DEIM Forum 2023 2a-4-4

再帰的演算を含む分析処理の高効率な並列実行方式の提案と有効性評価

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

† 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,haya,kitsure,kgoda}@tkl.iis.u-tokyo.ac.jp, ††udayrage@u-aizu.ac.jp

あらまし　蓄積されたデータから有用な情報を取り出す分析的処理では、分析対象となるデータの量が年々増加しており、この増加に対応するために処理の高速化が求められている。分析アルゴリズムの改良や索引データ構造の改善も一定の成果を上げているが、近年発展の著しいマルチコア計算機を最大限活用した並列処理による高速化に大きな期待が寄せられている。本研究では、分析的処理の中でも特に再帰的演算を含むもの対象とし、タスクを実行時に分割して並列実行する動的並列化とタスクやデータを投機的に分割・配置する投機的分散制御を組み合わせた高効率な並列実行方式を提案した。また、この実行方式を近年注目されているデータマイニング問題である高効用アイテムセットマイニングに適応した際の性能改善を実験的に評価した。

キーワード　並列・分散処理, 先進ハードウェア活用, データマイニング

1　はじめに

ビッグデータの活用が重要視されている現代において、蓄積されているデータから有用な情報を素早く取り出す技術の確立は必要不可欠である。加えて生成・蓄積されるデータの総数は年々指数関数的に増加しているため、その増加に対応できるようなデータ処理の高速化技術が重要である。蓄積されるデータや要求される情報が多様になるのに伴ってより多様なデータ処理が求められている。例えばデータベース処理においては、従来の関係データベースやキーバリューストアでは配列上の線形探索やハッシュ表、木構造への操作が主たるデータ処理だったが、近年注目されるグラフデータベースではより複雑なグラフ構造を用いることにより多様なデータ処理を可能としている。用いられるデータ処理は多様だが、複雑なデータ処理は基本的に再帰的演算を含むため、再帰的演算に対して一般的に用いることができる高速化技術を確立することが重要である。

従来はムーアの法則と言われるように CPU のシングルスレッド性能も指数関数的に向上していたため、既存のデータ処理手法をそのまま使い続けていれば自然と性能が向上していた。しかし、CPU のシングルスレッド性能の向上速度が低下している現代では、CPU の性能改善に頼らない処理内容そのものの効率化が必要不可欠である。特にデータマイニング分野の研究などにおいては、各問題に対して独立にアルゴリズムを改善することで大幅な高速化を達成しているが、一方で処理内容によらないより一般的な高速化手法の確立も重要である。近年の特にデータ処理に用いられる計算環境は一台の計算機に複数のプロセッサを搭載したマルチプロセッサ計算機である。複雑なデータ処理が要求される場合、単一のプロセッサでは実行に時間がかかりすぎるため、複数のプロセッサを協調的に用いることで実行時間を短縮する。また、マルチプロセッサであるだけでなく、それに伴って主記憶装置や補助記憶装置へのアクセス帯域も増大している。単一のプロセッサのみを用いるのではこれらの帯域を十分活用することができないが、複数のプロセッサから同時にアクセスすることによりハードウェアの備える潜在的な帯域を全て活用したデータ処理が可能となる。

データ処理等で用いる高い処理能力を有したマルチプロセッサ計算機は NUMA(Non-Uniform Memory Acceess) アーキテクチャであることが多い。マルチプロセッサ計算機であってもプロセッサ数があまり多くない場合は単一の CPU ソケット上に全てのプロセッサが乗り、単一バス上に全てのプロセッサとメモリが配置されるため、任意のプロセッサから任意のメモリ領域へのアクセスコストが均一である。一方、プロセッサ数が十分多い計算機では単一の CPU ソケットに全てのプロセッサが収まらない。この場合も単一バス上に全てを配置することは可能だが、配線長が長くなるなどの要因により、メモリアクセスコストが増大する。そこで、アクセスコストの均一性を諦め、CPU ソケットごとにバスを複数用意し、各バスにメモリを分配したものが NUMA アーキテクチャである。NUMA アーキテクチャでは、各プロセッサから各メモリ領域へのアクセスコストが異なるため、できる限り実行コストを下げるためには、どのプロセッサでどのタスクを処理し、どのメモリ領域にどのデータを配置するかということが重要になる。

