DEIM Forum 2023 2b-4-3

動的負荷分散を用いた関連パターン発見手法の共有メモリ型並列化

櫻井健太朗† 亀谷 由隆†

† 名城大学大学院理工学研究科情報工学専攻 〒 468–8502 愛知県名古屋市天白区塩釜口 1–501
E-mail: †ykameya@meijo-u.ac.jp

あらまし 注目するクラスと関連性の高いパターンの抽出を目的として、関連度の上限における逆単調性に基づく分枝限定法を FP-growth に導入したパターン発見手法が知られる。しかし、指定する関連度の指標やデータ規模、パターン間の冗長性を除くための制約により多大な計算時間を要する場合がある。そこで本研究では共有メモリ型システム上での並列化による高速化を図る。スケーラビリティ向上のため、タスク量の推定によって候補パターンの探索空間を動的に分割し、負荷を分散する。また、実装では Rust が持つ軽量性やメモリ安全性を利用する。実験の結果、探索空間が増大するような条件にて理想に近い速度向上が得られた。

キーワード データマイニング、並列・分散処理、先進ハードウェア活用

1 はじめに

関連（relevant）パターンとは、クラスラベルが付与されたトランザクションデータベースにおいて注目するクラスと関連性の高いパターンである。関連パターンの発見手法として RP-growth が提案されている [6]。これは、頻出パターン発見に用いられる FP-growth という手法に関連度を導入した手法である。多くの頻出パターン発見手法では、サポート値の逆単調性を用いて候補パターンの探索空間の枝刈りを行う [1] [3]。一方、RP-growth では、使用する関連度の指標が逆単調性を満たさないため、分枝限定法を導入している。また、冗長なパターンの出力を防ぐ様々な制約が提案されており、データベース内に現れる有用なパターンの抽出が可能である。本研究では、上位 k パターン制約 [4] [6] と最良カバー制約 [5] の 2 種類を別々に導入した RP-growth を扱う。

RP-growth は分枝限定法を導入しているものの、指定する関連度の指標やデータセットの規模、パターン間の冗長性を除くための制約によっては多大な処理時間を要する場合がある。特に、候補パターンの探索時間が増大する傾向があり、高速化が望まれている。そこで本研究では、共有メモリ型システム上での並列化によって RP-growth の高速化を図る。近年、多数のコアを搭載した CPU（メニーコア CPU）が入手しやすくなっており、メニーコア上での効率的な並列処理が求められている。具体的には、共有メモリ型システムでよく用いられる動的負荷分散手法である work-stealing を利用する [8]。RP-growth の並列化においては、候補パターン探索木の部分木に対するタスク量推定によって探索木を分割しタスクを生成してタスクの粒度を調節し、スケジューリングコストとロードバランスの最適化を図る。なお、関連パターン発見に類似したサブグループ発見の並列化手法が提案されているが [9]、こちらは非共有メモリ型の並列環境を前提としている。

さらに、本研究では実装言語として軽量性とメモリ安全性を持つ Rust を用いる。Rust は、コンパイル型言語であり、ガーベジコレクションを持たない。従って比較的小さなバイナリで実行でき、ランタイムでのメモリ消費を抑えることができるため、共有メモリ型並列処理に向いている。また、メモリ安全性によりデータ競合を適切に回避でき、メモリ関連のバグの削減に大きく貢献する。

2 準 備

ここでは、関連パターン発見とその並列処理に用いられる手法について説明する。

2.1 頻出パターン発見

先述の通り、本研究では頻出パターン発見の代表的手法である FP-growth に基づいた関連パターン発見手法である RP-growth を並列化の対象とする。頻出パターン発見は、トランザクションデータベースが与えられたときに、事前に定めた最小のサポート値（あるパターンがトランザクションに含まれる数）以上のパターンを全て列挙する問題である。FP-growth [3] は、データベースを FP-tree と呼ばれる Trie 木をベースとしたデータ構造に縮約しながら深さ優先的に頻出パターンを探索するアルゴリズムである。その際、FP-tree の構築は、データベース全体を列ごとに分割し、ノードに変換していくことで FP-tree を構築する方法であり、Borgelt の実装において初回 FP-tree 構築に用いられている [2]。

2.2 関 連 度

入力は、クラスラベルが付与された N 個のトランザクションからなるトランザクションデータベース $\boldsymbol{D} = \{t_1, t_2, ..., t_N\}$ である。頻出パターン発見と異なり、各トランザクションはクラス集合 C のうちの 1 つ c_i に属する $(1 \leqq i \leqq N)$。アイテム間に全順序 $\prec$ を導入し、トランザクション/パターン中のアイテムは常に $\prec$ に従った順序で並べられるとする。また、トランザクション/パターン間にも $\prec$ に基づく辞書順序を考える。

ここで、トランザクションデータベース $\boldsymbol{D}$ のいくつかの部分集合を定義する。$c \in \boldsymbol{C}$ を興味のあるクラスとすると、$\boldsymbol{D}_c =$

$\{t_i \mid c_i = c, 1 \leqq i \leqq N\}$、$\boldsymbol{D}(x) = \{t_i \mid \boldsymbol{x} \subseteq t_i, 1 \leqq i \leqq N\}$、そして、$\boldsymbol{D}_c(x) = \{t_i \mid c_i = c, \boldsymbol{x} \subseteq t_i, 1 \leqq i \leqq N\}$ と定義する。また、$\neg$ を否定に用いて、$\boldsymbol{D}_{\neg c} = \boldsymbol{D} \setminus \boldsymbol{D}_c$、$\boldsymbol{D}_c(\neg x) = \boldsymbol{D}_c \setminus \boldsymbol{D}_c(\boldsymbol{x})$ そして、$\boldsymbol{D}_{\neg c}(\boldsymbol{x}) = \boldsymbol{D}(\boldsymbol{x}) \setminus \boldsymbol{D}_c(\boldsymbol{x})$ とそれぞれ定義する。$\boldsymbol{D}_c$ に含まれるトランザクションのことを正トランザクションと呼ぶ。

確率は全てトランザクションデータベース $\boldsymbol{D}$ から計算される。特に、同時確率 $p(c, \boldsymbol{x})$ は $|\boldsymbol{D}_c(\boldsymbol{x})|/N$ より得られる。同様に、$p(c, \neg\boldsymbol{x}) = |\boldsymbol{D}_c(\neg\boldsymbol{x})|/N$、$p(\neg c, \boldsymbol{x}) = |\boldsymbol{D}_{\neg c}(\boldsymbol{x})|/N$ などというように得ることができる。この同時確率を用いることで、周辺確率や条件付き確率が計算される。頻出パターン発見では、条件付き確率である $p(\boldsymbol{x} \mid c)$、$p(\boldsymbol{x} \mid \neg c)$、そして $p(c \mid \boldsymbol{x})$ をそれぞれ、正サポート、負サポート、そして確信度と呼ぶ。また、単にサポートと言うときは正サポートのことを指す。

