社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－DBS－134 (II) (76)
2004／7／14

木構造を用いた並列頻出パターンマイニングにおける動的負荷分散機構

イコ プラムディオノ† 喜連川 優††

† 日本電信電話株式会社 NTT 情報流通プラットフォーム研究所
†† 東京大学生産技術研究所

あらまし 大規模データベースから頻出パターンを効率的に発掘するのに、データ特性への適応が課題とされてきた．ＰＣクラスタ上に FP-growth を基にした並列マイニングアルゴリズムを開発した．FP-growth が使用する FP-tree というデータ構造は分割が困難とされており、ノード間の処理負荷の偏りが増大する．提案する並列アルゴリズムは新たに導入される path depth とよばれるパラメータによって処理負荷を予測する．path depth は頻出パターンになりうる FP-tree の枝の長さから計算できる．しかし path depth による負荷分散制御はデータ特性依存パラメータを用いるため、未知のデータに対して、最適な値の決定が困難であった．本発表では、そのパラメータをマイニング実行中に最適化できる手法を提案する．

Adaptive Load Balancing on Tree Based Parallel Frequent Pattern Mining

Iko PRAMUDIONO† and Masaru KITSUREGAWA††

† NTT Information Sharing Platform Laboratories, NTT Corporation
†† Institute of Industrial Science, The University of Tokyo

Abstract Adaptability is a major challenge to efficiently mine frequent patterns from large scale databases. We develop FP-growth based parallel mining algorithms on a PC cluster. Since the FP-tree is a complex data structure, it is difficult to partition and also increases the processing skew among nodes. The parallel algorithm employs a parameter called "path-depth" to estimate the workload from the minimum length of the tree-branches that possibly become frequent. Since the path depth is a data dependent parameter, we develop adaptive approaches to dynamically adjust the parameter during the execution.

1. はじめに

ほとんどの無共有型計算機上の並列頻出パターンマイニングアルゴリズムは Apriori と呼ばれるアルゴリズムを基にしている [1], [4], [5], [7], [8].

我々の並列実行フレームワークは、pattern-growth パラダイムに基づく FP-growth アルゴリズムを採用する [3]. FP-growth の主な概念はデータベースを FP-tree と呼ばれるメモリ上のコンパクトなデータ構造に圧縮し、そして divide-and-conquer 手法で FP-tree からすべての頻出パターンを抽出する．FP-growth はその際に、アイテムセットごとに *conditional pattern bases* というサブデータベースを再帰的に生成する．

FP-growth を基にした並列アルゴリズムに関しては、Multiple Local Frequent Pattern Tree (MLFPT) がメモリ共有型計算機のために開発された [9].

FP-growth アルゴリズムは無共有型計算機上で実装する

場合、いくつかの制約がある．特に、効率的なポインター探索のために、そのデータ構造がメモリ上に常駐しなければならない．その上，FP-tree というデータ構造は複雑であり、分割は容易ではない．

無共有型計算機では、CPU だけではなく、メモリも貴重な資源であるため、メモリと負荷の分散機構がシステムのスケラビリティーに重要な役割を担う．特に、ヘテロ環境のＰＣクラスタでは、データ特性とシステム構成に適応する必要がある．並列マイニングシステムを設計する上で、いくつかの要因がある．

（1） 独立実行単位の特定

独立実行単位とは、任意の実行ノードで独立に実行可能な処理単位のことである．

（2） 負荷分散

並列実行負荷のサイズを制御する機構である．

（3） メモリ空間の配分

全体メモリ消費の最適化と同時に、ノード間のメモリ配分もバランスを図る．

我々は提案した FP-growth の並列実行フレームワークでは、conditional pattern base 処理を並列実行処理の単位として採用し、負荷予測のために、*path depth* という新たなパラメータを導入する [6]．固定 minimum path depth を用いた負荷分散機構では、ヘテロ構成のＰＣクラスタに適用できることを示した．本フレームワークはシステム全体のメモリ消費を空きメモリの容量に応じて動的に最適化する．さらに、本フレームワークはウェブアクセスパターンマイニングにも拡張できることを示した．しかし本フレームワークはいくつかの欠点を抱えている．とりわけ、最適な固定 minimum path depth を見つけることが容易ではない．