本研究では高効率並列実行方式として、動的並列化と投機的分散制御を提案した [6,13,14]。動的並列化では実行タスクを実行時に細粒度タスクに分割し分割されたサブタスクを動的に各プロセッサに割り当てることで、計算機の備えるプロセッサを最大限活用した高効率な実行を可能とし、タスクのプロセッサへの割り当て戦略として参照の局所性を活かした効率化が可能な局所拘束戦略を用いた。投機的分散制御では、データベースをあらかじめ各メモリ領域に分散配置し、一つのタスクをアクセスするメモリ領域に応じて分割する。分割されたサブタスク

Algorithm 1 再帰的演算を含む解析処理の一般化

Require: xs: a set of candidates, D: database
Ensure: R: a set of results
1: **function** ANALYZE(xs, D)
2: $R \leftarrow \emptyset$
3: **for all** $x \in xs$ **do**
4: **if** ISANSWER(x, D) **then**
5: $R \leftarrow R \cup \{x\}$
6: $ys, D' \leftarrow$ GETNEXT(x, D)
7: $R \leftarrow R \cup$ ANALYZE(ys, D')
8: **return** R

をアクセスコストが低いプロセッサで実行されるように制御することで、参照の局所性を前提とすることができない処理においても高効率な実行を可能とした。さらに、この手法を近年注目されているデータマイニング問題である高効用アイテムセットマイニング向けアルゴリズムである EFIM に対して適用し、逐次実行に対して大幅な性能向上を示すことを実験的に確かめた。本論文では、提案手法をさらに一般化し、再帰的演算を含む解析処理に対して広く適用可能であることを示す。

本論文は本章を含めて 5 章から構成される。第 2 章では再帰的演算を含む解析処理や並列実行に関する既存研究について述べる。第 3 章では本論文で提案する動的並列化と投機的分散制御を紹介する。第 4 章では提案手法を高効用アイテムセットマイニングに対して適用した実験結果を示す。第 5 章では本論文の総括を行い、今後の研究課題についても述べる。

2 背　　景

2.1 再帰的演算を含む解析処理

解析対象のデータ構造が木構造やグラフ構造などの再帰的な構造を持っている場合だけでなく、解候補が集合や配列等の再帰的構造を持っている場合など、多種多様な解析処理において処理アルゴリズムは再帰的演算を含みうる。

再帰的演算を含む解析処理を一般化したものをアルゴリズム 1 に示す。再帰的演算の 1 ステップは ANALYZE 関数として記述されており、この関数は解候補集合 xs と解析対象のデータベース D を引数に取る (1 行目)。最初に最終的に出力する解集合 R を空集合で初期化しておく (2 行目)。次に xs 中の各候補 x に対して以降の処理を繰り返し実行する (3 行目)。x は解の候補に過ぎないので ISANSWER 関数を用いて実際に解であるかを確かめ、x が実際に解であれば解集合 R に追加する (4-6 行目)。その後 MAKENEXT 関数により、現在の解候補 x とデータベース D から、次の解候補集合 ys とデータベース D' を構築し (7 行目)、構築した ys と D' を用いて再帰的に ANALYZE を呼び出し、得られた解集合を R に追加する (8 行目)。なお、ISANSWER 関数と MAKENEXT 関数は各解析処理に固有の関数である。解析対象のデータベース D は不変である必要はなく、他にもデータベース上の探索コストを下げるために D の一部に制限したり、D にキャッシュを追加したりするなど MAKENEXT 関数による新たなデータベース D' の構築方法が考えられる。