RP-growth では、クラス c に対するパターン $\boldsymbol{x}$ の関連度は $R_c(\boldsymbol{x})$ で表される。よく用いられているパターンの関連度を表す指標は、正クラス確率 $p(c)$、正サポート $p(\boldsymbol{x} \mid c)$、負サポート $p(\boldsymbol{x} \mid \neg c)$ の関数である。例えば本論文では、関連度 $R_c(\boldsymbol{x})$ として F 値や χ^2 値を考える。F 値は確信度（適合率）と正サポート（再現率）の調和平均であり、χ^2 値は 2 確率変数間の関連の強さを表す指標である。この関連度の指標となる χ^2 値は双単調性と呼ばれる性質を満たす。すなわち、$p(\boldsymbol{x} \mid c) \geqq p(\boldsymbol{x} \mid \neg c)$ なる任意のパターン $\boldsymbol{x}$ に対して $R_c(\boldsymbol{x})$ が $p(\boldsymbol{x} \mid c)$ に関して単調増加かつ、$p(\boldsymbol{x} \mid \neg c)$ に関して単調減少である。本論文で扱うアルゴリズムは双単調性を持つ関連度に対して動く。

2.3 上位 k パターン制約

関連パターン発見では冗長なパターンが多く含まれたり事前設定次第で出力パターン数が増大する。上位 k パターン制約は、出力パターンを関連度の上位 k 個に限定するものである。FP-growth のように最小サポートを設定する方法では、出力パターン数がデータセットの規模によって大きく変わり、有益でないパターンが大量に出力される場合があるが、上位 k パターン制約によってそれを防ぐことができる。

上位 k パターンを効率的に探索する方法として、最小サポート上昇法が知られる [4]。頻出パターン発見における最小サポート上昇法では、最小サポート σ_{min} を訪問した候補パターンを用いて上昇させて探索空間の削減を図る。具体的には、パターンの候補リストを用意し、見つかったパターンをサポートの大きい順に格納する。探索の途中で候補リストのサイズが k 以上になったとき、k 番目に大きいサポートを持つパターンを $\boldsymbol{z}$ とする。その後に探索されるパターン $\boldsymbol{x}$ は $p(\boldsymbol{x} \mid c) \geqq p(\boldsymbol{z} \mid c)$ を満たす必要があり、$p(\boldsymbol{x} \mid c) < p(\boldsymbol{z} \mid c)$ なら、$\boldsymbol{x}$ 以下の部分木を枝刈りできる。これは、$\sigma_{min} := \max\{p(\boldsymbol{z} \mid c), \sigma_{min}\}$ と上昇させ、最小サポートに従って枝刈りする操作と等価である。また、候補リストに存在する $p(\boldsymbol{z} \mid c)$ 未満のサポートを持つパターンは破棄される。

RP-growth では、構築する FP-tree 内の各ノードに正カウントと負カウントを持たせることで、パターンが含まれるトランザクション集合の正例と負例の数を保持する。また、正カウントと負カウントを用いてパターンの関連度を計算できる。FP-growth ではサポート値が逆単調性を満たすため、訪問したパターンのサポート値が最小サポート未満である場合、それ以下の部分木を枝刈りできる。一方、RP-growth の候補パターンの探索では、パターンの関連度が逆単調性を満たさないため分枝限定法を利用する。すなわち、探索木で訪問中の各パターン $\boldsymbol{x}$ について、逆単調性を満たすような $R_c(\boldsymbol{x})$ の上界 $\overline{R}_c(\boldsymbol{x})$ を計算し、必要とされる閾値をこの上界が下回ったら $\boldsymbol{x}$ 以下の部分木は枝刈りされる。具体的には、探索途中で k 番目に大きな関連度を持つパターンを $\boldsymbol{z}$ としたとき、$\overline{R}_c(\boldsymbol{x}) < R_c(\boldsymbol{z})$ となるパターン $\boldsymbol{x}$ が見つかったら、探索木での $\boldsymbol{x}$ を根とする部分木を安全に枝刈りする。

逆単調性を満たすような関連度の上界は次のようにして得る。まず $R_c(\boldsymbol{x})$ は双単調性を満たすことから、$R_c(\boldsymbol{x})$ は $p(\boldsymbol{x} \mid c)$ に関して単調増加し、$p(\boldsymbol{x} \mid c)$ と $p(\boldsymbol{x} \mid \neg c)$ ともにパターンの包含関係に関して逆単調である。このとき、まず上界 $\overline{R}_c(\boldsymbol{x})$ を得るために、$p(\boldsymbol{x} \mid \neg c) = 0$ を $R_c(\boldsymbol{x})$ の定義式に代入する。その上界 $\overline{R}_c(\boldsymbol{x})$ は R_c の双単調性によってパターンの包含関係において、常に逆単調性を満たす。例えば、F 値 $R_c(\boldsymbol{x}) = 2p(c \mid \boldsymbol{x})p(\boldsymbol{x} \mid c)/(p(c \mid \boldsymbol{x}) + p(\boldsymbol{x} \mid c))$ の上界は、$\overline{R}_c(\boldsymbol{x}) = 2p(\boldsymbol{x} \mid c)/(1 + p(\boldsymbol{x} \mid c))$ より得られる。

さらに、この分枝限定法を頻出パターン発見手法で用いられるような最小サポートを用いる枝刈りに翻訳する。$\overline{R}_c(\boldsymbol{x}) < R_c(\boldsymbol{z})$ なら、$\boldsymbol{x}$ 以下を枝刈りできるため、この不等式を $p(\boldsymbol{x} \mid c)$ について解き、右辺の式を $U_c(\boldsymbol{z})$ とすると、以下のようになる。

$$p(\boldsymbol{x} \mid c) < U_c(\boldsymbol{z})$$

F 値の場合、$U_c(\boldsymbol{z}) = R_c(\boldsymbol{z})/(2 - R_c(\boldsymbol{z}))$ である。これは、$\boldsymbol{x}$ のサポートが k 番目の関連度をもつパターン $\boldsymbol{z}$ より得られるサポート未満であるとき、$\boldsymbol{x}$ 以下を枝刈りするということを意味する。これを用いて、最小サポートを

$$\sigma_{min} := \max\{U_c(\boldsymbol{z}),\ \sigma_{min}\}$$

と上昇させることで枝刈りできる。

また、R_c の双単調性から、上で得られた $U_c(\boldsymbol{x})$ が $R_c(\boldsymbol{x})$ に関して単調増加であることを容易に示せる。よって、本研究ではパターン間で関連度の比較をする際にも R_c の代わりに U_c を使う。例えば $U_c(\boldsymbol{x}) < U_c(\boldsymbol{z})$ なるパターン $\boldsymbol{x}$ は上位 k パターンになれない。以降では $U_c(\boldsymbol{x})$ をパターン $\boldsymbol{x}$ の「上界サポート（upper support）」と呼ぶ。

