DEIM2022 J34-2(day3 p37)

# 高効用アイテムセットマイニングの高効率な並列化手法とその評価

木村　元紀† 　合田　和生†† 　Rage Uday Kiran††† 　喜連川　優††

† 東京大学 情報理工学系研究科　〒 113–8656 東京都文京区本郷 7–3–1
†† 東京大学 生産技術研究所　〒 153–8505 東京都目黒区駒場 4–6–1
††† 会津大学 情報システム学部門　〒 965–8580 福島県会津若松市一箕町鶴賀
E-mail: †{kimura-g,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp, ††udayrage@u-aizu.ac.jp

**あらまし**　高効用アイテムセットマイニング（High-utility itemset mining）は、アイテムの量と利得の両方を考慮した新しいマイニング問題であり、幅広い知識発見タスクに対して応用される。この問題は以前からよく知られている頻出パターンマイニング問題の一般化とみなすことができる。しかしこの一般化により、効率の良い探索の鍵であった単調性が失われるため、この問題は技術的に困難である。これまでの多数の研究により、高効用アイテム集合を探索する効率の良いアルゴリズムが提案され、成功を収めてきた。これらのアルゴリズムの改良と並行して、本論文では我々が開発した高効用アイテムセットマイニング向けの効率的な並行化手法である DPHIM を提案する。DPHIM は独自の動的なタスク分解戦略によりハードウェアリソースを最大限に活用することができる。我々の集中的かつ広範な実験により、様々なデータセットやパラメータにおいて、DPHIM は最適に調整された逐次実行よりも大幅に（最大で 33.9 倍）高速に動作することが確認された。

**キーワード**　並列処理, データマイニング, 先進ハードウェア

## 1　はじめに

データマイニングは大規模なデータから有用な知識を取り出す手法のことである。特に高効用データマイニングは新しい重要なデータマイニング問題の一つである [8,17]。高効用データマイニングの目的は量的トランザクションデータベースから効用の高いアイテム集合を抽出することである。このデータベースでは各アイテムは外部効用（例: 単価）に紐づけられ、トランザクションは各アイテムについて内部効用 (例: 購入量) を持つ。この問題のフレームワークは柔軟で、様々な問題に適応することができる。実際これまでにクリックストリーム解析 [21] やクロスマーケティング [5] などに用いられてきた。また、高効用アイテムセットマイニングは時系列高効用アイテムセットマイニング [9] や高効用逐次パターンマイニング [26]、高効用ストリームマイニング [15] などの問題に拡張していくことができる。

高効用アイテムセットマイニングは伝統的に知られる頻出アイテムセットマイニング [1,19] を一般化した問題である。頻出アイテムセットマイニングは、反単調性と downward-closure property（あるアイテム集合が頻出でないなら、その全ての上位集合は頻出でない）という二つの性質を持ち、多くのアルゴリズムはこの性質を利用することで大幅に探索空間を削減している。しかしながら、高効用アイテムセットマイニングに一般化するとこれらの性質は保たれず、代替となる効率的なアルゴリズムの考案が多くの研究によってなされてきた。最初の大きな発見は Two-phase [18] によって導入された transaction weighted utilization(TWU) である。TWU は一般化によって失われた反単調性と downward-closure property を持ち、探索空間を効率的に削減することができる。実際これ以降 TWU を用いた多くのアルゴリズムが提案されてきた。二つ目の大きな進歩は HUI-Miner [17] によって導入された utility-list である。utility-list は各アイテム集合の効用や膨大な数の探索候補を削減し探索効率を向上するためのヒューリスティックな情報を集約した構造体である。utility-list もこれ以降の多くのアルゴリズムで用いられている。

これらのアルゴリズムを改善する別のアプローチとして並列実行がある。本論文では我々が開発した現代的なハードウェア環境で高効用アイテムセットマイニングの効率の良い実行手法である DPHIM について紹介する。近年のハードウェアは多数のプロセッサ [4] やメモリチップ [20] を備えており、これらを最大限に活用することが高速な実行の鍵である。DPHIM はマイニングタスクを多数のサブタスクに分解し、これらを並列に実行する。そのため、DPHIM はプロセッサの能力やメモリ帯域などのハードウェアのリソースを最大限に利用することができ、これにより非常に高速なマイニング処理が可能になる。本稿では集中的かつ広範な実験により様々なデータセットや実験設定下で DPHIM が最適化した逐次実行と比較しても大幅な性能向上を示すことを検証した。

## 2　背景および関連研究

### 2.1　高効用アイテムセットマイニング

高効用アイテムセットマイニングは新しいデータマイニング問題の一つである [8]。現在この問題の定式化として広く用いられているものは次に紹介する Yao らによるものである [25]。

［定義 1］ (量的トランザクションデータベース) $I = \{i_1, i_2, \ldots, i_m\}$ をアイテムの有限集合とする。量的トラン

表 1: 量的トランザクションデータベースの例

| TID | トランザクション |
|---|---|
| $T_1$ | (a, 1) (b, 2) (c, 2) |
| $T_2$ | (a, 2) (b, 2) (e, 1) |
| $T_3$ | (a, 1) (c, 3) (d, 1) (e, 2) |
| $T_4$ | (b, 4) (c, 2) (e, 1) |
| $T_5$ | (b, 2) (c, 2) (d, 3) (f, 1) |

表 2: 外部効用の例

| アイテム | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| 外部効用 | 4 | 3 | 1 | 5 | 2 | 1 |

ザクションデータベースはトランザクションの有限集合 $D = \{T_1, T_2, \ldots, T_n\}$ であり、各トランザクション $T_j$ は $I$ に含まれるアイテムの集合である。各アイテム $i \in I$ には外部効用と呼ばれる正数 $p(i)$ が対応する。また $i \in T_j$ に対して正数 $q(i, T_j)$ が対応し、これを内部効用と呼ぶ。

