# FP-growth の無共有並列実行

## Shared Nothing Parallel Execution of FP-growth

イコ プラムディオノ♦ 喜連川 優♠

Pramudiono IKO Masaru KITSUREGAWA

**FP-growth という頻出パターンマイニングアルゴリズムが注目されている．FP-tree と呼ばれるそのメタデータによってそれまで発表されたアルゴリズムよりはるかに高速であることが示された．しかしそのような特殊なデータ構造の並列実行は、PC クラスタのような無共有型並列計算機では、特に困難である．本論文では、十分な台数効果を得るために、「パス深さ」と呼ばれるパラメータを活用して、実行ノード間の負荷を均等化するメカニズムを報告する．**

**FP-growth has become a popular algorithm to mine frequent patterns. Its metadata FP-tree has allowed significant performance improvement over previously reported algorithms. However, the parallelization of such special data structure is difficult, in particular for shared-nothing parallel machines such as PC cluster. Here we report how we utilize a characteristic called "path depth" to balance the workload among processing nodes to obtain sufficient parallel speedup ratio.**

## 1. はじめに

頻出パターンマイニングは現在、重要なマイニング技術の一つとして確立されてきた．さらに、頻出パターンマイニングは相関ルールマイニングや時系列パターンマイニングといった他の重要なマイニング技術の基礎をなしている．

FP-growthは頻出パターンマイニングアルゴリズムの新たな世代を切り開いた[4]．トランザクションデータベースをFP-treeというメモリ上データ構造に圧縮することで、Apriorioのような従来アルゴリズムよりはるかに高い性能が達成できた[2]．

さらなる高性能なマイニング処理は並列化により期待できる．並列エンジンは現在の大規模データウェアハウスにとって不可欠になりつつあり、PCクラスタのような高コストパフォーマンスな無共有型計算機が注目を集めている．本論文では、そのような環境上の並列アルゴリズム開発のために、並列化のボットルネック及びその対策を論ずる．

本論文の構成は第２章で関連研究を紹介し、第３章ではベースとなるFP-growthを簡略に説明する．第４章では、我々が提案する無共有型計算機上のFP-growth並列実行の手法を議論する．特に並列実行粒度の制御のために、「パス深さ」という新たなメトリクスの活用を検討する．第５章では、PCクラスタ上で我々の手法の実装及び性能評価を報告する．

♦ 学生会員　東京大学大学院工学系研究科博士課程
iko@tkl.iis.u-tokyo.ac.jp
♠ 正会員　東京大学生産技術研究所
kitsure@tkl.iis.u-tokyo.ac.jp

## 2. 関連研究

Apriorioは頻出パターンマイニングを取り上げる最初のアルゴリズムである[2]．その後、いくつかのAprioriを改良したアルゴリズムが開発された．

頻出パターンの並列実行に関する研究は[3,6]によって開拓された．Hash Partitioned Apriori(HPA)は無共有型計算機上のメモリ利用効率を図る[7]．さらに、一般化された相関ルールマイニングにも拡張された[8]．

Apriorioのような「候補生成とテスト」パラダイムに対して、TreeProjectionなどのアルゴリズムが提案されてきたが[1]、次世代の頻出パターンマイニングアルゴリズムの幕開けはFP-growthの登場を待たなければならなかった[4]．

その後、FP-growthの性能が劣化するスパースなデータセットに対して、H-mineが発表された[5]．データの特性に合わせて、処理途中でH-mineがFP-growthに切り替えることも可能である．

## 3. FP-growth

FP-growthの実行は二つのフェーズに分けることができる：FP-treeの構築とそのFP-treeからすべての頻出パターンを抽出する[4]．

### 3.1 FP-tree の構築

FP-treeは頻出アイテムだけが含まれる一種のtrieであり、頻出なアイテムが共有されやすいように、頻度順に並べる．頻出パターンを求めるのに、必要な情報がFP-treeにすべて含まれている．

FP-treeの構築はトランザクションデータベースを２回スキャンする必要がある．一回目のスキャンでは、頻出アイテム集合Fとそのサポートを求める．Fをサポートの降順にソートし、F-listと呼ぶことにする．二回目のスキャンで、FP-treeを作る．FP-treeのルートのラベルはnullである．それぞれのノードには（アイテム名、カウント、次のリンク）情報が保持される．トランザクション内のアイテムに対応するノードがあれば、そのノードのカウントを増加させるが、対応ノードがなければ、新たなノードが作成される．