2.4 最良カバー制約

最良カバー制約は、パターン間の冗長性に関する制約である [5]。最良カバー制約において、出力されるパターンはそれがカバーする正トランザクション t のいずれかにおいて、t をカバーするパターンの中で最大の関連度を持たなければいけない。表 1 のデータベースからは最良カバーパターンとして表 2 の 2 パターンのみが出力され、最良カバー制約は厳しい制約であることが分かる。

RP-growth で最良カバー制約を実現するには、正トランザクションごとに候補リストを持ち、それらの候補リストについ

表 1: クラスラベル付きトランザクションデータベースの例

番号	クラス	トランザクション
1	+	{a, b, d, e}
2	+	{a, b, c, d, e}
3	+	{a, c, d, e}
4	+	{a, b, c}
5	+	{b}
6	−	{a, b, d, e}
7	−	{b, c, d, e}
8	−	{c, d, e}
9	−	{a, b, e}
10	−	{a, d}

表 2: 最良カバーパターンの例

パターン	F 値	カバーされるトランザクションの番号
{a, c}	0.750	2, 3, 4
{b}	0.727	1, 2, 4, 5

て並行に上位 1 パターン発見を行うことによって最良カバーパターンを出力すればよい [5]。具体的には、パターンを訪問するたびに、パターンがカバーするトランザクションに対応する候補リストを参照し、それらが持つ最小サポートの中で最も小さいものをその時点での最小サポートとする。構築した FP-tree からパターンの関連度を算出後、パターンがカバーする候補リストにパターンを追加し、最良カバー制約を満たすように候補リストを更新する。この方法は、上位 k パターン発見と異なり、出力パターン数 k を指定する必要がないため、ユーザにとってより手軽に使うことができるようになる。

2.5 Work-stealing による動的負荷分散

本研究では探索木上のタスク並列処理を考える。一般的な並列処理手法としてフォーク・ジョイン並列がある。この方法は、事前にタスクをスレッドに分配してから並列処理を行う。この方法では、タスクを各スレッドに対して均等に分配する必要があり、均等でない場合は最も時間のかかるスレッドのタスクを消化し終わるまで他のスレッドが待機しなければならない。

本研究では、フォーク・ジョイン並列よりもロードバランスを向上させることが期待できる work-stealing による動的負荷分散を行う。Work-stealing は、使用するスレッドを置いておくスレッドプールと処理してほしいタスクを溜めるためのワークキューの 2 つから構成され、大量にスレッドを生成することがなくなるため使用メモリ量の増加を抑制できる。

Work-stealing の処理の様子を図 1 に示す。スレッドごとにワークキューがあり、あるスレッドが持つワークキューのタスクが 1 個以上存在するときは、そのスレッドがそのタスクを処理する。一方で、ワークキューのタスクが空になったら他スレッドのワークキューからタスクを取り出し自分のワークキューに追加する。これによって、あるスレッドが何もせず待機しているという状態が減り、効率的にプロセッサを使用できる。本研究の実装では、work-stealing による動的負荷分散を実現する Rust の Rayon（`https://github.com/rayon-rs/rayon`）[7] という並列処理ライブラリを用いた。

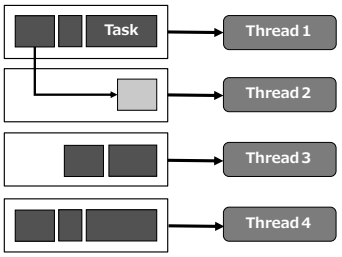


図 1: Work-stealing のイメージ

3 提案手法

本研究では、上位 k パターン制約と最良カバー制約それぞれを実現する RP-growth の並列化を提案する。どちらの制約でも、縮約データベースである FP-tree の構築と候補パターンの訪問を再帰的に繰り返し、候補リストを更新する。[11] や [12] など、既存研究の多くでは、初回の FP-tree 構築のみ並列化し、候補パターンの探索は最初の分岐におけるサイズ 1 のパターン以下の部分木それぞれの探索にタスクを分割する。

本研究では、データセットの規模や制約によって増大する探索空間をタスク量推定を用いることで適切な粒度に分割し、ロードバランスの向上を図る。また、上位 k パターン制約や最良カバー制約といったパターン間の制約を検査するにはスレッド間で候補リストを共有する必要があり、これは並列化におけるオーバヘッドとなる。本研究ではこのオーバヘッドを抑制するための工夫も行う。

3.1 初回 FP-tree 構築の並列化

初回 FP-tree 構築は特に処理時間がかかる部分であり、データセットの規模によっては以降の探索より時間がかかる場合もある。ここでは、Zaïane らによって提案されている並列構築方法を用いる [12]。初回の FP-tree の構築は以下の手順で行う。

（1） データベースをスレッド数分に分割する。

（2） アイテムの出現数を並列にカウントする。

（3） 各トランザクションから最小サポートより小さいアイテムを除外し、ソートする。

（4） 分割された各データベースで FP-tree を構築する。

これによりスレッド数分の FP-tree が構築される。そして、条件付きトランザクションをこの複数の FP-tree から取得するときはそれぞれの FP-tree のヘッダ表を参照する。

FP-tree の構築はトランザクション全体を列ごとにノードに変換する先述の Borgelt の方法 [2] と、トランザクションを逐次的に挿入する方法の 2 通りの方法がある。Rust による実装において、トランザクションを逐次的に挿入する方法は FP-tree が持つノードに可変性を持たせる必要があり、排他ロックが必要になる。一方で、データベース全体を再帰的に走査する方法は、ノードを生成してから変更が入らないため可変性が必要ない。すなわち、不変性を持つノードを生成できる。これにより、複数スレッドから同時に参照でき、排他ロックのコストやロッ

クを取得するまでの待ち時間を省ける。

3.2 候補パターン探索の並列化

候補パターン探索の並列化では、探索木の部分木の走査に対応する RP-growth の処理をタスクに割り当てる。また、探索木にて訪問中のパターンの各子供（分岐パターン）を根とする部分木に対して FP-tree の統計量に基づきタスク量推定を行い、推定したタスク量に基づき分岐パターンを均等なタスク量になるように分割し、タスクを生成する。生成されたタスクは work-stealing による動的負荷分散によって処理される。

3.2.1 タスクの定義

並列処理で扱うタスクのうち、タスク内でさらに複数のタスクを生成するものを分割タスク、割り当てられた部分木を逐次的に処理するものを逐次タスクと呼ぶ。また、分割タスクを生成するかどうかを判断するのに使用する閾値を分割閾値と呼ぶ。分割タスクは、条件付き FP-tree と分割閾値以上のタスク量推定値を持つ 1 つの分岐パターンを入力とする。逐次タスクは、条件付き FP-tree と分割閾値未満の合計タスク量推定値を持つ 1 つ以上の分岐パターンを入力とする。

まず、逐次タスクは以下の手順による処理を行う。ここで、ローカル候補リストは逐次タスク内で持つ候補リストであり、グローバル候補リストは候補パターンの管理を行い、スレッド間で共有される候補リストである。

