apriori である[1]。Apriori は、多くの拡張手法のベースとなるアルゴリズムである。

Apriori は、発見された頻出アイテムセットから候補アイテムセットを生成し、数え上げを行う。Apriori は、1 回目のスキャンで頻出 1-item を抽出し、k 回目のスキャンで頻出 k-itemset を抽出する。頻出 k-itemset から候補(k+1)-itemset を生成し、数え上げを行う。

Apriori アルゴリズムの問題点は、候補パターンの数が莫大になるということである。また、候補パターンをチェックするために、データベースを繰り返しスキャンする必要がある。具体的には、データベース中の頻出パターンを構成する要素数の最大値が k であるとすると、k 回のスキャンが必要である。

2.2.2 Apriori 拡張手法

Apriori の拡張手法は、前節で述べた問題点を緩和することで効率化・高速化を図る。

1997 年に Park らによって提案された Dynamic Hashing Pruning(DHP)は、ハッシュ表を用いることによって、候補アイテムセット数を減少させる手法である[3]。DHP は、特に要素数 2 の候補アイテムセット数を減少させることができる。

1997 年に Stanford 大学の Brin らによって提案された Dynamic Itemset Counting(DIC)アルゴリズムは、Apriori アルゴリズムのスキャンコストを抑える手法である[4]。DIC は、データベースを等分割する。1 回目のデータスキャンにおいて、パーティション k を読んでいるあいだに、要素数 k のアイテムセットを数え始める。DIC は、パーティションごとでデータ内容に均一性がある場合に効果的である。

2.2.3 FP-growth アルゴリズム

2000 年に、Han らが提案した FP-growth アルゴリズムは、候補パターンを生成せずに、全ての頻出パターンを抽出する[2]。FP-growth は、FP-tree 構造と呼ばれる特殊なデータ構造を利用する。FP-tree 構造は、巨大なトランザクションデータベースがコンパクトに圧縮されたデータ構造であるため、スキャン回数を減らすことができる。

FP-tree 構造においては、頻出アイテムのみが頻出パターン抽出に使われる。頻出アイテムを発見するために、データベースを 1 回スキャンする必要がある。抽出された頻出アイテムをサポートの値により、頻度が降順になるように並び替える(このリストを F-list とする)。そして、空（null）のラベルを持つ木のルートを作る（この木を T とする)。頻出 1-item から FP-tree 構造を構築するために、2 回目のスキャンが行われる。2 回目のスキャンでは、具体的には、以下の手順で FP-tree 構造を構築する。各トランザクションに対して、

1. トランザクションから頻出アイテムを抽出し、F-list に従ってソーティングを行う。
2. T が F-list の要素である子を持っていれば、その子のカウントを 1 増やす。そうでないときは、新しくカウント 1 を持つ子を作る。
3. F-list の最後の要素まで 1、2 の操作を繰り返す。

全てのトランザクションで処理を終えたら、同じ名前（アイテム ID）を持つノードにリンクを付ける。さらに、各頻出アイテム a_i に対して、先頭の a_i を指すヘッダテーブルを作成する。

表 1 のようなトランザクションデータベースから、FP-tree 構造を構築する例を示す。ここで、最小サポート値を 3 とする。

表 1：トランザクションデータの例

TID	Items	Frequent Items
100	f, a, c, d, g, I, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

1 回目のスキャンでは、頻出 1-item が抽出され、F-list<(f:4), (c:4), (a:3), (b:3), (m:3), (p:3)>が生成される。続いて 2 回目のスキャンを行う。各トランザクション中の頻出アイテムは、F-list の順序に従って並べ替えられる。最初のトランザクションから、最初の枝である<(f:1), (c:1), (a:1), (m:1), (p:1)>を得ることができる。2 番目のトランザクションから、<f, c, a, b, m>という枝が得られる。この枝は、最初に作られた枝<f, c, a, m, p>と<f, c, a>を共有するので、prefix 部分のノードのサポートがインクリメントされる。そして、新しいノード(b:1)が生成され、(a:2)の子としてリンクされる。もうひとつの新しいノード(m:1)が、(b:1)の子としてリンクされる。以降のトランザクションに対しても、同様の処理を繰り返して、図 1 に示す FP-tree 構造が構築される。