この一般化処理を用いてグラフ上の深さ優先探索を記述すると、xs は現在探索中のノードに接続しているノードのうち未訪問のものの集合、D はグラフとなる。また、is_acceptable 関数はノード x が探索対象のノードかを判定する関数であり、make_next 関数はノード x とグラフ D を受け取って、x に隣接した未訪問ノードの集合 ys と x に訪問済みフラグを付けたグラフ D' を返す。

2.2 再帰的演算を含む分析処理の効率的実行

既存の再帰的演算を含む分析処理の実行効率化の代表的手法として以下の 3 種類がある。第一に、より効率的なヒューリスティックの導入等によるアルゴリズム的な改善である。実行効率化の対象となる問題設定を固定した場合、その問題設定に特有のヒューリスティックを導入することにより、実行すべき処理の総量を大幅に削減することができうる。データマイニング問題の一つである高効用アイテムセットマイニング [1,3,7,8,11] では、あるアイテムセットが解か否かを判定するための効用値を逐次的に計算するのが困難であるため、最初は全探索を用いていた。その後 Transaction Weighted Utility (TWU) なる効用値の上限を与えつつ容易に計算可能なヒューリスティクスの導入により、飛躍的に探索空間が削減され、実行性能が向上した [8]。その後も TWU の計算を高速化するためのデータ構造や TWU 以外のヒューリスティクスを導入することにより、実行性能が改善されている [4,7,12]。対象とする問題が明らかな場合は、その問題に対する知識を活かして探索空間を削減し、飛躍的に実行を効率化することが可能となる。

第二に、コンパイラ最適化等で用いられるシングルスレッド実行向けの汎用実行効率化手法である。データベースの線形探索等の繰り返し処理に対する手法は、終了条件の確認処理やジャンプ等の分岐処理を削減するループアンローリングや繰り返される処理を小さくすることでキャッシュヒット率を向上させるループ分割などがある。また再帰的演算を含む処理に特有の手法として末尾呼び出し最適化 [5] がある。関数呼び出しではレジスタをスタック領域に退避する必要があるため、スタック操作のコストが余計にかかる。処理の末尾で行われる関数呼び出しでは、その関数呼び出しを単なるジャンプ命令に置き換えることでスタック操作を回避し、実行を効率化できる場合がある。再帰的な呼び出しでは、このジャンプが実行中関数の先頭へのジャンプになり、再帰的な関数呼び出しをより軽量なループへ変換することができる。

最後に、並列実行による効率化である。近年の計算機はプロセッサを複数備えるマルチプロセッサ環境であり、処理全体を適切に分割し各プロセッサで同時に実行することで実行時間を短縮することができる。特に分析処理ではデータベースに対する操作が主に検索等のデータベースを変更しない処理であるので、ロックやアトミック操作により並列実行されている処理間で同期をとる必要がないため、理想的にはほぼアムダール則に従う高速化が可能となる。また、近年のプロセッサの命令セットは SIMD 命令 [10] を備えておりベクトル演算が可能である

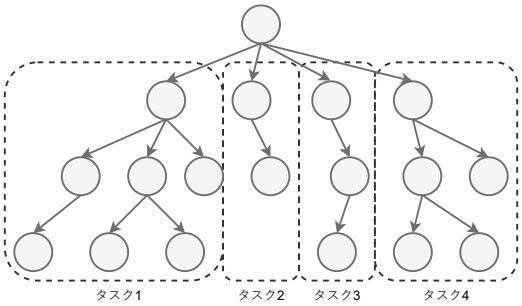


図 1: 静的パーティショニングによるタスク分割の例

ため、単一のプロセッサのみであっても並列実行が可能となる。特にデータ分析処理では、各データに対して同じ処理が繰り返される実行パターンが頻出であるため、ベクトル演算により実行を高速化できる可能性がある。