［定義 2］ (アイテムおよびアイテム集合の効用) あるトランザクション $T_j$ 中のアイテム $i$ の効用 $u(i, T_j)$ は $p(i) \times q(i, T_j)$ で定義される。また、$T_j$ 中のアイテム集合 $X$ の効用は $u(X, T_j) = \sum_{i \in X} u(i, T_j)$ である。さらに、アイテム集合 $X$ の効用は、$g(X)$ を $X$ を含むトランザクションの集合として、$u(X) = \sum_{T_j \in g(X)} u(X, T_j)$ となる。

量的データベースと外部効用が表 1 と 2 のようになっていると仮定する。この時トランザクション $T_2$ 中のアイテム $a$ の効用は $u(a, T_2) = 4 \times 2 = 8$ であり、$T_2$ 中のアイテム集合 $\{a, b\}$ の効用は $u(\{a, b\}, T_2) = u(a, T_2) + u(b, T_2) = 8 + 6 = 14$ である。また、データベース全体での $\{a, b\}$ の効用は $u(\{a, b\}) = u(\{a, b\}, T_1) + u(\{a, b\}, T_2) + u(\{a\}, T_3) + u(\{b\}, T_4) + u(\{b\}, T_5) = 10 + 14 + 4 + 12 + 6 = 46$ である。

［定義 3］ (高効用アイテムセットマイニング問題) 最小効用を $minutil$ とすると、アイテム集合 $X$ が高効用アイテム集合となるのは $u(X) \geqq minutil$ の時である。高効用アイテムセットマイニング問題とは与えられた量的トランザクションデータベース中の全ての高効用アイテム集合を見つけることである。

### 2.2 初期の高効用アイテムセットマイニング向けアルゴリズム

UMining [24] のような初期のアルゴリズムはヒューリスティクスによって探索空間を削減していたので、全ての高効用アイテム集合を見つける保証ができなかった。そこで 2005 年に Liu らは transaction weighted utilization (TWU) を導入した [18]。

［定義 4］ Transaction weighted utilization (TWU) あるトランザクション $T_j$ の効用を $TU(T_j) = \sum i \in T_j u(i, T_j)$ とする。この時アイテム集合 $X$ の TWU は $TWU(X) = \sum_{T_j \in g(X)} TU(T_j)$ である。

表 1, 2 の例では、トランザクション $T_1, T_2, T_3, T_4, T_5$ の効用はそれぞれ $12, 16, 16, 16, 24$ であり、$\{a\}$ の TWU は $TWU(\{a\}) = TU(T_1) + TU(T_2) + TU(T_3) = 12 + 16 + 16 = 44$ である。

**Algorithm 1** one-phase アルゴリズムの一般化フレームワーク

**Input:** $D$: transaction database, $minutil$: a threshold
**Output:** the set of high-utility itemsets
1: itemTWU, $I^* \leftarrow$ CalcTWU$(D, minutil)$
2: $A_0, E_0, K_0 \leftarrow$ Build$(I^*, D)$
3: Search$(\emptyset, A_0, E_0, K_0, minutil)$

**Input:** $D$: a transaction database, $minutil$: a minimum utility threshold
**Output:** itemTWU: mapping from item to TWU, $I^*$: a set of item
4: **function** CalcTWU$(D)$
5: itemTWU$[item] = 0$ for each $item$ in $D$
6: **for all** $d \in D$
7: **for all** $(item, utility) \in d$
8: itemTWU$[item] \leftarrow$ itemTWU$[item] + utility$
9: $I^* = \{item \mid \text{itemTWU}[item] \geqq minutil\}$
10: sort $I^*$ by TWU ascending values
11: **return** itemTWU, $I^*$

**Input:** $I^*$, $D$
**Output:** $A$, $E$, $K$
12: **function** Build$(I^*, D)$
13: remove item $i \notin I^*$ from $D$
14: sort transactions in $D$ and delete empty transactions
15: initialize $A, E, K$ by $I^*$
16: **for all** $d \in D$
17: update $A, E, K$ by $d$
18: **return** $A, E, K$

**Input:** $P$: itemset, $A$: auxiliary data, $E$: items to explore, $K$: items to keep, $minutil$: minimum utility
19: **function** Search$(P, A, E, K, minutil)$
20: **for all Item** $x \in E$
21: $Px \leftarrow P \cup \{x\}$
22: $u(Px), A' \leftarrow$ CalcUtilAux$(Px, A)$
23: **if** $u(Px) \geqq minutil$ **then**
24: Output$(Px)$
25: $E' \leftarrow \{\text{make}(P, x, y) \mid y \in K \wedge \text{filtE}(x, y, A')\}$
26: $K' \leftarrow \{\text{make}(P, x, y) \mid y \in K \wedge \text{filtK}(x, y, A')\}$
27: **if** $E' \neq \emptyset$ **then**
28: Search$(Px, A', E', K', minutil)$

TWU は次のような特徴を持つ。

- あるアイテム集合 $X$ の TWU はその効用よりも大きいか等しい。つまり $TWU(X) \geqq u(X)$。
- TWU は反単調性を持つ。つまりアイテム集合 $X, Y$ に対して $X \subset Y$ ならば $TWU(X) \geqq TWU(Y)$。
- アイテム集合 $X$ が $TWU(X) < minutil$ を満たすとき、$X$ およびその上位集合は高効用アイテム集合にはならない。

Liu らは Apriori [1, 2] を元にした探索戦略と TWU による探索空間削減を組み合わせた Two-phase というアルゴリズムを提案している [18]。さらに、Two-phase を発展させた FUM, DCG+ [16], GPA [13], PB [12] などのアルゴリズムも提案されている。