（1） 付与された分岐パターン集合を順に訪問する。

（2） 部分木を探索し、訪問したパターンをローカル候補リストに追加する。

（3） 全ての部分木を探索し終わったら、ローカル候補リストとグローバル候補リストとマージする。

次に、分割タスクは以下の手順による処理を行う。ここで、グローバル最小サポートは、スレッド間で共有され、すべての探索の枝刈りに使用される最小サポートである。

（1） グローバル最小サポートを取得する。

（2） 与えられたパターンの条件付き FP-tree を構築する。

（3） 構築した条件付き FP-tree から各分岐パターンのタスク量を推定する。

（4） FFD ヒューリスティックス（後述）によって均等に分岐パターン集合を分割する。

（5） 分割した分岐パターン集合ごとにタスクを生成する。

（5-a） 分割閾値以上のタスク量推定値をもつ分岐パターンは分割タスクとして生成する。

（5-b） 分割閾値未満のタスク量推定値をもつ分岐パターン集合は逐次タスクとして生成する。

分割タスクの 5 番目において分割閾値を利用して逐次タスクを実行する理由は、タスクを生成すること自体にコストがかかるからである。仮に全ての分岐でタスクを生成するとスケジューリングコストが大きくなり、実行時間に影響を及ぼす可能性があるため、ロードバランスを均一にする適度な数、粒度のタスクを生成する必要がある。これを解決するため、タスク量を推定して小さいタスクを並列化せずにまとめて処理する。

3.2.2 部分探索木に対するタスク量推定

次に、RP-growth における部分探索木のタスク量の推定方法を説明する。FP-growth の並列化におけるタスク量推定は、既にいくつかの研究で提案されている。Zaïane ら [12] の研究では、分岐パターンごとのサポート値をタスク量とする。また、Yu ら [11] の研究では、複数ある FP-tree のアイテムに相当するノードの深さの総和を利用してタスク量を推定する。これらの研究では初回の分岐パターンに対してのみタスク量を推定するが、Sakurai ら [8] の研究では、さらに深いパターンの分岐パターンに対してタスク量推定して動的にタスク生成を行っている。本研究では、タスク量推定式として、Yu ら [11] のノードの深さの総和を利用する。さらに本研究では、様々な規模・特性をもつデータセットに対応するため、初回 FP-tree の葉ノードの深さの総和に対する比率を考慮する。すなわち、FP-tree$T$ をもつ分岐パターンのタスク量は式 1 のように推定する。

$$\frac{\sum_{i=1}^{m} \mathrm{depth}_i}{\sum_{i=1}^{n} \mathrm{init\_depth}_i} \tag{1}$$

ここで n は初回 FP-tree に存在する葉ノードの数、$\mathrm{init\_depth}_i$ は i 番目の初回 FP-tree のノードの深さである。また、m はその FP-tree 内の item に相当するノードの数、depth_i は i 番目のノードの深さである。

3.2.3 各分岐パターンのタスクへの割り当て

最後に、推定されたタスク量に基づいて各分岐パターンをタスクに割り当てる方法を説明する。既存研究 [8] では、タスク量推定値が分割閾値以上の分岐パターンを分割タスクに割り当て、分割閾値未満の分岐パターンを逐次タスクに割り当てていた。しかし、分割閾値未満の分岐パターンが多く存在するため、本研究ではそれらを統合し 1 つの逐次タスクに複数の分岐パターンを割り当てて過剰なタスクの生成を抑える。そのためのヒューリスティックスとして、FFD (first fit decreasing) を用いる。FFD では、タスク量の降順でタスクに割り当てることを考え、タスクが空でなくかつ分割閾値以上の場合には新しいタスクに分岐パターンを割り当てることにより、できる限り少ないタスクに均等に分散させる。

3.3 上位 k パターンの並列探索

先述したように、上位 k パターン制約を導入した RP-growth では、候補リストに基づく最小サポート上昇法が用いられる。その際、並列化における主なオーバーヘッドとしてはグローバル最小サポートの更新、取得、そしてグローバル候補リストへの候補パターンのマージが挙げられる。グローバル候補リストの更新処理は、リストのソート、閾値となる k 番目のパターンの関連度の計算、閾値未満のパターンの破棄、グローバル最小サポートの更新の順で行い、処理コストが大きい。

これらのオーバヘッドを抑制するために、最小サポートの上昇タイミングを利用しグローバル変数の更新を必要最小限に抑える。また、グローバル候補リスト内には等しい関連度を持つパターンが複数存在しうることを考慮しなければならない。

具体的には、まずグローバル候補リスト内での更新処理を短くするために、各スレッドで訪問したパターンの中でも最小サ

Algorithm 1 並列上位 k RP-growth

```
 1: procedure ParRPGrowth(trans, minSup)
 2:     globalMinSup := minSup
 3:     globalPattList := Vector()
 4:     initFpTree := Build(trans)
 5:     for all item in initFpTree.branches do
 6:         patt := Pattern(item)
 7:         if patts.workload ≧ threshold then
 8:             GenerateTask(ParIterRPGrowth)
 9:         else
10:             GenerateTask(SeqIterRPGrowth)
11:         end if
12:     end for
13: end procedure
```

Algorithm 2 並列上位 k RP-growth 分割タスク

```
 1: procedure ParIterRPGrowth(fpTree, patt, minSup)
 2:     condTrans := Traverse(fptree, item)
 3:     condFptree := Build(condTrans, minSup)
 4:     patt.relScore = condFpTree.getRelScore()
 5:     Update(pattList, patt)
 6:     patts := Vector()
 7:     for all item in condFpTree.branches do
 8:         branchPatt := Pattern(patt, item)
 9:         patts.push(branchPatt)
10:     end for
11:     dividedPatts := FFD(patts)
12:     for all patts in dividedPatts do
13:         if patts.workload ≧ threshold then
14:             GenerateTask(ParIterRPGrowth)
15:         else
16:             GenerateTask(SeqIterRPGrowth)
17:         end if
18:     end for
19: end procedure
```

ポートの上昇に貢献するパターン（貢献パターン）のみをマージ対象とする。貢献パターンは、グローバル候補リスト内の k 番目のパターンの関連度以上の候補パターンと定義される。

そして、グローバル最小サポートの上昇に必要なパターン数を L としたときに、この L をスレッド間で共有し、グローバル最小サポートより大きい上界サポートを持つパターンが見つかれば、L をデクリメントする。L が 0 になったときがグローバル最小サポートが上昇するタイミングであり、グローバル候補リストとローカル候補リストをマージする。グローバル最小サポートの上昇に必要なパターン数 L は、グローバル候補リスト内の k 番目のパターンの最小サポート更新値より真に大きいパターンの数を S としたとき、$L = k - S$ により算出される。