2.3 静的パーティショニング

マルチプロセッサを活用した並列実行による実行効率化の典型的手法として静的パーティショニングがある。静的パーティショニングでは実行すべきタスクを事前に決まった個数のサブタスクに分割し、これらを並列実行する。典型的には計算機の備えるプロセッサ数に分割し、各サブタスクを各プロセッサに割り当てて実行する。静的パーティショニングは事前にサブタスクの個数や並列実行の様子が決定されるので実装が容易である一方、タスクを均等に分割できない場合には計算機の並列処理性能を十分に活用することができないという問題がある。

図 1 には再帰的処理における静的パーティショニングの様子を模式的に示した。図中の丸は再帰の 1 ステップ、つまりアルゴリズム 1 中の Search の呼び出し 1 回に相当する。この例では最初の 1 ステップで行われた 4 回の再帰的に呼び出しを分割して並列実行し、以降の再帰呼び出しは全て呼び出し元と同一プロセッサで実行している。つまり、破線で囲まれた各領域が並列実行の一つのタスクである。この例では、各サブタスクの大きさが大きく異なっているため、最小のタスク 2 を実行していたプロセッサは最大のタスク 1 の実行が完了するまで何もせず待機する必要があり、全てのプロセッサを無駄なく活用できていない。実行前に全ての処理内容がわかっていれば実行コストが均等になるような分割も可能であるが、実行時情報に基づいて枝刈りを行う場合など各タスクの実行コストが事前にわからない場合も多く、均等な分割は事実上不可能である。

2.4 NUMA

近年のマルチプロセッサ環境、その中でもプロセッサ数が多いものは Non-Uniformed Memory Access(NUMA) というアーキテクチャが用いられる。通常のアーキテクチャでは複数のプロセッサにより共有されるメモリは、そのアクセスコストがプロセッサやメモリ上の位置によらず一定であるが、NUMA アーキテクチャではどのプロセッサからどのメモリ領域にアクセスするかによってアクセスコストが変わる。現在よく用いられているメモリ管理システムでは同一のメモリ領域に同時にアクセスできるプロセッサは一つだけであるため、プロセッサ数

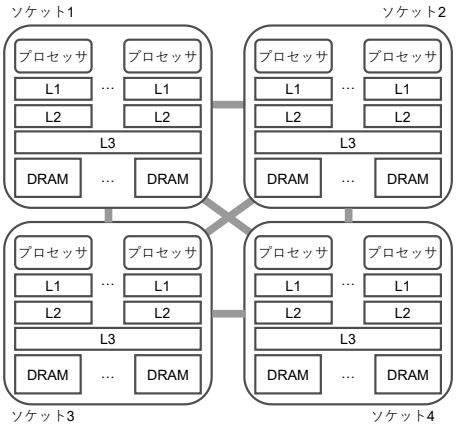


図 2: 4 ソケットの NUMA アーキテクチャ

を増やしていくとプロセッサ間でメモリアクセスが競合して性能がスケールしないという問題がある。

これを解決するために、NUMA アーキテクチャではメモリ領域を各ソケットごとに分割し、各領域にはその所有ソケットのみアクセスできるようにすることで、複数のプロセッサから同一のメモリ領域への多数の同時アクセスが発生しないようにした。図 2 には、4 つのソケットを備える計算機のアーキテクチャの例を示した。各プロセッサは自身の所属するソケット上のメモリ領域へはこれまで通りのメモリアクセスを行うが、他ソケット上のメモリ領域へはそのソケットを経由してアクセスする。これによりメモリアクセスは前者の高速なローカルアクセスと後者の低速なリモートアクセスに分けられる [9]。

メモリアクセスが空間的な局所性を持つアプリケーションではメモリ割り当てを適切に行うことによりほぼ全てのメモリアクセスをローカルアクセスにすることが可能であるため、NUMA アーキテクチャにより性能をスレッド数に対してスケールさせることが可能となる。また、他に同一のメモリ領域に対して複数のプロセッサが同時にアクセスできるようにすることでもスケール性を向上させることが可能だと考えられるが、データをやりとりするアクセスバスが複雑になり、回路面積が増大するため現実的には困難である。