本論文では、その欠点を克服する為に、適応的な負荷分散機構について検討をする．本フレームワークでは、以下の三つの階層の適応的な負荷分散機構を実装する．

（1） デマンドベース方式の conditional pattern base 交換

負荷をビジー状態の実行ノードからアイドル状態の実行ノードに移す

（2） minimum path depth による conditional pattern base 分解

各々の並列実行単位に上限を設定することで、負荷偏りを解消する

（3） 実行中に minimum path depth を適応的に決定する

複雑な実行パラメータ最適化作業を解消する

各々の機構は下位の階層の機構を基に実装され、実行単位の配分する機能を拡張する．実験結果により、提案された適応的な負荷分散機構の性能を評価する．

2. FP-growth

まず、並列実行フレームワークの元となる FP-growth アルゴリズムについて簡単に説明する．FP-growth は FP-tree の構築と FP-tree から全ての頻出パターンのマイニングという二つのフェイズに分けられる．

2.1 FP-tree 構築

FP-tree の構築には２回のデータベーススキャンが必要である．最初のスキャンで、各アイテムの支持度（サポート）を求め、頻出アイテムと呼ばれる最小支持度を満たすアイテムを見つけ出す．その頻出アイテムは頻出１－アイテムセットとも呼ばれており、支持度順にソートされ、F-list と呼ばれるアイテムのリストを形成する．２回目のスキャンで実際に FP-tree を構築する．

まず，各トランザクションを頻出ではないアイテムを落としながら、F-list のアイテム順に再構成する．再構成されたトランザクションを一つずつ FP-tree に挿入する．トランザクションのアイテムに該当する FP-tree ノードが存在すれば、そのノードにあるカウントを増加させるだけだが、該当する FP-tree ノードがなければ、新たなノードが生成され、そのカウントを１に設定する．FP-tree 構築の際に、アイテム間の順番は重要な意味を持つ．トランザクション内の共通プリフィックスはノードを共有できるため、データベースを圧縮できるのである．

FP-tree は頻出アイテムに関するヘッダーテーブルを持っており、同アイテムのノードをつなぐ node-link のヘッドはそのテーブルに置かれている．node-link は頻出パターンをマイニングする際の FP-tree 探索の基点となる．

2.2 FP-growth の再帰的プロセス

FP-growth アルゴリズムの入力は FP-tree と最小支持度である．最小支持度を満たす全ての頻出パターンを発掘するのに、FP-growth は同サフィックスをもつパターンを順次処理する conditional-search という新たなパラダイムを採用した．

FP-growth は F-list の最も支持度の低いアイテムから FP-tree のノードを探索する．node-link に従い、サフィックスとなるノードを一つずつ訪れ、それを基点に FP-tree のルーツまでその FP-tree のパスを一つずつ辿る．各ノードを探索する際に、prefix-path というサフィックスノードからルーツまでのパス情報を収集する．prefix-path には、ノードラベルの他に、サフィックスノードのカウントも prefix-path の支持度として記録される．一つのサフィックスアイテムの全ての prefix-path はそのアイテムに関する conditional pattern base を形成する．

あるアイテムセットに関する conditional pattern base はそのアイテムセットに共起するトランザクションデータベース内の全てのトランザクション部分を収納し、そのアイテムセットに関するすべての頻出パターンをマイニングするのに必要十分な情報を持っている．その conditional pattern base よりそのアイテムに関する conditional FP-tree という小さな FP-tree を生成する．この再帰的な繰り返しごとに処理中のアイテムセットに一つのサフィックスを追加し、最小支持度を満たせば、新たな頻出パターンを生成する．新たな conditional pattern base が生成されないまで、再帰的なプロセスが繰り返し実行される．