ヘッダテーブルは（アイテム名、ノードリンク）を持っており、そこから同じラベルのノード同士をたどることができる．

FP-tree 構築の例として図１の Node1 と Node2 の SEND プロセスがローカルデータベースからそれぞれの FP-tree を構築する．

### 3.2 FP-growth

FP-growthの入力はFP-treeと最小サポート値である．すべての頻出パターンを求めるのに、FP-tree内のノードをヘッダテーブルの末尾のノードリンクからたどる．ヘッダテーブル末尾のアイテムがmだとすると、ノードリンクはFP-tree内のmに対応するノード同士をつなげており、ノードリンク上のノードからルーツまでのパスからなる接頭パスをm条件つきパターンベース（conditional pattern base、CPB）に集めることで、mを含むすべてのアイテム集合をFP-treeから抽出できる．つまり、m条件つきパターンベースはmと共起するすべてのアイテム集合の部分データベースとなっている．

その条件つきパターンベースから条件つきFP-treeを構築し、新たな条件つきパターンベースが生成できないまでFP-growthが再帰的に繰り返され,パターン片を成長させていく．

ヘッダテーブルの次のアイテム処理では、もはやmを考慮する必要がなく、分割統治的な処理で効率化を図る.

## 4. FP-growth の並列実行

条件つきパターンベースの処理は他のアイテムの条件つきパターンベース処理とは独立に行えるので、並列実行の単位として考えるのが自然である.

ここでは、まず単純なバージョンの並列化について説明する.

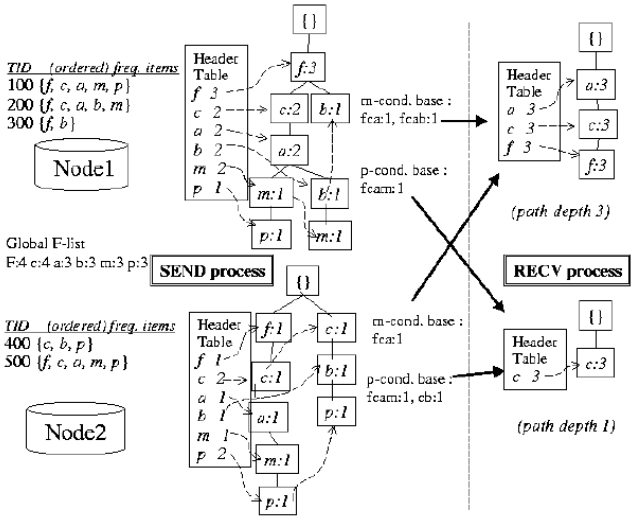


図 1 FP-growth 単純並列化の例

Fig.1 Illustration of FP-growth Trivial Parallel Execution

```
input : database D, items I,
        minimum support min_supp;

SEND process :
{
1:local_support = get_support(D,I);
2:global_support = exchange_support(local_support);
3:FList = create_flist(global_support, min_supp);
4:FPtree = construct_fptree(D, FList);

;exchange conditional pattern base
5:forall item in FList do begin
6:    cond_pbase = build_cond_pbase(FPTree, item);
7:    dest_node = item mod num_nodes;
8:    send_cond_pbase(dest_node, cond_pbase);
9:end
}

RECV process :
{
1:cond_pbase = collect_cond_pbase();
2:cond_FPtree = construct_fptree(cond_pbase, FList);

3:FP-growth(cond_FPtree, NULL);
}
```

図 2 FP-growth 単純並列化の擬似コード

Fig.2 Pseudo code of FP-growth Trivial Parallel Execution

### 4.1 単純並列化

基本的な考え方はそれぞれのノードが完全な条件つきパターンベースを受けてから、独立にその条件つきパターンベースの処理を完成させる.

単純並列化の例は図１に、その擬似コードが図２に示される.

SENDプロセスとRECVプロセスという二つのプロセスが必要である. 一回目のデータベーススキャンの後、各ノードのSENDプロセスがすべてのアイテムのカウントをお互いに交換し、全体の頻出アイテムを決定する. 交換後、各ノードのSENDプロセスが全アイテムのカウントを保有するので、グローバルのF-listを生成できる. すべてのノード同一のF-listを保持することになる. 二回目のデータベーススキャンで、各々のSENDプロセスがノードのローカルデータベースからローカルFP-treeを構築する.

ローカルのFP-treeより条件つきパターンベースを生成するが、自ら処理するより、SENDプロセスがハッシュ関数を用いて、条件つきパターンベースを他のノードのRECVプロセスに送る. RECVプロセスがすべてのノードのSENDプロセスから条件つきパターンベースを収集し、グローバル条件つきパターンベースを再構築する. グローバル条件つきパターンベースを再現したRECVプロセスは独自にFP-growthを実行する.