3 再帰的演算を含む解析処理の効率的な並列実行

3.1 動的並列化

事前にタスクの分割方法を決定する静的パーティショニングではタスクの不均衡によりハードウェアの有する並列度を十分に活かせない。これを解決するために動的にタスクを多数のサブタスクに細粒度分割する。分解により得られた各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。この時、既に割り当てられているタスクが少ないプロセッサに優先的に新たなタスクを割り当てることで、プロセッサ間のタスク数の不均衡が解消され、実行を効率化することができる。

通常のタスクでは適切なタスク分割境界の決定が難しいが、アルゴリズム 1 に示すような再帰的演算を含む解析処理は再帰

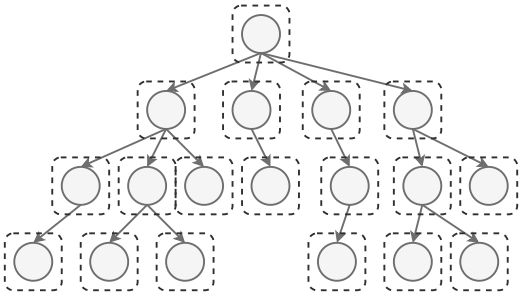


図 3: 動的並列化によるタスク分割の例

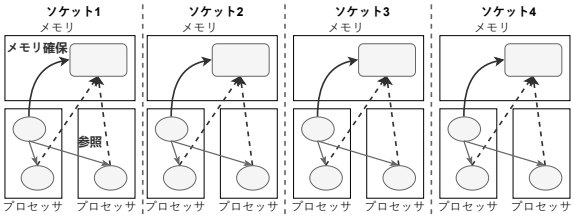


図 4: 局所拘束の例

Algorithm 2 動的並列化

Require: x: a candidate, D: database
Ensure: R: a set of results
1: **function** AnalyzeOne(x, D)
2: $R \leftarrow \emptyset$
3: **if** IsAnswer(x) **then**
4: $R \leftarrow \{x\}$
5: $ys, D' \leftarrow$ GetNext(x, D)
6: $R \leftarrow R \cup \{$Analyze(ys, D')$\}$
7: **return** R

Require: xs: a set of candidates, D: database
Ensure: a set of results
8: **function** Analyze(xs, D)
9: $T \leftarrow \emptyset$
10: **for all** $x \in xs$ **do**
11: $t \leftarrow$ MakeTask(AnalyzeOne, x, D)
12: $T \leftarrow T \cup \{t\}$
13: **return** ExecAndWaitAll(T)

的演算を分割境界とすることで容易に細粒度分割が可能である。アルゴリズム 1 に対して動的並列化を適応したものをアルゴリズム 2 に示す。なお、AnalyzeOne はアルゴリズム 1 のループ内部を抽出したものである。まず実行結果を格納する R を空集合で初期化し (2 行目)、もし現在探索中の解候補 x が実際に解なら R に追加する (3-5 行目)。その後 GetNext により次の探索候補集合 ys とデータベース D' を作成し (6 行目)、これらを引数として Analyze を再帰的に呼び出した結果を R に追加する (7 行目)。最後に呼び出し元に R を返却する (8 行目)。

Analyze 関数では解候補集合 xs 中の各解候補 x に対する処理を全てサブタスクに分割し並列実行する。まずサブタスクを格納する集合 T を空集合で初期化し (11 行目)、xs 中の各 x に対して AnalyzeOne(x, D) を実行するタスク t を生成し、T に追加する (12-15 行目)。MakeTask は第一引数に指定した関数に第二引数以降の値を引数として渡して実行するタスクを生成する関数である。最後に T 中の全タスクを非同期的に実行し、その実行完了を待機する (16 行目)。ExecAndWaitAll の実行結果は引数中のタスクの実行結果の和集合である。