表 3: 一般化フレームワーク (アルゴリズム 1) の具体化規則

| Generalized argument | HUI-Miner [17] | FHM [10] | EFIM [27] |
|---|---|---|---|
| $A$ | $\emptyset$ | EUCS | $(\alpha\text{-}D, su(\alpha,\cdot), lu(\alpha,\cdot))$ |
| $E$ | $Px.UL$ | ExtensionsOfP | Primary($\alpha$) |
| $K$ | $Px.UL$ | ExtensionsOfP | Secondary($\alpha$) |
| MAKE$(P,x,y)$ | construct$(P,x,y)$ | construct$(P,x,y)$ | $y$ |
| FILTE$(x,y,A)$ | $x \prec y$ | $x \prec y \wedge EUCS(x,y) \geqq minutil$ | $su(\beta,z) \geqq minutil$ |
| FILTK$(x,y,A)$ | $x \prec y$ | $x \prec y \wedge EUCS(x,y) \geqq minutil$ | $lu(\beta,z) \geqq minutil$ |
| CALCUTILAUX$(Px,A)$ | $(SUM(Px.iutils),\emptyset)$ | $(SUM(Px.iutils),$EUCS$)$ | Scan $\alpha$-$D$ to calculate $u(Px)$ and create $\beta$-$D$; Calculate $su(\beta,\cdot)$ and $lu(\beta,\cdot)$ by scanning $\beta$-$D$; return $(u(Px),(\beta\text{-}D, su(\beta,\cdot), lu(\beta,\cdot)))$; |

### 2.3 One-phase アルゴリズム

2012 年に Liu と Qu によって提案された HUI-Miner では、アイテムセットの情報を utility-list という構造にまとめることで探索候補の増加を抑制している [17]。

［定義 5］ (Utility-list) アイテム集合 $X$ の utility-list はタプル $(tid, iutil, rutil)$ の集合であり、各トランザクション $T_{tid}$ は $X$ を含んでいる。$iutil$ は $T_{tid}$ 中の $X$ の効用、つまり $u(X,T_{tid})$ であり、$rutil$ は $\sum_{i \in T_{tid} \wedge \forall x \in X. x \prec i} u(i,T_{tid})$ として定義される。ただし $\prec$ は TWU の昇順で定義される全順序である。

HUI-Miner は既存の手法の性能を大きく上回っており、今日までこれをベースにした手法が state-of-the-art である。HUI-Miner を改良したアルゴリズムとしては、FHM [10], HUP-Miner [11], EFIM [27], HUI-Miner-PR [23] などがあり、これらは two-phase アルゴリズムに対して one-phase アルゴリズムと呼ばれる。

これらの one-phase アルゴリズムは様々な枝刈り手法を用いているが、同じフレームワークを用いている。最初に各アイテムの TWU を計算し、アイテムの順序付き集合を得る。次に 1-item utility-list （もしくはそれに等価なもの）を構築する。最後に再帰的に utiliy-list を拡張することで、高効用アイテム集合を探索する。アルゴリズム 1 に一般化した one-phase アルゴリズムのフレームワークを示し、表 3 にはフレームワークを具体化する時の規則を示した。なお、表 3 では元の論文で用いられている記法を用いている。

この一般化フレームワークは 3 章で導入する並列かの基礎となるので、更に詳しく説明する。このフレームワークは 3 つのステップから構成されており、それぞれ CALCTWU (1 行目), BUILD (2 行目) and SEARCH (3 行目) である。

最初のステップである CALCTWU では、探索対象となるデータベース $D$ 全体を走査し、$D$ 中に出現する全てのアイテムに対して TWU を計算する (5-8 行目)。その後、TWU が最小効用 ($minutil$) よりも大きいアイテムだけを集めた集合 $I^*$ を構築し、TWU の昇順にソートする (9-10 行目)。

第二段階の BUILD はデータベース $D$ を再度走査し、TWU が最小効用よりも小さいアイテムと空のトランザクションを削除する。その後、1-item utility-list (もしくはそれに等価なもの) $A, E, K$ を構築する (15-17 行目)。これらの実体はアルゴリズムによって様々である。HUI-Miner では単純な utiliyt-list のみであるが、FHM では utiliy-list に加えて更に効率的な枝刈りを行うために EUCS と呼ばれる補助的な構造体を持つ。EFIM では utility-list ではなく探索候補のアイテムの単純な集合を用いるが、意味的には utility-list と等価である。これらの構造体は全てアルゴリズム 1 中の $A, E, K$ として表すことができる。

最終ステップの SEARCH は与えられたアイテム集合 $P$ のありうる全ての拡張 $Px$ に対して、$Px$ が高効用アイテム集合ならそれを出力する (20-24 行目)。更に、$Px$ の拡張が高効用でありうるなら SEARCH を再帰的に呼び出していく (25-28 行目)。この間、$A, E, K$ は段階的に更新されていき (22,25,26 行目)、$Px$ を拡張するのに用いられる。

## 3 手　　法

ここまでで述べたように、高効用アイテムセットマイニング向けのアルゴリズムは様々に改良されてきた。過去の研究ではより効率のよい枝刈り戦略を用いることでアルゴリズムの改善を試みていた。一方、本稿で紹介する DPHIM ではマイニングのタスクを動的に分解し、並列に処理することで、実行の高速化を達成した。

### 3.1 静的パーティショニング

動的並列化を導入する前に、まず静的パーティショニング戦略を簡単に導入する。静的パーティショニングは並列実行のためのよく知られた手法の一つであり、一つのタスクを予め決められた個数の独立したサブタスクに分解する。その後これらを並列に実行し、サブタスクは実行中に新たなタスクを生成することはない。静的パーティショニングは実行するべきタスクの全容が予めわかっており、そのタスクをほぼ同じ大きさの独立なサブタスクに分解しやすい場合に用いると良い性能を発揮しやすい。例えばアルゴリズム 1 の ITEMTWU や BUILD に静的パーティショニングを適用することを考える。ここではデータベース $D$ に対するスキャンを行っているので、$D$ を同じ大きさのサブデータベース $D_1, D_2, \ldots, D_n$ に最初に分割し、これらに対するスキャンを並列に実行することで静的パーティショニングを実現できる。基本的にはプロセッサの個数と同じ個数