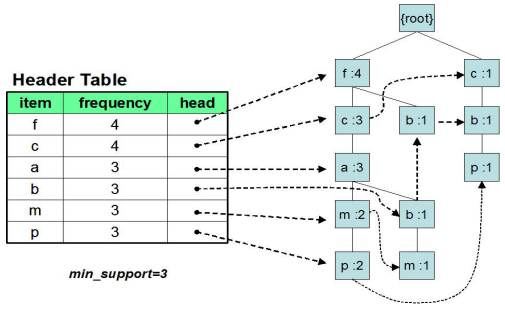

図 1: FP-tree 構造の例（文献[2]より）

FP-growth アルゴリズムは、FP-tree 構造の以下の 2 つの特徴を利用する。1 つ目は、どんな頻出アイテム a_i に対しても、先頭の a_i を示すヘッダテーブルから、a_i のノードリンクをたどることにより、a_i を含む生成可能な頻出パターンを全て得ることができるということである。2 つ目は、パス P にあるノード a_i を含む頻出パターンを数えるためには、パス P におけるノード a_i の prefix-path を求めるだけでよく、prefix-path にあるノードのカウントは、ノード a_i のカウントと等しいということである。

2.2.4 Pattern Decomposition

2001 年に Zou らによって提案された Pattern Decomposition(PD)は、FP-growth アルゴリズムと同様に、データベースを別のデータ構造に変換し、頻出パターンを抽出する手法である[5]。FP-growth と異なる点は、PD は新しいデータ構造を前もって構築しないという点である。その代わりに、パスを経るごとに、非頻出アイテムセットを用いて、データセットを分解しサイズを小さくする。

PD は、頻出パターンを発見するためにボトムアップ探索を行う。データセット D_1 に対して、パス 1 から開始する。パス k において、長さ k の頻出アイテムセットが抽出される。以下に、パス k における処理を示す。

1.D_k にある全ての k-itemset を数え、頻出アイテムセット L_k と非頻出アイテムセット～L_k を生成する。
2.～L_k にあるアイテムセットを含まない D_{k+1} を得るために、D_k を分解する

ここで、アイテムセット I の分解(decomposition)とは、～L_k に含まれる非頻出アイテムセットを含まない、I の最も大きい部分集合 S を見つけることである。つまり、S に含まれる k-itemset はすべて L_k で頻出である。

2.2.5 逐次アルゴリズムのまとめ

従来の多くの頻出パターン抽出手法は、Apriori をベースとして、改良を加えるものであった。FP-growth が提案された 2000 年以降は、トランザクションデータを、独自のデータ構造に変換することで、効率よく頻出パターンを抽出する手法が提案されている。本稿で紹介した FP-growth や、PD 以外にも、H-Mine[6] や Oppotunistic Projection(OP)[7]といった手法が提案されている。H-Mine は、密なデータセットに対しては、FP-growth を適用する。図 2 に頻出パターン抽出手法の変遷を示す。

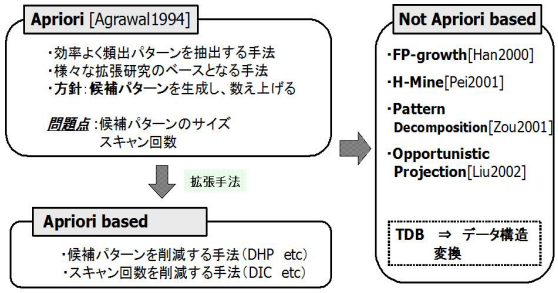

図 2: 頻出パターン抽出手法の変遷

2.3 従来の並列化アルゴリズム

頻出パターン抽出を高速化するために、並列化の研究が行われている。以下では PC クラスタなどの分散メモリ型並列計算機をターゲットとした並列化手法について述べる。

2.3.1 Count Distribution

Apriori の最も簡単な並列化手法は、1996 年に Agrawl らによって提案された Count Distribution(CD)である[8]。CD では、トランザクションデータベースを複数のノードに分配し、各ノードで独立に候補アイテムセットの出現頻度を求める。Apriori アルゴリズムの k 回目のパ