3. FP-growth の単純並列実行方式

下記の補助定理により、conditional pattern base 処理を並列実行単位にするのが自然である．

[補助定理 1] conditional pattern base の処理は他の conditional pattern base 処理とは独立である．

Proof

conditional pattern base の定義より、アイテム a の conditional pattern base 生成はデータベース内のそのアイテムの存在にしか依存しない．conditional pattern base は node-link を辿ることで生成できるが、node-link は同アイテムだけをつなぐこととノード削除が行われないことのために、生成された conditional pattern base は他のアイテムの node-link からの影響はない．

3.1 基本動作

基本的なアイデアはそれぞれのノードが全てのノードからアイテムに関する prefix path を集め、完全な conditional pattern base を再構築する．そしてその conditional pattern base を独立に処理する．本手法の擬似コードは図 2 に示し、図 1 はその並列実行の例を示す．

基本的に二つのプロセスが必要である：SEND プロセスと RECV プロセス．最初のスキャン後、SEND プロセスはグローバルな支持度を求めるため、全てのアイテムの支持度を交換する．そして、そのグローバル支持度を基にそれぞれのノードが F-list を作成する．2 回目のデータベーススキャンで、SEND プロセスがローカルな FP-tree を構築する．

ローカルな FP-tree よりローカルな conditional pattern base が生成される．そして、SEND プロセスがハッシュ関数を用いて conditional pattern base を処理するノードを決定する．送り先ノードの RECV プロセスはすべての SEND プロセスから conditional pattern base を収集し、再構築をする．

3.2 Conditional pattern bases 交換手法

RECV プロセスが conditional pattern base を全ての

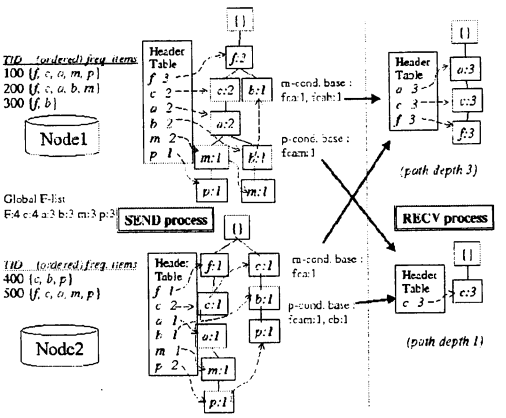

図 1 FP-growth の単純並列実行

```
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:while(not end_proc()) do
2:  cond_pbase = collect_cond_pbase();
3:  cond_FPtree = construct_fptree(cond_pbase, FList);
4:  FP-growth(cond_FPtree, NULL);
5:end while
}
```

図 2 単純な並列実行方式の擬似コード

SEND プロセスからの受信を完了してからしか conditional pattern base の構築を始められないので、conditional pattern base 交換のフェイズがシステム全体の性能を大きく左右する．

交換手法を決定する上で重要な要素は RECV プロセスの間に処理負荷が大きく異なるという点である．そのため、交換の際に、処理ノード間の同期が非常に困難となる．ビジーなノードは処理終了まで、他のノードからの要求を受け付けず、システム全体のボトルネックとなる可能性がある．

Conditional pattern base 交換方式として以下のハイブリッド方式を提案する．提案する方式は 2 つの実行フェイズからなる：

（1） ラウンドロビンフェイズ

F-list のアイテム順に従い、サフィックス処理を行い、送

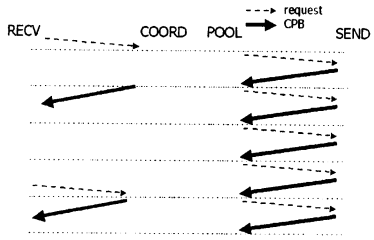

図 3　デマンドベースフェイズタイムライン

り先ノードをハッシュ関数で決定する.

（2） デマンドベースフェイズ

F-list の全てのアイテムの処理が終了したら、COORD と POOL という二つの軽量プロセスが制御するデマンドベースフェイズに移る. デマンドベースフェイズでは、conditional pattern base をアイドル状態の実行ノードに送信する. POOL プロセスが SEND プロセスより conditional pattern base を収集し、COORD プロセスがアイドル状態にある RECV プロセスより要求を受け付けて、収集された conditional pattern base を必要な RECV プロセスに送信する.