グローバル候補リストへのアクセスは排他ロック制御を用いており、この候補リストをあるスレッドが参照している間に他のスレッドは参照できない。一方、グローバル最小サポートは探索空間の削減に使用される変数であるため、より頻繁に参照される。本研究では、グローバル最小サポートのデータ型として Rust のアトミック型を用いており、値の更新処理をアトミック操作で実行できる。さらに、アトミック型はロックフリーで複数スレッドから読み書きでき、そのコストも小さい。具体的

Algorithm 3 並列上位 k RP-growth 逐次タスク

```
 1: procedure SeqIterRPGrowth(fpTree, patts, minSup)
 2:     localPattList := Vector()
 3:     for all patt in patts do
 4:         item := patt[0]
 5:         condTrans := Traverse(fpTree, item)
 6:         condFpTree := Build(condTrans, minSup)
 7:         patt.relScore := condFpTree.getRelScore()
 8:         IterRPGrowth(condFpTree, patt, minSup)
 9:     end for
10:     Update(globalPattList, foundPatts)
11: end procedure
12:
13: function IterRPGrowth(fpTree, patt)
14:     localPattList.push(patt)
15:     if patt.upperSup > globalMinSup then
16:         globalNumRequiredPatts −= 1
17:     end if
18:     if globalNumRequiredPatts ≦ 0 then
19:         Update(globalPattList, localPattList)
20:     end if
21:     for all item in fpTree.branches do
22:         patt := Pattern(item)
23:         condTrans := Traverse(fpTree, item)
24:         condFpTree := Build(condTrans, globalMinSup)
25:         patt.relScore := condFpTree.getRelScore()
26:         IterRPGrowth(condFpTree, patt)
27:     end for
28: end function
```

に Algorithm 1 で示す。入力はトランザクションデータベース *trans* と出力パターン数 k である。タスクの粒度を調節する分割閾値 *threshold* は一定値で与える。初回の FP-tree を並列処理で構築し（行 4）、各分岐パターンに対してタスク量を推定し、分割閾値以上なら *ParIterRPGrowth* 関数を実行する分割タスクを生成する（行 8）。分割閾値未満なら *SeqIterRPGrowth* 関数を実行する逐次タスクを生成する（行 10）。

ParIterRPGrowth 関数の疑似コードを Algorithm 2 に示す。分割タスクで訪問中のパターンは関連度が高いパターンであることが多いため、訪問ごとにグローバル候補リストとマージしている (行 5)。この *Update* 関数は、先述した更新処理を行っている。また、最小サポートの上昇に必要なパターン数 *globalNumRequiredPatts*（先述の L に相当）も算出し直す。生成した分岐パターン集合を *FFD* 関数によって均等に分割する (行 11)。その後、分割された各分岐パターン集合に対して、分割閾値と比較し再び分割タスクまたは逐次タスクのどちらかを実行する（行 12–18）。

SeqIterRPGrowth 関数の疑似コードを Algorithm 3 に示す。逐次タスク内では、ローカル候補リストを持つ（行 2）。各分岐パターンは逐次的に実行され、訪問した貢献パターンは一時的にローカル候補リストに格納される（行 14）。また、訪問したパターンの上界サポートとグローバル最小サポートが毎回比較され、グローバル最小サポートより大きい場合（行 15）、グローバル最小サポートの上昇に必要なパターン数 *globalNumRequiredPatts* がデクリメントされる（行 16）。全

Algorithm 4 並列最良カバー RP-growth

```
 1: procedure ParRPGrowth(trans, initMinSup)
 2:     globalMinSupTable := Vector(initMinsup; len(trans))
 3:     globalPattTable := Vector(Vector(); len(trans))
 4:     initFpTree := Build(trans, initMinSup)
 5:     for all item in initFpTree.branches do
 6:         patt := Pattern(item)
 7:         if patts.workload ≧ threshold then
 8:             GenerateTask(ParIterRPGrowth)
 9:         else
10:             GenerateTask(SeqIterRPGrowth)
11:         end if
12:     end for
13: end procedure
```

Algorithm 5 並列最良カバー RP-growth 分割タスク

```
 1: procedure ParIterFPGrowth(fpTree, patt, minSup)
 2:     item := patt[0]
 3:     condTrans := Traverse(fpTree, item)
 4:     tranIDs := fpTree.getTranIDs(item)
 5:     minSup := LeastMinSup(globalMinSupTable, tranIDs)
 6:     condFpTree := Build(condTrans, minSup)
 7:     patt.relScore := condFpTree.getRelScore()
 8:     Update(globalPattTable, patt)
 9:     patts := Vector()
10:     for all item in condFpTree.branches do
11:         branchPatt := Pattern(patt, item)
12:         patts.push(branchPatt)
13:     end for
14:     dividedPatts := FFD(patts)
15:     for all patts in dividedPatts do
16:         if patts.workload ≧ threshold then
17:             GenerateTask(ParIterRPGrowth)
18:         else
19:             GenerateTask(SeqIterRPGrowth)
20:         end if
21:     end for
22: end procedure
```

スレッドで共有されるグローバル最小サポートの上昇に必要なパターン数が 0 になったら、ローカル候補リストをグローバル候補リストとマージし、更新処理を実行する（行 19）。これで即座に訪問したパターンを用いて最小サポートを上昇させることができ、無駄な訪問パターン数の増大を抑制する。

3.4 最良カバー制約における並列探索

最良カバー制約を導入する RP-growth では、上位 1 パターンの候補リストが正トランザクションの数だけ用意され、各々に最小サポート上昇法が適用される。グローバル候補テーブル全体に排他ロック処理はせず、貢献パターンがカバーするトランザクションごとに設けられたグローバル候補リストへ個別に排他ロック処理を行う。また、各候補リストの最小サポートを別のテーブルとすることで、各最小サポートの取得と更新がロックフリーで行える。グローバル候補リストの更新処理では、最良カバー制約と飽和制約を満たすように追加する候補パターンの関連度や包含関係を比較し、更新する。

具体的な方法を Algorithm 4 に示す。入力はトランザクションデータベース *trans* である。タスクの粒度を調整する分割閾値 *threshold* は一定値で与えられる。初回の FP-tree を並列処理で構築（行 4）した後、各分岐パターンに対してタスク量を推定し、分割閾値以上なら *ParIterRPGrowth* 関数を実行する分割タスクを生成（行 8）、分割閾値未満なら *SeqIterRPGrowth* 関数を実行する逐次タスクを生成する（行 10）。