スが以下のように並列化される。

1.要素数 k の候補アイテムセットを全ノードに送信する。
2.各ノードで、割り当てられたトランザクションデータをスキャンして、候補アイテムセットの出現頻度を数える。
3.1 つのノードで 2.の結果を候補アイテムセットごとにマージして、データベース全体での出現頻度（グローバルサポート）を求める。
4.同じノードで要素数 k+1 の候補アイテムセットを生成する。

以上の 1～4 の処理を候補アイテムセットがなくなるまで繰り返す。

CD ではローカルディスク中の候補アイテムを数えている間は、各ノードが独立して動作できるので、候補アイテムセットが 1 つの主記憶に収まる場合は、高い台数効果が期待できる。

2.3.2 Hash Partitioned Apriori

CD は候補アイテムセットを全ノードに複製するため、候補アイテムセットが各ノードの主記憶に収まらない場合、並列に処理を行っても高速化が達成できなかった。1996 年に東京大学の Shintani らによって提案された Hash Prtitioned Apriori(HPA)は、ハッシュ関数を用いて候補アイテムセットをノードごとに分割し、ノード全体での記憶効率を高めることができる手法である[9]。HPA の手順は以下のとおりである。

1.要素数 k の各候補アイテムセットをハッシュ関数によって決定されるノードに送信する
2.各ノードで、割り当てられたデータベースの各トランザクション t に対して
a) t から要素数 k の部分集合をすべて作り、各々について 1.のハッシュ関数を適用して決定されるノードに送る。
b) 要素数 k のトランザクションを受信したノードは、それと一致する候補アイテムセットを探し、その出現数を 1 増やす。
3.各ノードが、候補アイテムセットが頻出か否かを決定し、結果を 1 つのノードに集める。
4.結果が集められたノードで、要素数 k+1 の候補アイテムセットを生成する。

HPA では記憶効率を高めることができる代わりに、ノード間の通信やハッシュ関数の計算を繰り返す必要がある。この問題を緩和するために提案されたのが HPA-ELD(HPA with Extremely Large itemset Duplication)アルゴリズムである[9]。HPA-ELD では頻度の高い候補アイテムセットを各ノードで複製して保持し、複製された候補は CD と同様に処理される。この方法によって、アイテムセットの通信量を減らすことができる。

3. FP-growth 並列化手法

第 2 節で述べたように、従来の頻出パターン抽出並列アルゴリズムの多くは、Apriori を並列化した手法であった。本論文では、Apriori をベースとしない FP-growth アルゴリズムを並列化することで、頻出パターン抽出の高速化を図る。

3.1 FP-growth 並列化のポイント

FP-growth アルゴリズムは、大きく分けて以下の 2 ステップに分けることができる。

STEP1 FP-tree 構造の構築

STEP2 FP-tree 構造に対するマイニング

ここで、並列化を行っていない FP-growth を実行したときに、STEP1 と STEP2 を処理するために必要な時間を図 3 に示す。また、FP-growth 総実行時間に対する、STEP1 処理時間と STEP2 処理時間の割合を図に示す。以上の実行時間は、データセット T25I15D100k を入力データとして、最小サポートを変化させて計測した。

図 3 を見ると最小サポートが 0.4%以上の場合、FP-tree 構築時間が FP-growth の総実行時間の 80%以上を占めていることがわかる。つまり、最小サポートが 0.4%以上の場合は、FP-tree 構築を高速化すれば総実行時間を大幅に短縮できると考えられる。

いっぽう、最小サポートが 0.4%未満の場合は、総実行時間に対する FP-growth 時間の割合が増えてくるため、FP-tree 構築の高速化とともに木の走査を高速化することが必要となる。