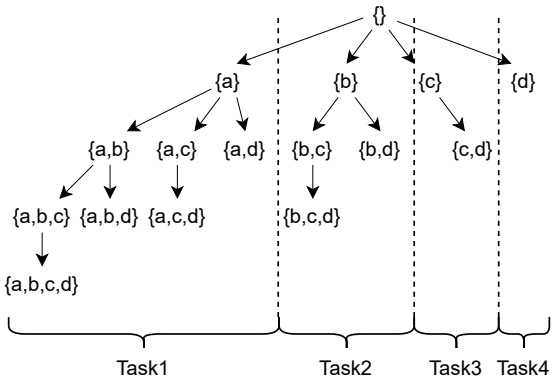



図 1: Search に静的パーティショニングを適用した例 (初期条件: $E_0 = \{a, b, c, d\}$)。図中の円一つが一つのタスクを表す。最初に $E$ を $\{a\}, \{b\}, \{c\}, \{d\}$ に分割し、それぞれを Task1,2,3,4 に割り当てる。その後 Task1 は 8 つのアイテム集合を探索するが、Task4 はただ一つのアイテム集合しか探索せず、タスクの不均衡が発生している。ただしこの例では簡単のために utility-list による枝刈りを無視している。

のサブタスクに分割し、一つのプロセッサに一つのタスクを割り当てて実行するので、並列実行によるオーバーヘッドを無視すれば実行時間はプロセッサの個数を $n$ として $1/n$ になることが期待できる。

一方アルゴリズム 1 の Search に静的パーティショニングを適用することを考える。Search では自身を再帰的に呼び出しながら探索を進めていき、実行中にヒューリスティクスによる枝刈りを行う。そのため実際に実行してみるまでタスクの全容は明らかでなく、同じ大きさのサブタスクに分割することは難しい。Search をサブタスクに分割する最も単純な方法として探索木の最初の分岐で分割する方法があるが、図 1 に示したように各タスクの大きさが大きく異なってしまう場合がある。静的パーティショニングでは各サブタスクは自身の実行が終了しても他のサブタスクの終了を待つ必要がある。そのため図 1 のようにサブタスクの不均衡が発生すると最も大きなサブタスクの実行終了を待つのにほとんどの時間を費やしてしまい、マルチプロセッサ環境の性能を最大限に活用することができない。

### 3.2 動的並列化

DPHIM(Dynamic Parallelization for High-utility Itemset Mining) では、マイニングのタスクを実行時に分解して大量の小さなタスクを生成する。分割されたタスクは各プロセッサに割り当てられ、並列に実行される。これにより DPHIM は複数のプロセッサを持つ現代的なハードウェアのリソースを最大限に活用することができる。

アルゴリズム 2 には、アルゴリズム 1 に示した one-phase アルゴリズムを動的並列化したものを示した。アルゴリズム 2 も同様に CalcTWU, Build, Search の 3 段階から構成されている。

最初の CalcTWU では元のデータベース $D$ からその部分集合である $D'$ を取り出した後、$D'$ に対して CalcTWUBody を実行するようなタスクを生成し、並列に実行する (8-13 行目)。なお、タスクはデータベース中のトランザクション一つずつに対して生成することも可能だが、タスクの粒度が細かくなりすぎると無視できないオーバーヘッドが発生するので、いくつかのトランザクションをまとめてタスクを生成している。第二段階の Build も同様にして、$D$ の部分集合 $D'$ を用いて $A, E, K$ を更新するタスクを作り、並列に実行する (23-28 行

**Algorithm 2** one-phase アルゴリズムの一般化フレームワークに対する動的並列化 (DPHIM)

```
1: function CalcTWU(D)
2:     initialize itemTWU
3:     function CalcTWUBody(D)
4:         for all d ∈ D
5:             for all (item, utility) ∈ d
6:                 itemTWU[item] ← itemTWU[item] + utility
7:     T ← ∅
8:     repeat
9:         D' ← a subset of D
10:        t ← create a task to exec CalcTWUBody(D')
11:        D ← D \ D'
12:        T ← T ∪ {t}
13:    until D = ∅
14:    Exec and wait all tasks in T
15:    I* = {item | itemTWU[item] ≧ minutil}
16:    sort I* by TWU ascending values
17:    return itemTWU, I*
18: function Build(I*, D)
19:    remove item i ∉ I* from D
20:    sort transactions in D and delete empty transactions
21:    initialize A, E, K by I*
22:    T ← ∅
23:    repeat
24:        D' ← a subset of D
25:        t ← create a task to update A, E, K by each d in D'
26:        D ← D \ D'
27:        T ← T ∪ {t}
28:    until D = ∅
29:    Exec and wait all tasks in T
30:    return A, E, K
31: function Search(P, A, E, K, minutil)
32:    function SearchX(x)
33:        Px ← P ∪ {x}
34:        u(Px), A' ← CalcUtilAndAux(Px, A)
35:        if u(Px) ≧ minutil then
36:            Output(Px)
37:        E' ← {make(P, x, y) | y ∈ K ∧ filtE(x, y, A)}
38:        K' ← {make(P, x, y) | y ∈ K ∧ filtK(x, y, A)}
39:        Search(Px, S', E', K', minutil)
40:    T ← ∅
41:    for all x ∈ E
42:        t ← Create a task to exec SearchX(x)
43:        T ← T ∪ {t}
44:    Exec and wait all tasks in T
```

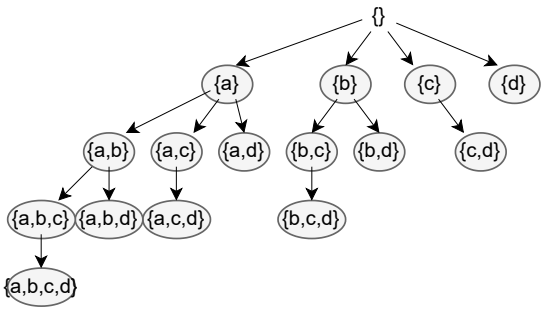



図 2: SEARCH による動的並列化の例 (初期条件: $E_0 = \{a, b, c, d\}$)。アルゴリズム 2 に従ってアイテム集合を探索する度に、その拡張を探索する新たなタスクを作成している。最終的に SEARCH のタスクは 15 個の細かなタスクに分解される。

目)。最終ステップの SEARCH では、現在探索中のアイテム集合 $P$ の可能な拡張 $Px$ を探索するために SEARCHX を実行するタスクを作る。その中では再帰的に SEARCH が呼び出されており、タスクも再帰的に作られていく (42-43 行目)。