Algorithm 6 並列最良カバー RP-growth 逐次タスク

```
 1: procedure SeqIterRPGrowth(fpTree, patts, minSup)
 2:     localPattList := Vector()
 3:     for all patt in patts do
 4:         item := patt[0]
 5:         condTrans := Traverse(fpTree, item)
 6:         tranIDs := fpTree.getTranIDs(item)
 7:         minSup := LeastMinSup(globalMinSupTable, tranIDs)
 8:         condFpTree := Build(condTrans, minSup)
 9:         patt.relScore := condFpTree.getRelScore()
10:         IterRPGrowth(condFpTree, patt)
11:     end for
12:     Update(globalPattList, localPattList)
13: end procedure
14:
15: function IterRPGrowth(fpTree, patt)
16:     if patt.minSup > globalMinSup then
17:         Update(globalPattList, patt)
18:     else if patt.upperSup = globalMinSup then
19:         localPattList.push(patt)
20:     end if
21:     for all item in fpTree.branches do
22:         patt := Pattern(patt, item)
23:         condTrans := Traverse(fpTree, item)
24:         tranIDs := fpTree.getTranIDs(item)
25:         minSup := LeastMinSup(globalMinSupTable, tranIDs)
26:         condFpTree := Build(condTrans, globalMinSup)
27:         patt.relScore := condFptree.getRelScore()
28:         IterRPGrowth(condFpTree, patt)
29:     end for
30: end function
```

ParIterRPGrowth 関数の疑似コードを Algorithm 5 に示す。探索中の最小サポートはグローバル最小サポートテーブル *globalMinSupTable* から算出する（行 5）。具体的には、パターンがカバーするトランザクションに相当する最小サポートを参照し、それらの最小値を最小サポートとする。分割タスクで訪問中のパターンは関連度が高いパターンであることが多いため、訪問ごとにグローバル候補リストとマージし、更新処理を実行する（行 8）。生成した分岐パターン集合を *FFD* 関数によって均等に分割する（行 14）。その後、分割された各分岐パターン集合に対して、分割閾値と比較し再び分割タスクまたは逐次タスクのどちらかを実行する（行 15–21）。

SeqIterRPGrowth 関数の疑似コードを Algorithm 6 に示す。逐次タスク内では、ローカル候補リストを持つ（行 2）。各分岐パターンは逐次的に実行される。また、訪問したパターンの上界サポートとグローバル最小サポートが毎回比較され、グローバル最小サポートより大きい場合（行 16）、即座にグローバル候補リストにマージし、更新処理を実行する（行 17）。グローバル最小サポートと同じ場合（行 18）はローカル候補リス

表 3: データセットとその構成

データセット	トランザクション数	アイテム数
20news	17930	5666
anneal	812	93
audiology	216	148
australian-credit	653	125
german-credit	1000	112
heart-cleveland	296	95
hepatitis	137	68
hypothyroid	3247	88
kr-vs-kp	3196	73
lymph	148	68
mushroom	216	199
primary-tumor	653	31
soybean	1000	50
splice-1	296	287
tic-tac-toe	137	27
vote	3247	48
zoo-1	3196	36

トに追加（行 19）され、それ以外は破棄する。これによって即座に訪問したパターンを用いて最小サポートを上昇させることができ、無駄な訪問パターン数の増大が抑制される。

4 実　　験

4.1 データセットと実行環境

提案手法のスケーラビリティを示すために、20news（`http://people.csail.mit.edu/jrennie/20Newsgroups/`）から 3 個のデータセット、cp4im（`http://dtai.cs.kuleuven.be/CP4IM/datasets/`）から 16 個のデータセットに対して提案手法を適用する。各データセットの構成を表 3 に示す。20news データにおける各ニュースグループの記事は前処理され、記事中の単語をアイテムとする 17,930 個のトランザクションデータとなる [6]。そして、各ニュースグループについて、そのニュースグループに属するトランザクションに正ラベル、残りに負ラベルが付けられる（結果として 20 通りのラベル付きトランザクションデータが得られる）。cp4im はクラスラベル付きトランザクションデータベースであり、様々な正例率と密度のデータセットからなる。実験では, 64 コアを搭載する Ryzen Threadripper 3990X (2.9GHz) の CPU、RAM が 256GB、そして OS が Ubuntu 20.04 から構成されている計算機を用いる。

4.2 タスク量の粒度を決定する最適な分割閾値の調査

提案手法の性能とスケーラビリティを示す前に、タスクの粒度を調整するための分割閾値について調査する。また、本研究では、ユーザが適用するデータセットごとに特化された閾値を調査するということを避けるため、多くのデータセットで共通して使用することのできる一定の閾値を選択し、次の性能評価の実験に利用する。分割閾値の調査は上位 k パターン発見における RP-growth の並列化と最良カバー制約における RP-growth の並列化のそれぞれで行う。特に、実行時間の長い条件においてより効果的なものを優先して選択する。

まず、上位 k パターン制約を導入した RP-growth の並列化における最適な分割閾値を調査する。タスク量推定式から算出される値を考慮して 10^{-8} から 10^{-1} までの範囲で実行時間の短くなるような分割閾値を探索する。また、実行条件として関連度の指標が χ^2 値、指定パターン数が $k = 1000$、そして使用スレッド数は 64 スレッドという条件を試す。さらに、各実行条件に対して 5 回実行し、実行時間の平均と標準誤差を求める。

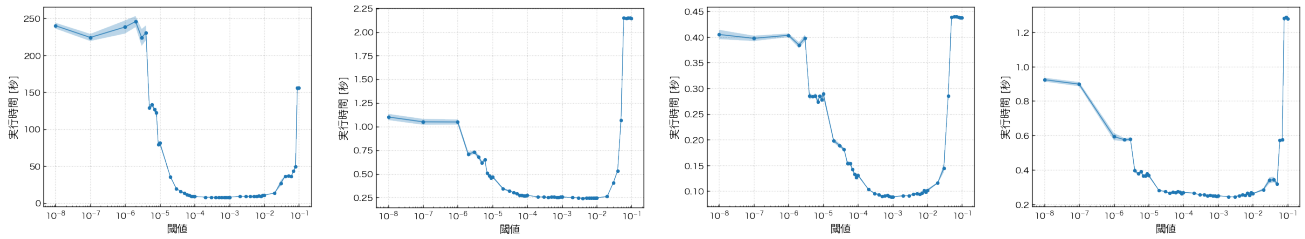
並列化の効果が大きかった全体的に実行時間が長い場合のグラフを図 2 に示す。青いマーカーが実行時間の平均を示し、水色で塗られた範囲が標準誤差である。グラフの横軸がタスクの粒度を調整する分割閾値であり、値が小さいほどより探索木がより細かく分割され、値が大きいほど探索木がより粗く分割される。図 2 (a) が示すように、分割閾値が小さいとタスク生成やパターンのマージなどのオーバーヘッドが大きくなり実行時間が長くなっている。一方、分割閾値が大きいとロードバランスが悪化し、実行時間が長くなると考えられる。図 2 では、10^{-4} から 10^{-2} までの範囲で実行時間が短いことが分かる。その他の多くの実行条件では探索時間が短いものが多いため、特に効果的に作用する上述の実行条件で有効な分割閾値を優先し、多くのデータセットで効果的な値として 4×10^{-4} を選択する。