### 4.2 パス深さを用いる粒度均等化

効率的な並列化を得るために、並列化の粒度が重要な課題である. 並列実行粒度とは全体プログラムに対する並列化された計算量の割合で表すことができる.

上述のFP-growth単純並列化は条件つきパターンベースのランダム的な配布を採用しているが、それぞれの条件つきパターンベースの処理時間には大きなばらつきが生じうる.

並列実行の単位として、条件つきパターンベースを用いる場合、並列実行粒度は次の条件つきパターンベースを生成する繰り返し回数によって決まる. その回数は条件つきパターンベース内の最も長い頻出パターンの長さに指数的に比例する. そのため、粒度の目安として「パス深さ」を定義する.

**パス深さの定義:** 条件つきパターンベース内の最小サポート値を満たす最長のパターンの長さである.

パス深さの例：図１では、mの条件つきパターンベース内で最も長いパターンは<acf>のため、そのパス深さは３である.

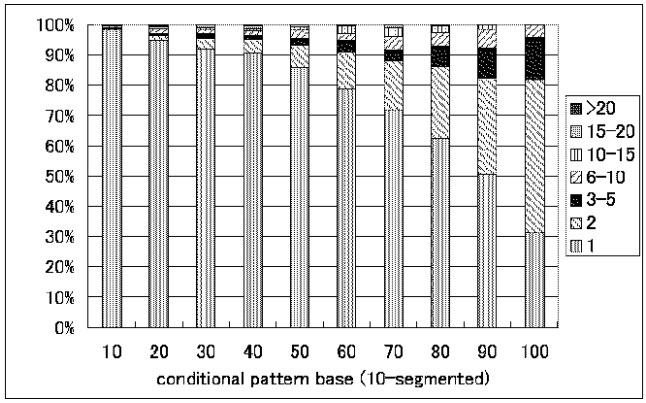


図 3 パス深さの分布

Fig.3 Distribution of Path Depth

パス深さの典型的な分布は図３に示される. F-list内のア

イテムを１０つのセグメントに分割した時に、それぞれのアイテムに対して生成される条件つきパターンベースのパス深さの割合を表している．

粒度が大きく異なるため、小さい粒度を割り当てられたノードが大きい粒度を実行するノードの終了を待たなければならず、スケラビリティーを損なってしまう．図５の左側では、粒度均等化なしの単純並列化の実行トレースが示される．赤い線が各ノードのCPU占有率を表す．そこで、ノード１の終了を待っている他のノードの様子が確認できる．

並列実行の効率化のために、大きい粒度を有する並列タスクを分割する必要がある．パス深さがFP-treeを構築する時、あらかじめ計算できるので、パス深さを用いることで、その並列タスクの分割を予測できるのである．

ここでは、条件つきパターンベースより条件つきFP-treeを構築してから、さらに小さい条件つきパターンベースを生成するFP-growthの再帰的な性格を利用する．より小さい条件つきパターンベースが生成される時、パス深さが一つ減らされる．

```
SEND process :
{
1:cond_pbase = get_stored_cond_pbase();
2:if(cond_pbase is not NULL) then
3:   send_cond_pbase(cond_pbase);
4:end if
}

RECV process :
{
1:cond_pbase = get_stored_cond_pbase();
2:if(cond_pbase is NULL) then
3:   cond_pbase = receive_cond_pbase();
4:end if
5:cond_FPtree = construct_fptree(cond_pbase, FList);
6:FP-growth(cond_FPtree, cond_pbase.itemset);
}

procedure FP-growth(FPtree, X);
input : FP-tree Tree, itemset X;
{
1:for each item y in the header of Tree do {
2:   generate pattern Y = y U X with
       support = y.support;
3:   cond_pbase = construct_cond_pbase(Tree, y);
4:  if (cond_pbase.path_depth < min_path_depth) then
5:      Y-Tree = construct_fptree(cond_pbase,Y-FList);
6:      if (Y-Tree is not NULL) then
7:         FP-growth(Y-Tree, Y);
8:      end if
9:   else
10:      store_cond_pbase(cond_pbase, Y);
11:  end if
12:end for
}
```