図 2 には SEARCH がタスクを動的に分割しながら動作する例を示した。アルゴリズム 2 中の SEARCH は新たなアイテム集合を探索するために新たなタスクを一つ生成していく。最終的にこの例ではタスクは 15 個の細かなタスクに分割される。これらの分割されたタスクは手の空いているプロセッサに割り当てられ、並列に実行される。これにより現代的なアーキテクチャのプロセッサやメモリ帯域を最大限活用して探索を進めていくことが可能になる。

### 3.3 タスク制御系

DPHIM は最終的に非常に大量のタスクを生成する。本節では、これらのタスクの効率的な実行手法について説明する。

容易な手法の一つは Unix や Linux における pthread のようなスレッド機構を用いることである [14, 22]。具体的には、各タスクに対してスレッドを個別に生成すると、オペレーティングシステムによって自動的にスケジューリングされ、適切なプロセッサ時間が確保される。しかし、これらのオペレーティングシステムによって管理されるスレッドはコンテキストスイッチなどにより無視できないオーバーヘッドがある場合がある。

そこで我々はタスクを効率よく実行できるようなタスク制御系を作成した。このタスク制御系はワークスティーリング戦略 [3] を用いることにより、複数のプロセッサを備えたアーキテクチャ上で多数のタスクを効率よく実行できる。タスク制御系は一つのプロセッサに一つのスレッドを固定し、各スレッドは 1 つのワーカーとして働き、自身のタスクキューを管理する。タスクの実行中に新しいタスクが生成されると、そのタスクを実行しているスレッドが管理するタスクキューに入れられる。スレッドがタスクの実行を中断または終了する度に、スレッドは自身のキューから次のタスクを取り出し、実行する。もし自身のキューが空であれば、他のスレッドの管理するタスクキューからタスクを取り出すことで自身の実行すべきタスクを得る。これにより、利用可能なプロセッサを常にタスクの処理に用いることができる。このタスク制御による実行例を図 3 に示した。

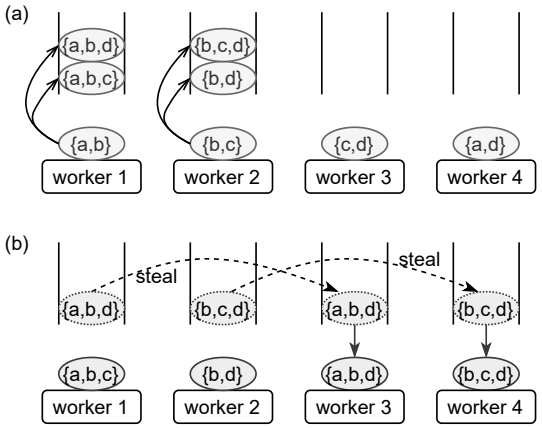
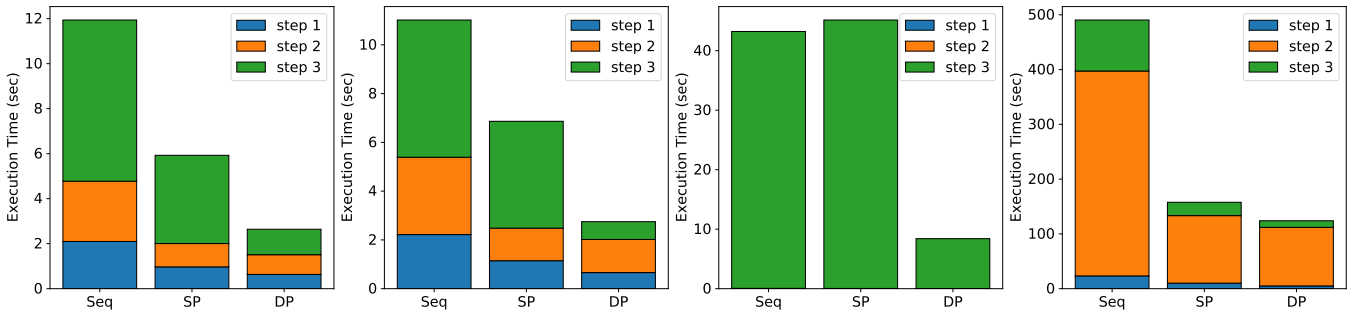



図 3: タスク制御の例。(a) ワーカー 1, 2, 3, 4 はそれぞれ $\{a, b\}$, $\{b, c\}$, $\{c, d\}$, $\{a, d\}$ を探索している。その後、ワーカー 1 と 2 は新しい候補 $\{a, b, c\}$, $\{a, b, d\}$, $\{b, d\}$, $\{b, c, d\}$ を探索するタスクを作り、自身のタスクキューに積む。(b) ワーカー 1 と 2 は $\{a, b\}$, $\{b, c\}$ を探索し終わった後に次の候補 $\{a, b, c\}$, $\{b, d\}$ を自身のタスクキューから取り出して探索する。一方、ワーカー 3 と 4 は $\{c, d\}$, $\{a, d\}$ を探索し終わるとタスクキューは空になる。そこでワーカー 1,2 のキューから $\{a, b, d\}$, $\{b, c, d\}$ を奪い、代わりに実行する。

### 3.4 並行性制御

動的並列化ではデータ競合などの並列実行に起因する問題が発生する場合がある。アルゴリズム 2 中の下線部はデータ競合が起こる箇所である。itemTWU は並列に実行されているタスク間で共有され、同時に更新される可能性がある (6 行目)。また、utility-list$(A, E, K)$ も同様に複数のタスクから同時に更新されうる (25 行目)。プログラムの安全な実行や正しい答えが得られることを保証するために、これらのデータ競合を取り除く必要がある。itemTWU は単純な整数値の配列で実装することができるため、その更新処理に `atomic_fetch_add` のような atomic 命令を用いることでデータ競合を回避することができる。データ競合を避ける最も簡単な方法の一つに mutex を用いて排他制御をするものがあるが、これは mutex で保護されている区間を同時に実行できるプロセッサが一つに限られてしまい、実行速度が低下してしまう場合がある。一方 atomic 命令を用いるとそのような問題は発生せず、高速に処理を行うことができる。