上記の点を踏まえると、FP-growth を高速化するためには、FP-tree 構築実行時間（STEP1 実行時間）を短縮することが主要課題となる。

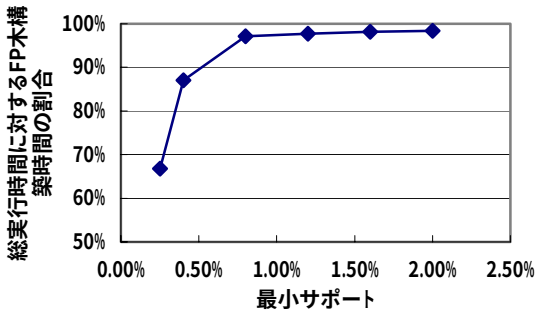

図 3: 最小サポートと FP-tree 構築時間の関係

3.2 FP-tree 構築の並列化

FP-tree 構造は、トランザクションデータ内の頻出パターン抽出に必要な情報のみを圧縮したコンパクトなデータ構造である。つまり、FP-tree を構築するためには、トランザクションデータに対してスキャンを行う必要がある。並列化に

よって得られる大きな利点の一つとして考えられるのが、トランザクションデータに対するスキャンを並列化できることである。例えば、p 台のノードのディスクにデータが格納されていれば、シーケンシャルにデータを読む場合に比べて、単位時間あたり p 倍のデータ量を読むことができる。この利点を生かして、FP-tree 構築を並列化する。各ノードはローカルディスクから、ローカル FP-tree を構築する（図 4)。

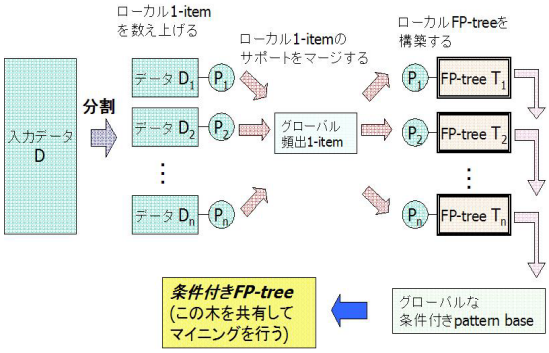

図 4: FP-tree 構築の並列化

具体的には以下の手順をとる。CD と同様に、トランザクションデータを先頭から均等に各ノードに分配しておく。

1.各ノード P_n がローカルディスク D_n からアイテムのサポートを求める。
2.データベース全体でのサポートを得るためにローカルサポートをマージする。
3. 各ノード P_n がローカルな FP-tree である T_n を構築する

以降で、各手順をさらに詳細に説明する。

3.2.1 ローカルアイテムの数え上げ

1 度目のスキャンでは頻出 1-アイテムを数える。効率よく頻出アイテムを列挙するために、入力データを各ノードにあらかじめ分配しておく。各ノードには同数のトランザクションが分配される。各ノードは、トランザクションに現れるローカルなアイテムを数える。ローカルなアイテムをカウントしている間、各ノードは独立して動作する。

3.2.2 グローバルサポートの数え上げ

各ノードにおいてローカルなアイテムのカウントが終了したら、入力データ全体でのサポートを得るために、ローカルサポートをマージする。マージにあたっては、各ノードが他のノードにローカルサポートを送信することによって、全てのノードでグローバルサポートが数えられ、頻出でないアイテムが除かれる。見つかった頻出アイテムは、サポートが降順に並ぶようにソートされる。このリストが FP-growth における F-list である。

3.2.3 ローカル FP-tree 構築

ローカル FP-tree を構築するために、2 度目のデータベーススキャンを行う。各ノードはローカルなトランザクションからローカル FP-tree を構築する。各トランザクションに対して F-list にある頻出アイテムだけが集められる。F-list の順序にしたがって各アイテムの頻度が降順に並ぶようにソートされる。