図 1 と同様の処理に対して動的最適化を適用したものを図 3 に示す。再帰的処理の各ステップに対して 1 つのタスクが生成されるので、この例では最終的に 18 個のサブタスクに分割される。この最終的な分割数は事前にはわからないため、タスクの割り当ては実行時に制御する必要がある。また、得られた各サブタスクは静的パーティショニングで生成されるタスクよりも細粒度なため、タスクの実行制御を適切に行うことにより、プロセッサ間の処理の不均等を解消した並列実行が可能である。これによりマルチプロセッサ環境の処理能力を最大限活用して分析処理を行うことができる。

3.2 局所拘束

多くのデータ処理は参照に局所性を持つ、つまりあるデータが参照された場合にその近傍のデータが近いうちに参照される

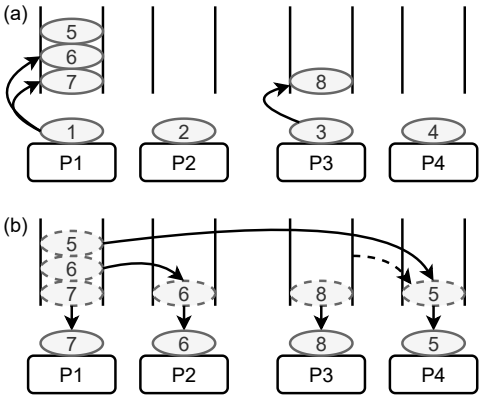


図 5: 局所拘束を行ったタスク制御の例

傾向にある。さらに再帰的演算を含むデータ処理では、あるタスクで参照されたデータはその子孫タスクによっても参照される。現代の計算機は参照の局所性を活かすために、直近でアクセスしたデータをプロセッサの近くにキャッシュし、次回以降のアクセスを高速化する。そのため、図 4 に示すように親タスクと子タスクが同じプロセッサで実行されるようにするとこのキャッシュ機構によりメモリアクセスが高速化される。また NUMA アーキテクチャを有する計算機では別の NUMA ノード上のメモリにアクセスするコストは非常に高いため、データを参照するタスクはデータが配置されているノードと同じ NUMA ノードで実行された方が良い。データを配置するために確保されたメモリ領域は、初期化や書き込みのためにメモリ確保を行ったタスク自身によってまず最初にアクセスされる。このアクセスをローカルアクセスとするためにデータを作成するタスクが実行されている NUMA ノードと同じノード上のメモリ領域から割り当てる。さらにデータを作成するタスクの子タスクを次のように制御することで子タスクからの参照も可能な限りローカルアクセスにすることができ、実行コストを下げることが可能となる。