図 3 はデマンドベースフェイズの様子を時系列的に表すタイムラインを示す. そのタイムラインでは、要求及び、conditional pattern base の流れを表している. RECV プロセスが自ノードの conditional pattern base を消化し、他のノードから conditional pattern base を必要とするとき、COORD プロセスに要求を出す. その結果、POOL プロセスを通じて、SEND プロセスに要求が伝わり、十分な利用可能な conditional pattern base を保持する SEND プロセスがその要求に応じて余分な conditional pattern base を送り返す. COORD プロセスがそれらの conditional pattern base を必要な RECV プロセスに配分する.

4. Path depth を用いた負荷分散

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 conditional pattern base のランダムな配布を採用しているが、それぞれの conditional pattern base 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、conditional pattern base を用いる場合、並列実行粒度は次の conditional pattern base を生成する繰り返し回数によって決まる. その回数は conditional pattern base 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

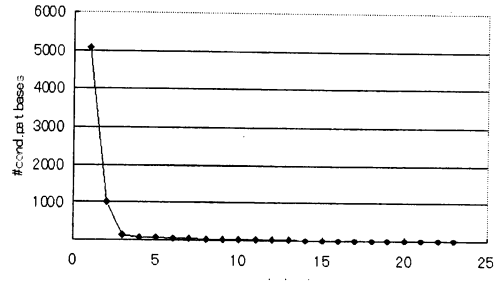

図 4　Path depth 分布

「path depth」を定義する.

[定義 1] path depth の定義：conditional pattern base 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.

path depth の例：図 1 では、m の conditional pattern base 内で最も長いパターンは {acf} のため、その path depth は 3 である.

path depth の典型的な分布は図 4 に示される. ほとんどの conditional path depth が短い path depth をもっているが、非常に長い path depth もいくつかある. 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.

4.1 固定 minimum path depth 方式

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. path depth が FP-tree を構築する時、あらかじめ計算できるので、path depth 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、conditional pattern base より conditional FP-tree を構築してから、さらに小さい conditional pattern base を生成する FP-growth の再帰的な性格を利用する. より小さい conditional pattern base が生成される時、path depthp が一つ減らされる.

minimum path depth 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. path depth を利用する粒度均等化メカニズムの擬似コードを図 5 に示す. minimum path depth 以下の path depth を持つ conditional pattern base 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 conditional pattern base 生成までは実行されるが、生成された conditional pattern base 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.

4.2 Adaptive path depth reduction

最適な固定 minimum path depth の決定は容易なこと

```
SEND process :
{
1:while(not end_proc()) do
2:  if(received(wait_request()) then
3:    cond_pbase = get_stored_cond_pbase();
4:    if(cond_pbase is not NULL) then
5:      send_cond_pbase(cond_pbase);
6:    end if
7:  end if
8:end while
}
RECV process :
{
1:while(not end_proc()) do
2:  cond_pbase = get_stored_cond_pbase();
3:  if(cond_pbase is NULL) then
4:    cond_pbase = receive_cond_pbase();
5:  end if
6:  cond_FPtree = construct_fptree(cond_pbase, FList);
7:  FP-growth_pdepth(cond_FPtree, cond_pbase.itemset);
8:end while
}
procedure FP-growth_pdepth(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_pdepth(Y-Tree, Y);
8:    end if
9:  else
10:    store_cond_pbase(cond_pbase, Y);
11:  end if
12:end for
}
```

図 5　固定 minimum path depth 擬似コード

ではない．通常数回の実行が必要だと思われる．未知のデータに対しては，特に困難である．

そのため，path depth を決定する適応的な手法を検討する．固定 minimum path depth α のようなデータ依存パラメータを解消し，負荷分散を均等化するため path depth を実行中のシステム状況に応じて動的に変化させる．