複雑な構造を持つ utility-list には atomic 命令を用いることはできない。utility-list の各要素は 1 つのトランザクションから計算されるので、要素の構築はデータ競合の恐れなく同時に実行することができる。構築された要素から utility-list を生成する箇所のみ逐次的に行うことで、mutex により保護されるクリティカルセクションを短くすることができ、効率的な実行が可能になる。




(a) Kosarak (b) Chainstore (c) BMS (d) T200N100KD1M

図 4: 並列化により削減された実行時間

表 4: Datasets and minimum utility configurations

| Dataset | # of items | # of transactions | *minutil* |
|---|---|---|---|
| Kosarak [7] | 41270 | 990002 | 1100000 |
| Chainstore [7] | 46086 | 1112949 | 500000 |
| BMS [7] | 3340 | 77512 | 2250000 |
| T200N100KD1M [6] | 100000 | 1000000 | 10000 |

## 4 実　　験

本章では DPHIM による性能改善検証のための実験を示す。

### 4.1 実験設定

実験を行ったコンピュータは 4 ソケットの Intel Xeon Gold 6252 processors (24 cores, 2.10 GHz) を備えており、各ソケットには 192GB の DRAM が付属している。また、オペレーティングシステムは Ubuntu 20.04.3 (5.4.0-88-generic Linux Kernel) である。

表 4 には、実験で用いた Kosarak, Chainstore, BMS, T200N100KD1M の 4 つのデータセットについて示した。Kosarak, Chainstore, BMS は公開されているデータリポジトリ [7] から取得した。これらは外部効用の無い実際のデータセットを元に、外部効用を後から追加したものである [17]。T200N100KD1M は SPMF dataset generator [6] を用いて生成したデータセットである。

One-phase アルゴリズムの例として FHM [10] を用いた。比較のために次の実装を用意した。

- **Seq**: いくつかの最適手法を適用した C++実装
- **SP**: **Seq** に静的パーティショニングを適用したもの
- **DP**: **Seq** をアルゴリズム 2 により動的並列化したもの

### 4.2 動的並列化 (DPHIM) による性能向上

第一に、動的並列化による性能向上が他の実行と比較した実験結果を図 4 に示した。動的並列化 (**DP**) は 4 つの異なるデータセットにおいて、最適化された逐次実行 (**Seq**) と比較して 2.47 倍から 5.15 倍高速であり、静的パーティショニングと比較しても 1.04 倍から 5.37 倍高速である。

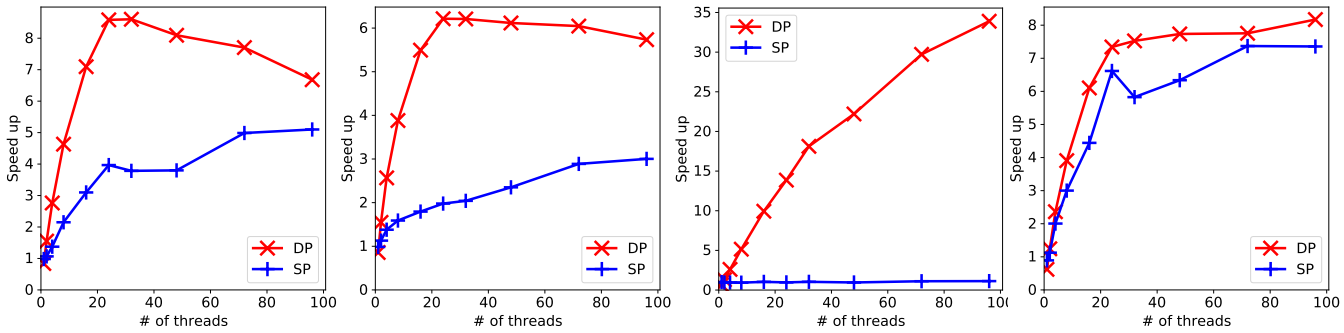
図 4(a) は Kosarak データセットで最小効用 (*minutil*) を 1100000 に設定した時の各実行形態に対する実行時間を包括的に示している。積み上げ棒グラフには実行の各段階の実行時間が積み上げられている。Step1,2,3 はそれぞれアルゴリズム 1, 2 における CalcTWU, Build, Search に対応している。最も左のバーは最適化された逐次実行 (**Seq**) であり、全体で 11.9 秒かかっている。二つ目のバーは静的パーティショニングを用いた実行 (**SP**) であり、実行時間は全体で 5.93 秒である。これは **Seq** より 2.01 倍高速である。三番目のバーは本論文で提案した動的並列化 (DPHIM) を適用した実行 (**DP**) であり、実行時間は最も短い 2.65 秒である。つまり **DP** は **Seq** より 4.51 倍速く、**SP** よりも 2.24 倍速い。

図 4(b-d) は他のデータセット (Chainstore、BMS、T200N100KD1M) での結果を示している。最小効用 (*minutil*) の値はそれぞれ 500000、2250000、10000 に設定されている。どのようなデータセットに対しても動的並列実行 (**DP**) は一貫して最適化した逐次実行 (**Seq**) 及び静的パーティショニングを適用した実行 (**SP**) よりも著しく高速である。データセットによって実行時間に特徴がある。例えば、BMS は実行のほとんどの時間を第三ステップ (Search) に費やしているが、T200N100KD1M は第二ステップ (Build) に費やしている。また、興味深い結果としてデータセットが BMS の場合に **SP** が **Seq** よりも遅くなっていることがある。これはおそらくマルチスレッド化したことによるオーバーヘッドがマルチスレッド化による高速化幅を上回ってしまったことによるものだと考えられる。一方で **DP** は常に他のどの実行手法よりも高速である。