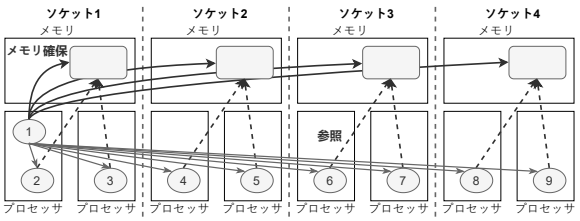


図 6: 投機的分散制御の例

局所拘束におけるタスク制御系を図 5 に示す。各プロセッサは 1 つずつタスクを実行している (図 5(a)-タスク 1,2,3,4)。また、現在実行されていないが今後実行するべきタスクを管理するためのタスクキューを各プロセッサに 1 つずつ割り当て、その中に実行すべきタスクを入れておく (図 5(a)-タスク 5)。各プロセッサは現在実行中のタスクから新たなタスクが生成されたら、自身のタスクキューにそのタスクを追加する (図 5(a)-タスク 6,7,8)。実行中のタスクが終了すると、各プロセッサは自身のタスクキューから未実行のタスクを取り出して実行する (図 5(b)-タスク 7,8)。自身のタスクキューが空である場合は、何のタスクも実行しないタスクが存在しないようにするために、他のプロセッサのタスクキューからタスクを取り出して代わりに実行する。この時に同じ NUMA ノードに属するプロセッサから優先的に取り出すようにすることで、タスクを可能な限り生成された NUMA ノード上に拘束することが可能である。図 5 ではプロセッサ P1 と P2、P3 と P4 がそれぞれ同じ NUMA ノード上に存在するので、P2 は自身のタスクキューが空であるため優先的に P1 のタスクキューからタスクを取り出して実行する (図 5(b)-タスク 6)。P4 も同様に自身のタスクキューが空であるため、まず最初に P3 のタスクキューからタスクを取り出すことを試みる。しかし、P3 のタスクキューはこの時点で空であるため、別の NUMA ノード上にある P1 のタスクキューからタスクを取り出して実行する (図 5(b)-タスク 5)。これにより全てのプロセッサを活用しつつ、参照の局所性を考慮した効率的な実行が可能となる。

3.3 投機的分散制御

通常の再帰的演算を含む解析処理では、再帰深度が増すごとに解くべき問題が小さくなるので、検索対象のデータベースは小さくなることが多い。この縮小によりデータベースがメモリ上のより小さな領域に収まる場合、再帰深度が増すごとに参照しうる範囲が狭くなり、データベースに対する任意の参照操作の局所性が増すため、局所拘束により効率的な実行が可能となる。一方、データベースは小さくなるが、メモリ上のデータ配置はまばらで参照しうる範囲はあまり変わらない場合もある。この時、スキャンなどの局所性を持たない参照が主たる参照パターンであった場合、参照の局所性はなくなり、参照の局所性を前提とする局所拘束では実行の効率化を期待できない。さらに、参照の局所性がないことにより複数のプロセッサから同時に同じメモリ領域へのアクセスが発生する可能性があるため、これらのアクセスが競合することによりメモリアクセスコストが増大しうる。投機的分散制御ではデータの分散配置及びタスクの動的な分割及び移動により、局所拘束では対応できない場合に対しても実行の効率化を行う。

Algorithm 3 投機的分散制御

```
Require: i: node id, f: a function, args...: arguments
function ExecOn(i, f, args...)
    Switch(i)
    return f(args...)

Require: f: a function, args...: arguments
function ScatterMode(f, args...)
    tmp ← malloc
    malloc ← scatter_malloc
    f(args...)
    malloc ← tmp

Require: x: a candidate, D: database
function AnalyzeOne(x, D)
    R ← ∅
    if IsAnswer(x) then
        R ← {x}
    T ← ∅
    for i ← 1...NUMA_NUM do
        D_i ← the portion of D on node i
        l ← estimated size of the result
        if l > scatter_threshold then
            f ← Curry(ScatterMode, MakeNext)
        else
            f ← MakeNext
        if local_cost > remote_cost + task_creation_cost then
            t ← MakeTask(ExecOn, i, f, x, D_i)
        else
            t ← MakeValue(f(x, D_i))
        T ← T ∪ {t}
    ys, D' ← ExecAndWaitAll(T)
    R ← R ∪ {Analyze(ys, D')}
    return R
```

NUMA アーキテクチャではプロセッサからメモリへのアクセスコストが均一でないため、特定のメモリ領域に全データが集中していると必ずいずれかのプロセッサからのアクセスコストは他と比較して大きくなる。投機的分散制御では図 6 のようにあらかじめデータをメモリ全体に分散配置しておき、各データを扱う処理はそのデータが配置されているメモリ領域へのアクセスコストが小さいプロセッサで行うようにすることで平均的なメモリアクセスコストを下げる。同時にメモリアクセスが特定のメモリ領域に集中しなくなり、メモリアクセスの競合によるレイテンシの増加が改善される。しかし、常にデータを分散配置したり、タスクを分割することはコストが大きい。データの分散配置では、遠くにある NUMA ノードからメモリを確保しそこに書き込みを行う必要があるため、通常のデータ構築よりコストが高い。タスクの動的分割は新たなタスクを生成することになるため、その構築コストや管理すべきタスク数が増加することによる管理コストの増加が無視できない。また、各タスクはデータベースにアクセスするだけでなくタスクにロー