各トランザクション内でソートされたアイテムは、ローカル FP-tree を構築するために以下のように使われる。

1.トランザクションに出現するアイテムに対して、root に子ノードが存在するかどうかをチェックする。
2. 子ノードが存在すれば、子ノードのサポートをインクリメントする。
　子ノードが存在しなければ、出現したアイテムのために、新しいノードを追加し、サポートを 1 とする。
3.読み込んだアイテムを新しい root として、トランザクションに出現する次のアイテムに対しても手順 1.を繰り返す。

以上の手順で、各ノードのローカルディスクからローカル FP-tree を構築する。ヘッダテーブルには、データベース全体で頻出するアイテムと、そのサポートが格納されている。また、ヘッダテーブルは、各ノードのローカル FP-tree で最初に出現する各アイテムへのリンクを持つ。

3.2.4 FP-growth 並列化

STEP2 では STEP1 で構築したローカル FP-tree に対して、マイニングを行う。各ノードがローカル FP-tree から、条件付き pattern base を生成する。条件付き pattern base とは、あるアイテム集合 X よりも前に現れたアイテムのリストである。各ノードが、X についての条件付き pattern base の対応するアイテムをマージし、同一アイテムのグローバルなサポートを求める。X の条件付き pattern base のサポートが最小サポートより大きければ、条件付き FP-tree にアイテムを追加する。生成された条件付き FP-tree は、全ノードで共有され、この木に対してマイニングを行う。

4. 性能評価

第 3 節で述べた手法を 32 ノード PC クラスタ上で実装した。各ノードは、Intel Pentium4 プロセッサ 2.40GHz と、1GB(512MB×2)のメモリを持つ。また、各ノードは、1000Mbps イーサネットで接続されている。並列アルゴリズムの実装には、フリーな通信ライブラリである MPICH(Version 1.2.5)を用いた。また、性能を評価するために、IBM 人工データセット生成プログラム[10]を用いた。このプログラムは、入力パラメータによって性質の異なるデータセットを生成する。今回の実験では、T25I15D100k としてデータセットを生成した。

ノード台数を 1 台から 32 台まで変化させて、最小サポート値 2.00%、1.60%、1.20%、0.80%、0.40%、0.25%の場合について実験を行った。ノード台数が 1 台の場合は、並列化を施していないプログラムを実行し、実行時間を測定した。実験結果を、PU 台数と速度向上率の関係でグラフに表したのが図 5、図 6 である。また、最小サポートと実行時間の関係でグラフに表したのが図 7 である。

図 5 を見ると、最小サポートが 2.00%の場合、ノードを 32 台投入すると約 130 倍の速度向上を得ることができている。投入したプロセッサ台数以上の台数効果が得られている要因は、ディスクアクセスを並列化したことによって、ボトルネックであるディスクアクセスにかかる時間が短縮されたためであると考えられる。