次に最良カバー制約における RP-growth の並列化での最適な閾値の調査を行う。最良カバー制約での実行条件として、関連度の指標を χ^2 値、スレッド数を 64 スレッドとする。特に効果的に作用する条件となった実験結果のグラフを図 3 に示す。上位 k パターン発見における RP-growth よりも制約の厳しさにより探索時間が長くなる傾向があり、トレードオフの最適化が探索時間に影響しやすいことが分かる。図 3 (a) では、10^{-4} 以下で実行時間が短くなっている。この実行条件では、訪問パターン数が他の実行条件より多くなるため、タスクの粒度が大きくなりやすく、より多くの分割が必要であると思われる。一方、図 3 (b) と 3 (c) では、10^{-2} 以下から 10^{-5} あたりまで実行時間が短くなっている。これらのことから、多くのデータセットで効果的な値として 1×10^{-4} を選択する。

4.3 速 度 向 上

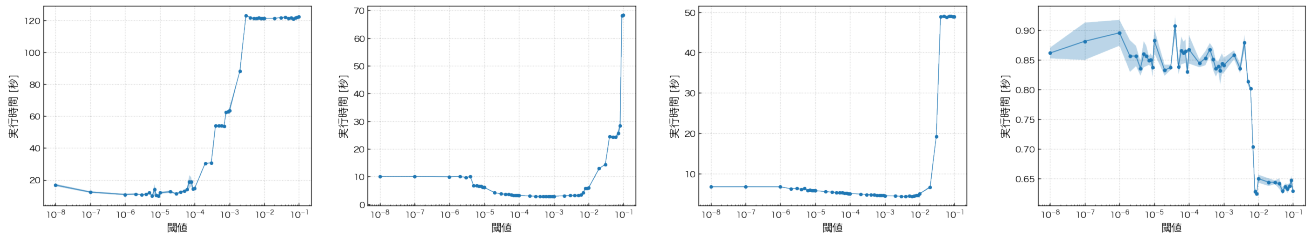
次に、前の実験で選択した効果的な閾値を用いて、提案手法の性能とスケーラビリティの評価を行う。提案手法の性能では、スレッド数を上げることによってどれだけ探索時間が短くなるかを評価する。スケーラビリティに対しては、あるスレッド数における探索時間の 1 スレッドでの探索時間に対する比率を「速度向上」とすることでどれだけ理想に近い速度向上を達成しているかを評価する。

まず、上位 k パターン発見における RP-growth の並列化に対する性能の調査を行う。具体的には、閾値を 4×10^{-4}、関連度の指標として χ^2 値、指定パターン数を $k = 1000$、スレッド数を $1, 2, 4, 8, 16, 32, 64$ と変化させたときの実行時間の推移について調べる。1 スレッドでの実行時間が 1 秒以上であるグラフを図 4 に示す。これらは探索時間が長いことから、訪問パターン数が多い場合はよりタスクの粒度を細かくでき、多くの実行条件でスレッド数に比例して実行時間が短くなっている。しかし、図 4 (d) では 32 スレッドから 64 スレッドで実行時間が短くなっていない。これは、逐次処理となっている FP-tree



(a) anneal (b) german-credit (c) heart-cleveland (d) kr-vs-kp

図 2: 分割閾値に対する実行時間の推移（実行時間の長いグループ）, 上位 k パターン, $k = 1000$, 関連度: χ^2



(a) 20news（comp.graphics） (b) anneal (c) australian-credit (d) splice-1

図 3: 分割閾値に対する実行時間の推移（実行時間の長いグループ）, 最良カバー, 関連度: χ^2

構築部分がボトルネックとなっていることが考えられる。アムダールの法則に従い、逐次処理部分をオーバーヘッドが大きくならないように並列化する必要があると考えられる。

次に、スケーラビリティの実験結果を図 5 に示す。図 5 (b) の anneal データセットは 20news と比べると小さく、探索時間が長いためロードバランスが良く 64 スレッドでも理想に近い速度向上となった。また、図 5 (c) の german-credit は探索時間は anneal と比べると短いが、64 スレッドまで速度向上を維持している。図 5 (d) の splice-1 は、32 スレッドまでは速度向上を維持するが、64 スレッドで低下している。これは、グローバル候補リストへのマージによる排他ロック待ちや条件付きトランザクション集合取得の処理時間が影響したと思われる。

次に、最良カバー制約を導入した RP-growth の並列化に対する性能を評価する。具体的には、閾値を 1×10^{-4}、関連度の指標として χ^2 値、スレッド数を $1, 2, 4, 8, 16, 32, 64$ と変化させたときの実行時間の推移を調べる。1 スレッドでの実行時間が 1 秒以上であるグラフを図 6 に示す。より厳しい制約であるため、探索時間が多くなりオーバーヘッドの影響が小さくなることで全般的に並列化の効果が向上する。しかし、図 6 (b) と図 6 (d) では 32 スレッドから 64 スレッドで実行時間が短くなっていない。これは、上位 k パターン発見と同様に、逐次処理である FP-tree 構築部分がボトルネックになったと考えられる。

次に、スケーラビリティに関する実験結果を図 7 に示す。トランザクションの多い 20news （comp.grapchis）のようなデータセットでも探索時間が長い場合は理想的な速度向上を達成した。splice-1 は、32 スレッドまでは速度向上を維持しているが、64 スレッドで低下した。FP-tree 構築による逐次処理部分のボトルネックや図 3 (d) から分かるように、最適な閾値から外れていることが要因であると思われる。

5 ま と め

本研究では、関連パターンの冗長性を除く上位 k パターン制約と最良カバー制約を導入した RP-growth の並列化を行った。候補パターンの探索木に対してタスク量推定を用いることで、より細かい粒度のタスクを生成しつつ、スケジューリングコストを抑えるトレードオフの最適化を行った。また、共有メモリ型システムでロードバランスの向上に大きく貢献する work-stealing を用いることでスケーラビリティの向上を行った。実験の結果、タスク量推定によってトレードオフの最適化が作用していることを示した。多くの実行条件で使用できる一定値に固定した閾値を用いて、探索空間が増大するような実行条件において理想に近い速度向上を達成した。今後の課題として、探索の序盤における FP-tree 構築の並列化によるボトルネックの解消や、飽和パターン [10] のみを探索するアルゴリズムの並列化による実行不可条件の解決が挙げられる。

文 献

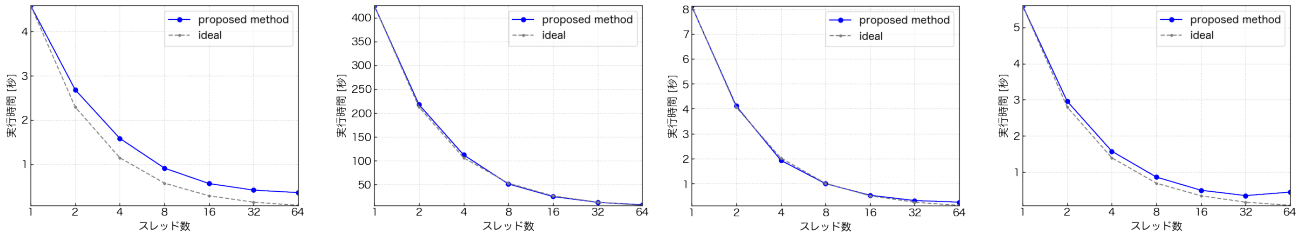
[1] R. Agrawal and R. Srikant: Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (VLDB-94), pp. 487-–499, 1994.