適応的 path depth 決定手法は conditional pattern base 交換におけるデマンドベースフェイズに機能する．本手法は，明示的に送信通知を受けた SEND プロセスのみが余分な conditional pattern base を送信することになっている．minimum path depth α は最大トランザクション長より大きいというような十分高い値に予め設定し，最初では conditional pattern base 分割が行われないようにする．全体としてより少ない conditional pattern base が生成され，オーバヘッドが少なくなる．

提案される手法の擬似コードが図 6 に示され，図 5 に示された SEND プロセスに変更を加えたことで実装できる．conditional pattern base に対する要求が着いた時点に，SEND プロセスが余分な conditional pattern base を保持しない場合，minimum path depth が次第に減少され，新

```
SEND process :
{
1:min_path_depth = longest_transaction;
2:while(not end_proc()) do
3:  if(received(wait_request()) then
4:    cond_pbase = get_stored_cond_pbase();
5:    if(cond_pbase is not NULL) then
6:      send_cond_pbase(cond_pbase);
7:    else if (min_path_depth > system_lim) then
8:      reduce(min_path_depth);
9:    end if
10:  end if
11:end while
}
```

図 6　Adaptive Path Depth Reduction 擬似コード

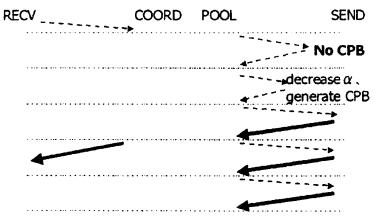

図 7　Adaptive Path Depth Reduction タイムライン

たな conditional pattern base が生成されるようにする．実装では，システム環境依存の *system_lim* という minimum path depth の下限になるパラメータを設定し，過剰な conditional pattern depth 生成を防ぐ．system_lim はいくつかの PC クラスタ構成の要素に依存する：ネットワークスループット，ネットワークレイテンシ，CPU 性能，メモリサイズ，メモリアクセス性能，実行ノード数等である．system_lim を用いる利点はデータセット特性への依存性が高くないことである．提案される手法が minimum path depth の減少を利用することで，Adaptive Path Depth Reduction (APDR) とよぶ．

APDR のデマンドベースフェイズのタイムラインは図 7 に示される．POOL プロセスから要求を受け付けたら，十分な connditional pattern base が生成できるまで，SEND プロセスが徐々に minimum path depth を減少させる．

4.3　Turbo adaptive path depth

APDR は，実行ノード間のタイミング等のため，十分な台数効果を得ることができない．実行トレースを解析した結果，以下の改良点を提案する：

- 不要なときに，minimum path depth リセット

Minimum path depth を減らした後，その値を低いままに維持すると多数の conditional pattern base が生成されることがある．あるいは，生成される conditional pattern base が非常に小さく，保存するオーバヘッドが大きくなり，効率が悪化することもある．そのため，minimum path depth を初期値に戻し，不要な conditional pattern base

```
RECV process :
{
1:while(not end_proc()) do
2:   cond_pbase = get_stored_cond_pbase();
3:   if(cond_pbase is NULL) then
4:     cond_pbase = receive_cond_pbase();
; reset the min_path_depth
5:     min_path_depth = longest_transaction;
6:   end if
7:   cond_FPtree = construct_fptree(cond_pbase, FList);
8:   FP-growth_pdepth(cond_FPtree, cond_pbase.itemset);
9:end while
}
```

図 8　Turbo Adaptive Path Depth 擬似コード

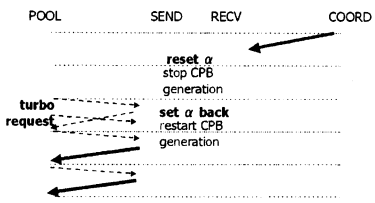

図 9　Turbo Adaptive Path Depth タイムライン

生成を回避することができる.

- 過剰な要求を SEND プロセスに送信し、conditional pattern base 生成のレイテンシを隠蔽する

Conditional pattern base が実際に生成されるまで、minimum path depth が一つずつ減少させられるため、要求が着いてから生成までレイテンシが生じることがある. そのため、一つの要求送信に続いてもう一つ過剰な要求を送信するターボ手法で、minimum path depth の減少率を向上させ、レイテンシを減らすことができる.