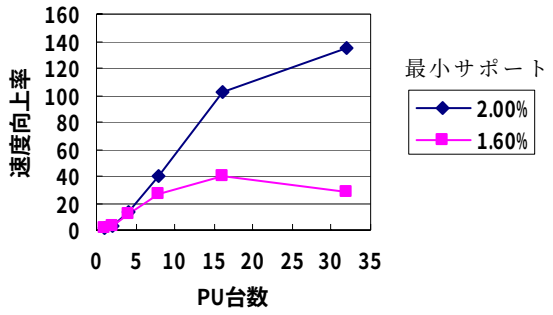

図 5: PU 台数と速度向上率の関係

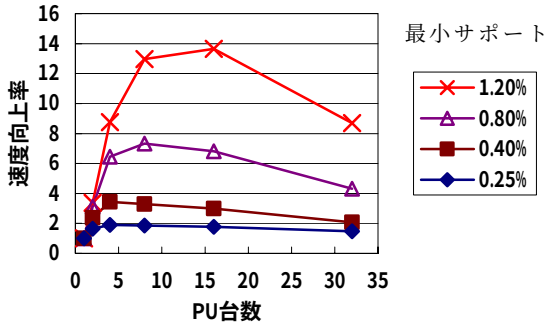

図 6: PU 台数と速度向上率の関係

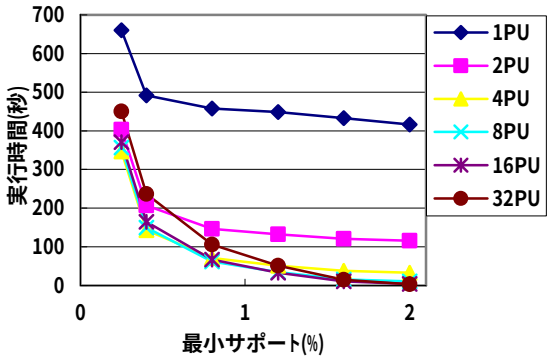

図 7: 最小サポートと実行時間の関係

図 7 において、ノード台数を固定して見ると、最小サポートが小さくなるにつれて、台数効果が小さくなっている。最小サポートが 0.25%の場合、得られた速度向上は約 2 倍（4 ノード使用時）だった。これは 3.1 で述べたように、最小サポートが大きいほど、FP-tree 構築に占める時間的割合が大きく、FP-tree 構築の並列化が効果的に働くためである。もう一つの原因は、最小サポートが小さければ小さいほど、頻出アイテム総数が多くなり、FP-growth（STEP2）実行時の通信コストが高くなるためであると考えられる。

以上の結果により、ディスクアクセスを並列化することによる、FP-tree 構築部分の並列化が有効であることがわかった。FP-tree 構築部分を並列化することによって、頻出アイテムセットを抽出する時間を短縮することができた。

5. おわりに

データマイニング分野の重要な問題として、頻出パターン抽出がある。頻出パターン抽出は、バスケット解析や DNA 解析などに適用されるが、大規模なデータを対象に、処理を行うため時間が掛かるという問題がある。従来の頻出パターン抽出を高速化する研究は、多くが Apriori に基づいていた。本稿では、Apriori に基づかない FP-growth の並列化を提案した。ディスクアクセスを並列化し、FP-tree をローカルに構築することで並列化を実装した。最小サポートを 0.25%として 32 ノードクラスタで実験を行った結果、約 2 倍の速度向上を得ることができた。また、最小サポートを 0.25%として同じ実験を行った場合は、約 130 倍の速度向上を得ることができた。

今後の課題としては、最小サポートが小さい場合に、並列処理効率が低下するという問題を解決するアルゴリズムの検討があげられる。また、Apriori アルゴリズムをベースとする CD や HPA といった従来の並列化手法と比較を行う必要がある。

参 考 文 献

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," In Proceedings of the International Conference on Very Large Data Bases, pp. 487-499, 1994.

[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," In Proceedings of the ACM SIGMOD Conference

on Management of Data, pp.1-12, 2000.

[3] J.S. Park, M. Chen, and P.S. Yu, "An effective hash-based algorithm for mining association rules," In Proceedings of the ACM SIGMOD Conference on Management of Data, pp.175-186, 1995.

[4] S. Brin, R. Motowani, J. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," In Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 255-264, 1997.

[5] Q. Zou, W. Chu, D. Johnson, and H. Chiu, "A Pattern Decomposition (PD) Algorithm for Finding All Frequent Patterns in Large Datasets," In Proceedings of the 2001 IEEE International Conference on Data Mining(ICDM'01), pp.673-674, 2001.

[6] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases," In Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01), pp.441-448, 2001.

[7] J. Liu, Y. Pan, K. Wang, and J. Han, " Mining Frequent Item Sets by Opportunistic Projection," Proc. 2002 Int. Conf. on Knowledge Discovery in Databases (KDD'02), 2002.

[8] R. Agrawal and R. Srikant, "Parallel mining of association rules," IEEE Transactions on Knowledge and Data Engineering, 8(6), 1996.

[9] T. Shintani and M. Kitsuregawa, "Hash based parallel algorithms for mining association rules," In Proceeding International Conference on Parallel and Distributed Information Systems, pp.19-30, 1996.

[10] IBM Quest Data Mining Project.
Quest synthetic data generation code.
http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html