**図４　パス深さを用いた粒度均等化の擬似コード**

**Fig.4　Pseudo code of granularity balancing with path depth**

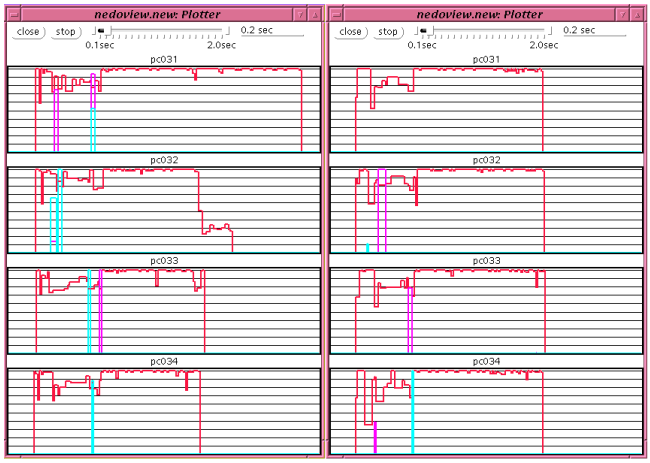


**図５　実行トレース：(左) 単純並列化（右）パス深さを用いる粒度均等化**

**Fig.5　Execution Trace: (left) trivial parallelization (right)granularity balancing with path depth**

最小パス深さを指定することで、並列実行粒度を制御することができる．パス深さを利用する粒度均等化メカニズムの擬似コードを図４に示す．最小パス深さ以下のパス深さを持つ条件つきパターンベースは割り当てられた実行ノードで終了までそのまま実行される．それ以外の場合、次の条件つきパターンベース生成までは実行されるが、生成された条件つきパターンベースは保存される．その一部が同じノードによって実行されるか、他のアイドル状態の実行ノードに送られ、そこで実行される．

## 5. 実装と性能評価

提案した方式の性能を評価するために、ＰＣクラスタ上で実装を行った．実行環境として、100Base-TX Ethernet Switchに相互接続される３２ノードからなるＰＣクラスタである．各々のノードはPentiumIII 800Mhz及び128MBメモリを搭載している．

### 5.1　実装

各ノード上では、三つのプロセスがある：

１．SENDプロセス

FP-treeの構築、条件つきパターンベースの送信

２．RECVプロセス

条件つきパターンベースの受信、受信終了後後条件つきパターンベース処理の実行

３．EXECプロセス

条件つきパターンベース受信中、背後で条件つきパターンベースの実行

その他に、アイドル状態の実行ノードから条件つきパターンベースの要求を受け、それを振り分けるCOORDと呼ばれる軽量プロセスもある．

### 5.2　性能評価

性能評価のために、Apriori論文で紹介された方法で生成された人工データセットを利用する[2]．データ生成のパラメータは：トランザクション数１０万、アイテム数１万、平均トランザクションサイズ２５、平均頻出アイテム集合の長さ２０に設定されている．

まず実行時間に関して、パス深さの影響を調べるために、

いくつかの最小パス深さ設定を変えて実行時間を図６に示す．横軸として実行ノード数を１，２，４，８，１６，３２に取っている．最小サポート値は０．１%に設定されている．最速実行は最小パス深さ１２と３２ノードの時で40秒を記録していることがわかる．1ノードの時の実行時間は平均904秒である．

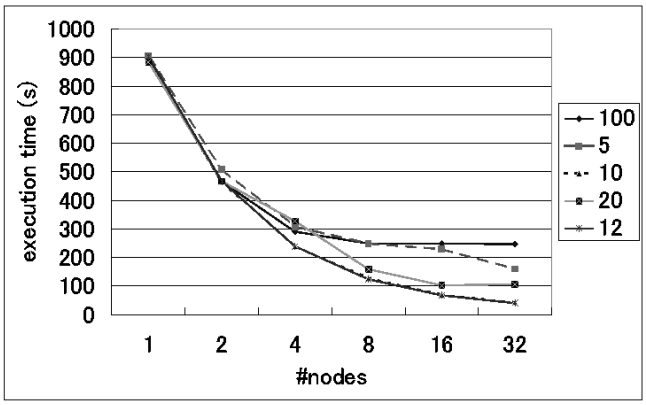


図６　実行時間

Fig.6　Execution Time

次に並列化の効率を示す指標として、ノードを増やすことによって１ノードの実行時間に対してどれくらい実行時間が早くなることを示す台数効果の結果も図７に示す．最小パス深さの設定は台数効果を大きく左右することがわかる．”Simple”で表される単純並列化はノード数を4以上に増やしても台数効果が横ばいになるという最悪の性能を示す．単純並列化では、最も負荷が重いノードによって全体の性能が制限されることから自明な結果ともいえる．