### 4.3 スレッドスケーラビリティ

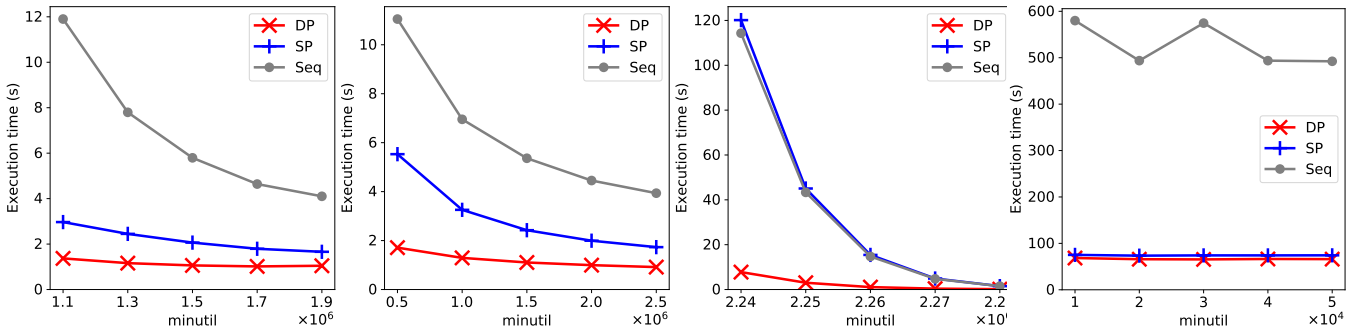
次に大量のプロセッサを活用できているかを検証するために、スレッド数のスケーラビリティに関する実験を行った。これまでの実験とは異なり、静的パーティショニングど動的並列化 (DPHIM) の実行に関して、スレッド数を変化させた場合の実行を観察した。この実験の結果を図 5 に示した。動的並列化 (**DP**) は様々なデータセットに対して最大で 6.21 から 33.9 倍の高速化を達成している。なお、ベースラインは最適化した逐次実行 (**Seq**) であることに注意されたい。

図 5(a) は Kosarak データセットを用いてスレッド数を変えながら実行したときの実行速度の向上率を示している。青色の曲線は静的パーティショニングを適用した実行 (**SP**) の結果を




(a) Kosarak (b) Chainstore (c) BMS (d) T200N100KD1M

図 5: スレッド数を変化させた時の速度向上率




(a) Kosarak (b) Chainstore (c) BMS (d) T200N100KD1M

図 6: 様々な最小効用 (*minutil*) 設定での実行時間

示しており、スレッド数が 192 の時に実行速度が最大であり、5.10 倍に達している。一方、赤色の曲線は動的並列化 (DPHIM) を適用した実行 (**DP**) の結果を示しており、実行速度は最大で 8.60 倍まで向上した。この時のスレッド数は 24 であり、これは CPU ソケット 1 つあたりのコア数に等しい。また、**DP** はスレッド数を変えても一貫して **SP** よりも高い速度向上率を達成していた。

図 5(b-d) も同様に他のデータセット (Chainstore、BMS、T200N100KD1M) に対してスレッド数を変えた時の実行速度の向上率を示している。動的並列化 (**DP**) は著しい性能向上を示しており、それぞれ最大で 6.21 倍、33.9 倍、8.17 倍になっている。また、これらの実行結果には興味深い特徴が見受けられる。BMS データセットではスレッド数が 192 になるまで常に性能向上を続けている一方で、他のデータセットではスレッド数が 24 のあたりで飽和し、それ以降は速度が低下もしくはほぼ変わっていない。これは BMS データセットの特性によるものだと考えられる。BMS データセットは実行時間のほぼ全てを第三ステップ (Search) に費やしており、そこでは各タスクは一切他のタスクとの共有変数を持たないので、各々が完全に独立して実行を進めていくことができる。つまり第三ステップは特によく並列化による速度向上の恩恵を受けることができるために、BMS では著しい性能向上を示したと考えられる。

この実験により、動的並列化 (**DP**) はスレッド数が CPU ソケット 1 つあたりのコア数に収まる程度であればスケーラブルであり、もしデータセットが BMS のような特に動的並列化に適しているものであれば 192 スレッドまでスケールするということが明らかになった。また全ての場合おいて動的並列化 (**DP**) は最適化された逐次実行 (**Seq**) や静的パーティショニング (**SP**) よりも一貫して著しく高速であった。

### 4.4 最小効用による影響

最後に、最小効用 (*minutil*) を変えながら実行したときの実験結果を図 6 に示す。動的並列化 (**DP**) は最小効用を変化させても全てのデータセットにおいて一貫して **SP** や **Seq** よりも著しく高速である。なお、この実験においては **DP** 及び **SP** のスレッド数は 24 に設定した。

図 6(a) は Kosarak データセットにおいて逐次実行 (**Seq**) と静的パーティショニング (**SP**) と動的並列化 (**DP**) を比較している。最小効用を減らしていくと実行時間が増加しているが、これは最小効用がより小さくなればなるほど探索すべき状態空間が大きくなるからである。静的パーティショニング (**SP**) は逐次実行 (**Seq**) と比較して 2.59 倍から 4.01 倍高速に実行している。一方、動的並列化 (**DP**) は更に高速な実行 (3.92 倍から 8.69 倍) を達成している。

図 6(b-d) も同様に他のデータセットに対する似たような傾向を示している。最小効用を変えたことによる影響を最もよく受けたのは BMS データセットであり、一方最も影響が少なかったのは T200N100KD1M である。静的パーティショニング (**SP**) は最適化された逐次実行 (**Seq**) に対して 0.95 倍から 7.75 倍の速度向上を示している。特に BMS データセットでは 0.95 倍と逐次実行よりも遅くなってしまっている。一方で動的

並列化 (**DP**) は一貫して逐次実行よりも高速であり、速度向上率は 4.29 倍から 14.7 倍である。これらの実験結果は動的並列化 (**DP**) が様々なデータセットに対して最小効用 (*minutil*) を変えながら実行しても一貫して著しい性能向上を達成するということを示している。

## 5 おわりに

本稿では、我々が開発した現代的なハードウェア環境で高効用アイテムセットマイニングを効率よく実行するための手法である DPHIM を紹介した。DPHIM は高効用アイテムセットマイニングのタスクを大量のサブタスクに動的に分解し、これらを並列に実行する。実験により DPHIM は様々なデータセットや実験設定の下で著しい性能向上を示し、最大で最適化された逐次実行に対して 33.9 倍もの高速化に成功した。