[2] C. Borgelt: An Implementation of the FP-growth Algorithm. Proc. of the 1st Int'l Workshop on Open Source Data Mining (OSDM-05), pp. 1–5, 2005.

[3] J. Han, J. Pei, and Y. Yin: Mining Frequent Patterns without Candidate Generation. ACM SIGMOD Record, Vol. 29, Issue 2, pp. 1–12, 2000.

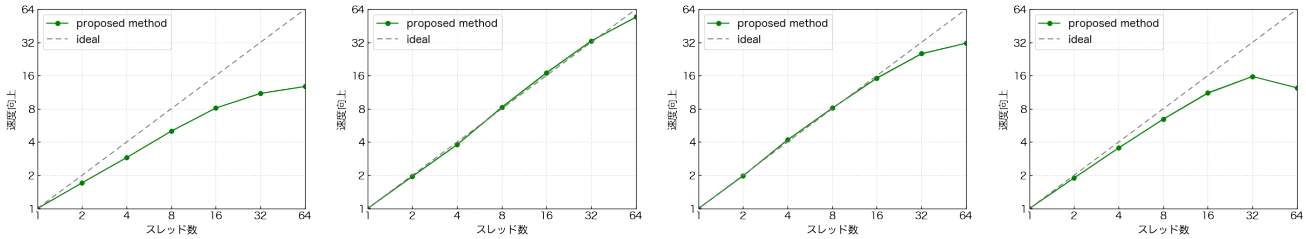
[4] J. Han, J. Wang, Y. Lu and P. Tzvetkov: Mining Top-K Frequent Closed Patterns without Minimum Support. Proc. of the 2002 IEEE Int'l Conf. on Data Mining (ICDM-02), pp. 211–218, 2002.

[5] Y. Kameya: An Exhaustive Covering Approach to Parameter-Free Mining of Non-redundant Discriminative Itemsets. Proc. of Int'l Conf. on Big Data Analytics and



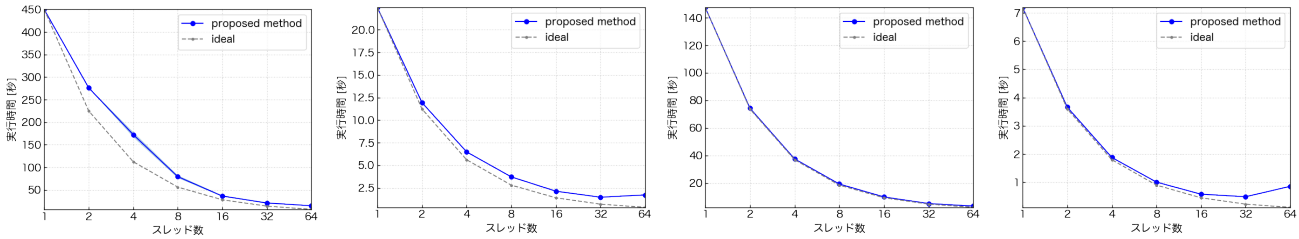
(a) 20news（comp.graphics） (b) anneal (c) german-credit (d) splice-1

図 4: スレッド数に対する実行時間の推移（実行時間の長いグループ）, 上位 k パターン, $k = 1000$, 関連度: χ^2



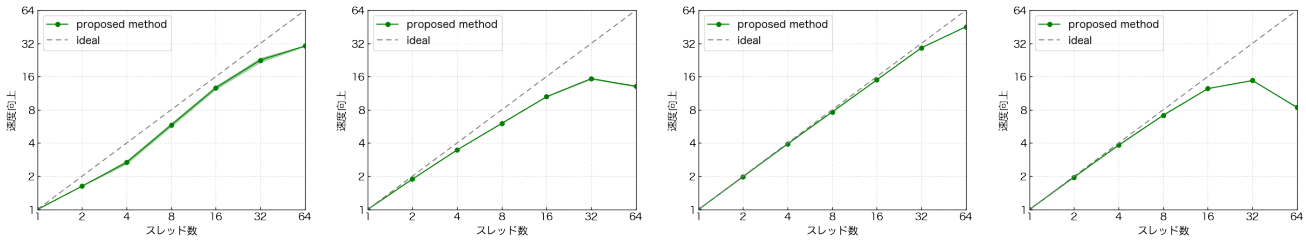
(a) 20news（comp.graphics） (b) anneal (c) german-credit (d) splice-1

図 5: スレッド数に対する速度向上（実行時間の長いグループ）, 上位 k パターン, $k = 1000$, 関連度: χ^2



(a) 20news（comp.graphics） (b) 20news（talk.politics.guns） (c) anneal (d) splice-1

図 6: スレッド数に対する実行時間の推移（実行時間の長いグループ）, 最良カバー, 関連度: χ^2



(a) 20news（comp.graphics） (b) 20news（talk.politics.guns） (c) anneal (d) splice-1

図 7: スレッド数に対する速度向上（実行時間の長いグループ）, 最良カバー, 関連度: χ^2

Knowledge Discovery (DaWaK-16), pp. 143–159, 2016.

[6] 亀谷由隆, 佐藤泰介: 最小サポート上昇法に基づく上位 k 関連パターン発見. 人工知能学会データ指向構成マイニングとシミュレーション研究会予稿集, SIG-DOCMAS-B101-4, 2011.

[7] N. Matsakis: Rayon: Data Parallelism in Rust, `http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/`, 2015.

[8] K. Sakurai and Y. Kameya: Shared-Memory Parallelization of FP-growth with Dynamic Load Estimation and Balancing. Proc. of the 12th Int'l Workshop on Computational Intelligence and Applications (IWCIA-21), 2021.

[9] D. Trabold and H. Grosskreutz: Parallel Subgroup Discovery on Computing Clusters – First Results. Proc. of the 2013 IEEE Conf. on Big Data, pp. 575–579, 2013.

[10] T. Uno, T. Asai, Y. Uchida, and H. Arimura: An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases. Proc. of Int'l Conf. on Discovery Science (DS-04), pp. 16–31, 2004.

[11] K.-M. Yu, J. Zhou, and W. C. Hsiao: Load Balancing Approach Parallel Algorithm for Frequent Pattern Mining. In V. Malyshkin, editor, Parallel Computing Technologies, pp. 623–631, 2007.

[12] O. R. Zaïane, M. El-Hajj, and P. Lu: Fast Parallel Association Rule Mining without Candidacy Generation. Proc. of the 2001 IEEE Int'l Conf. on Data Mining (ICDM-01), pp. 665–668, 2001.