改良された手法を Turbo Adaptive Path Depth (TAPD) とよぶ. 1番目の改良点は図 8 に示される通り、RECV プロセスに実装される. RECV プロセスが他のノードから conditional pattern base を受信したら、minimum path depth を初期値、つまり最大トランザクション長にリセットされる.

2番目の改良点は POOL プロセスに実装される. APDR のデマンドベースフェイズのタイムラインは図 9 に示される. RECV プロセスが conditional pattern base を受けたら、minimum path depth をリセットし、conditional pattern base 生成を停止させる. 過剰な要求の送信によって、conditional pattern base 生成を加速させている.

5. PC cluster 上の性能評価

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

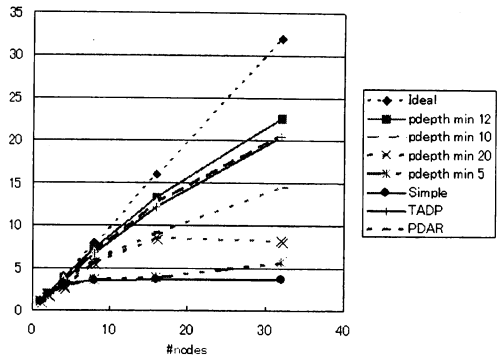

図 10　台数効果 T25.I20.D100K.i5K 0.1%

メモリを搭載している. プログラムは C 言語により実装され、OS として Solaris8 を採用した.

5.1 データセット

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.

5.2 台数効果

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 10 に示す. minimum path depth 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 minimum path depth 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 minimum path depth が小さすぎる場合でも、小さい conditional pattern base 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 minimum path depth 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.

適応的な手法の場合、"APDR" で表される Adaptive Path Depth Reduction は 32 ノードで、14.6 倍しか

高速化されないのに対して、TAPD で表される Turbo Adaptive Path Depth は２０．４倍を記録できる．この時、system_lim は７に設定されている．適応的な手法は minimum path depth を調整する際のレイテンシ等のオーバヘッドのため、最適な固定 minimum path depth 手法より性能が劣っている．

6. おわりに

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、「path depth」を導入し、最適な minimum path depth 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．データ依存パラメータを解消した適応的な手法である Turbo Adaptive Path Depth でも、最適な手法に迫る性能を出すことができることを示した．これから提案した手法を多くのデータセットへの適用を試み、データセット特性への依存性について検討する．

文　献

[1] R. Agrawal and J. C. Shafer. "Parallel Mining of Association Rules". In *IEEE Transaction on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 962–969, December, 1996.

[2] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In *Proceedings of the 20th Int. Conf. on Very Large Data Bases(VLDB)*, pp. 487–499, September 1994.

[3] 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.

[4] K. Goda, T. Tamura, M. Oguci, and M. Kitsuregawa "Run-time Load Balancing System on SAN-connected PC Cluster for Dynamic Injection of CPU and Disk Resource". In *Proc of 13th Int. Conf. on Database and Expert Systems Applications (DEXA'02)*, 2002.

[5] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri "Adaptive and Resource-Aware Mining of Frequent Sets". In *Proc. of the Int. Conf. on Data Mining(ICDE)*, 2002.

[6] I. Pramudiono and M. Kitsuregawa "Tree Structure based Parallel Frequent Pattern Mining on PC Cluster". In *Proc of 14th Int. Conf. on Database and Expert Systems Applications (DEXA'03)*, 2003.

[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] M. Tamura. and M. Kitsuregawa "Dynamic Load Balancing for Parallel Association Rule Mining on Heterogeneous PC Cluster Systems". In *Proc. of the 25th Int. Conf. on Very Large Data Bases(VLDB)*, 1999.

[9] O. R. Zaiane, M. El-Hajj, and P. Lu. "Fast Parallel Association Rule Mining Without Candidacy Generation" In *Proc. of the IEEE 2001 Int. Conf. on Data Mining (ICDM'2001)*, pp. 665–668, 2001