今後の展望としては、第一に現在 DPHIM はマルチコアプロセッサやメモリ帯域を最大限に活用するために単一サーバー内の並列化に限定されているが、これをサーバー間の並列化に拡張することが挙げられる。特にサーバー間での並列化を行うためには複数のサーバー間でタスクを効率的に均等に分配する方法について研究を進めていく必要がある。第二に、並列化によって得られる性能は実行環境やデータセットの性質によって異なるので、性能を最大化するために自動でチューニングする技術を模索する必要がある。最後に DPHIM は高効用アイテムセットマイニング向けの One-phase アルゴリズムの動的並列化手法として設計されているが、これを他のマイニング問題やアルゴリズムに拡張していくことが可能である。今後、さらに広範な実験を行い、様々なデータセットやアルゴリズムを用いた DPHIM の有用性を検証していきたい。

## 謝　辞

本研究の一部は、日本学術振興会科学研究費補助金 20H04191 の助成を受けたものである。

## 文　献

[1] Agrawal, R., Imielinski, T., and Swami, A. N. Mining association rules between sets of items in large databases. In *Proc. SIGMOD 1993* (1993), ACM Press, pp. 207–216.

[2] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules in large databases. In *Proc. VLDB'94* (1994), Morgan Kaufmann, pp. 487–499.

[3] Blumofe, R. D. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994* (1994), IEEE Computer Society, pp. 356–368.

[4] Borkar, S. Thousand core chipsa technology perspective. In *Proc. DAC 2007* (2007), IEEE, pp. 746–749.

[5] Erwin, A., Gopalan, R. P., and Achuthan, N. R. Efficient mining of high utility itemsets from large datasets. In *Proc. PAKDD 2008* (2008), vol. 5012 of *Lecture Notes in Computer Science*, Springer, pp. 554–561.

[6] Fournier-Viger, P. Generating synthetic utility values for a transaction database without utility values. `http://www.philippe-fournier-viger.com/spmf/Generating_synthetic_utility_values.php`.

[7] Fournier-Viger, P. SPMF: An open-source data mining library. `https://www.philippe-fournier-viger.com/spmf/`.

[8] Fournier-Viger, P., Chun-Wei Lin, J., Truong-Chi, T., and Nkambou, R. *A Survey of High Utility Itemset Mining*. Springer International Publishing, Cham, 2019, pp. 1–45.

[9] Fournier-Viger, P., Lin, J. C., Duong, Q., and Dam, T. PHM: mining periodic high-utility itemsets. In *Proc. Industrial Conference ICDM 2016* (2016), vol. 9728 of *Lecture Notes in Computer Science*, Springer, pp. 64–79.

[10] Fournier-Viger, P., Wu, C., Zida, S., and Tseng, V. S. FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Proc. ISMIS 2014* (2014), vol. 8502 of *Lecture Notes in Computer Science*, Springer, pp. 83–92.

[11] Krishnamoorthy, S. Pruning strategies for mining high utility itemsets. *Expert Syst. Appl. 42*, 5 (2015), 2371–2381.

[12] Lan, G., Hong, T., and Tseng, V. S. An efficient projection-based indexing approach for mining high utility itemsets. *Knowl. Inf. Syst. 38*, 1 (2014), 85–107.

[13] Lan, G.-C., Hung, T.-P., and Tseng, V. An efficient gradual pruning technique for utility mining. *Int'l J. of Innovative Computing, Information and Control 8* (07 2012), 5165–5178.

[14] Lewis, B., and Berg, D. J. *Multithreaded Programming With PThreads*. Prentice Hall, 1997.

[15] Li, H., Huang, H., Chen, Y., Liu, Y., and Lee, S. Fast and memory efficient mining of high utility itemsets in data streams. In *Proc. ICDM 2008* (2008), IEEE Computer Society, pp. 881–886.

[16] Li, Y., Yeh, J., and Chang, C. Isolated items discarding strategy for discovering high utility itemsets. *Data Knowl. Eng. 64*, 1 (2008), 198–217.

[17] Liu, M., and Qu, J. Mining high utility itemsets without candidate generation. In *Proc. CIKM'12* (2012), ACM, pp. 55–64.

[18] Liu, Y., Liao, W., and Choudhary, A. N. A two-phase algorithm for fast discovery of high utility itemsets. In *Proc. PAKDD 2005* (2005), vol. 3518 of *Lecture Notes in Computer Science*, Springer, pp. 689–695.

[19] Luna, J. M., Fournier-Viger, P., and Ventura, S. Frequent itemset mining: A 25 years review. *WIREs Data Mining and Knowledge Discovery 9*, 6 (2019), e1329.

[20] Majo, Z., and Gross, T. R. Memory system performance in a NUMA multicore multiprocessor. In *Proc. SYSTOR 2011* (2011), ACM, p. 12.

[21] Shie, B., Tseng, V. S., and Yu, P. S. Online mining of temporal maximal utility itemsets from data streams. In *Proc. SAC 2010* (2010), ACM, pp. 1622–1626.

[22] Silberschatz, A., Galvin, B., and Gagne, G. *Operating system concepts*. Wiley, 2013.

[23] Wu, J. M., Lin, J. C., and Tamrakar, A. High-utility itemset mining with effective pruning strategies. *ACM Trans. Knowl. Discov. Data 13*, 6 (2019), 58:1–58:22.

[24] Yao, H., Hamilton, H. J., and Butz, C. J. A foundational approach to mining itemset utilities from databases. In *Proc. SDM 2004* (2004), SIAM, pp. 482–486.

[25] Yao, H., Hamilton, H. J., and Geng, L. A unified framework for utility based measures for mining itemsets. In *Proc. UBDM'06* (2006), pp. 28–37.

[26] Yin, J., Zheng, Z., and Cao, L. Uspan: an efficient algorithm for mining high utility sequential patterns. In *Proc. KDD '12* (2012), ACM, pp. 660–668.

[27] Zida, S., Fournier-Viger, P., Lin, J. C., Wu, C., and Tseng, V. S. EFIM: A highly efficient algorithm for high-utility itemset mining. In *Proc. MICAI 2015* (2015), vol. 9413 of *Lecture Notes in Computer Science*, Springer, pp. 530–546.