同様な現象が”pdepth min 20”のように最小パス深さが大きすぎる場合も起こる．並列実行の粒度が大きいので、負荷が十分均等化されていない．それに対して、”pdepth min 5”のように最小パス深さが小さすぎる場合でも、小さい条件つきパターンベースが多数生成・保存されるオーバヘッドのため、台数効果もほとんど向上しない．

”pdepth min 12”のように最小パス深さが最適な場合、十分な台数効果が得られることも分かる．８ノード実行時では、７．３倍高速化され、３２ノードの時でも約２２．６倍の高速化が達成できる．

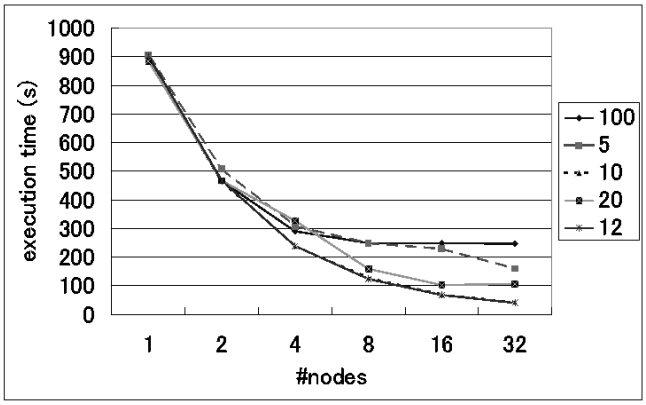


図７　台数効果

Fig.7　Speedup ratio

## 6. まとめと今後の課題

無共有型計算機上で実行される並列 FP-growth の開発を報告した．FP-tree のようなデータ構造は複雑で、並列化には適していないと思われるが、提案した手法が３２ノードからなる PC クラスタ上で実装し、十分な台数効果を上げることができることが確認された．並列実行粒度の制御のために、「パス深さ」を導入し、最適な最小パス深さの設定では、実行ノードを３２ノードに増やした時、２２．６倍の高速化が達成できる．

これから他の並列アルゴリズムとの比較及び、提案した手法を他のアルゴリズムへの適用を試みる．

## [文献]

[1] R. Agarwal, C. Aggarwal and V.V.V. Prasad``A Tree Projection Algorithm for Generation of Frequent Itemsets". In J. Parallel and Distributed Computing, 2000

[2]R. Agrawal and R. Srikant."Fast Algorithms for Mining Association Rules". In Proceedings of the 20th Int. Conf. on VLDB, pp.487-499, September 1994.

[3]R. Agrawal and J.~C. Shafer.``Parallel Mining of Associaton Rules". In IEEE Transaction on Knowledge and Data Engineering, Vol.8, No.6, pp.962-969, December, 1996.

[4]J. Han, J. Pei and Y. Yin "Mining Frequent Pattern without Candidate Generation"In Proc. of the ACM SIGMOD Conf. on Management of Data, 2000

[5]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 Proc. of Int. Conf. on Data Mining, 2001

[6]J.S.Park, M.-S.Chen, P.S.Yu "Efficient Parallel Algorithms for Mining Association Rules" In Proc. of 4th Int. Conf. on Information and Knowledge Management (CIKM'95), pp.31-36, November, 1995

[7]T. Shintani and M. Kitsuregawa ``Hash Based Parallel Algorithms for Mining Association Rules". In IEEE Fourth Int. Conf. on Parallel and Distributed Information Systems, pp.19-30, December 1996.

[8]T. Shintani, M. Kitsuregawa ``Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy." In Proc. of the ACM SIGMOD Conf. on Management of Data, 1998.

**イコ プラムディオノ　Pramudiono IKO**

東京大学大学院工学系研究科博士課程在学中．2000 東京大学大学院工学系研究科修士課程修了．並列データマイニング・ウェブマイニングの研究・開発に従事．日本データベース学会学生会員．

**喜連川 優　Masaru KITSUREGAWA**

1978 年東京大学工学部電子工学科卒．1983 年同大大学院工学系研究科情報工学博士課程了．工学博士．同年同大生産技術研究所講師．現在，同教授．平成 15 年 4 月より、同所 戦略情報融合国際研究センター長．データベース工学，並列処理，Ｗｅｂマイニングに関する研究に従事． 情報処理学会理事，SNIA-Japan 顧問，ACM SIGMOD Japan Chapter Chair．平成 9，10 年本学会データ工学研究専門委員会委員長． VLDB Trustee，IEEE ICDE，PAKDD，WAIM ステアリングコミティメンバ．