カルなデータも扱うため、タスクを別の NUMA ノード上にアクセスすると、データベースへのアクセスコストは下がるが元々の NUMA ノード上に配置されているローカルなデータへのアクセスコストは増加しうる。そのため、必ずしも常にデータの分散配置及びタスクの分割・移動をすることで性能改善するわけではない。

投機的にデータ配置及びタスク実行を制御する投機的分散制御をアルゴリズム 2 に適用したものをアルゴリズム 3 に示す。投機的分散制御用のユーティリティ関数として ExecOn と ScatterMode を導入する。MoveTask(f, i) は関数 f を i 番目の NUMA ノード上のプロセッサで実行する (1-4 行目)。なお、Switch(i) 関数は現在の実行コンテキストを中断して NUMA ノード i 上で再開する命令である。ScatterMode($f, args\ldots$) は $f(args...)$ を実行するが、このときにメモリ確保関数の malloc を通常のメモリ確保関数から各 NUMA ノードから均等にメモリ領域を確保する scatter_malloc に置き換える (5-10 行目)。

AnalyzeOne 関数内では、データベース D を配置されている NUMA ノードによってに D_i に分割する (17-18 行目)。その後各 D_i に対して MakeNext(x, D_i) を実行するが、その結果をどのように配置するかが異なる。19-24 行目では推定される実行結果の大きさによって結果の配置方法を決定している。実行結果が十分小さければ局所拘束に基づきタスクを実行した NUMA ノード上のメモリに配置されるが、実行結果が大きい場合は各 NUMA ノードに分散配置する。この決定方法は一つのヒューリスティクスであり、その他の決定方法についても後述する。なお、21 行目で用いている Curry はカリー化を行う関数である。

25-29 行目では先ほど決定した実行タスク f をどこで実行するかを決定している。その場で実行した時のコスト (local_cost) が、NUMA ノード i 上で実行した時のコスト (remote_cost) と移動して実行するためにタスクを作成するコスト (task_creation_cost) の和よりも大きい場合は (25 行目)、f を NUMA ノード i 上で実行するタスクを生成する (26 行目)。反対にその場で実行するコストの方が小さい場合は f をその場で実行する (28 行目)。なお、MakeValue は引数に渡された値をそのまま返す軽量なタスクを生成する関数とする。その後作成されたタスク群 T を並列に実行及び実行完了を待機し、次の解候補集合 ys 及びデータベース D' を得る (32 行目)。最後に Analyze を再帰的に呼び出して再帰の 1 ステップは終了である (33 行目)。

データを分散配置するかどうかを決定する指標としてアルゴリズム 3 中では実行結果のサイズを用いたが、他にもいくつかの指標が考えられる。データの分散配置の目的は NUMA アーキテクチャにおけるリモートアクセスを可能な限り減らすとともに、メモリ領域全体にメモリアクセスを分散させることでハードウェアの持つ帯域幅を最大限活用することである。そのため分散配置におけるオーバーヘッドがないのであれば常に均等に分散配置するのがよい。アルゴリズム 3 で用いたタスクの実行結果の大きさの推定値という指標は、データの分散配置が厳密に均等である必要はないため、一定以上大きいデータを配置する時のみ分散配置するという近似である。また他にも子孫タスクの数を推定し、その数が一定以上多い場合に分散配置するという指標も考えられる。これは子孫タスクの数が将来的なメモリアクセスの回数に比例すると近似し、分散配置を行わなければメモリアクセスが偏りそうな場合にのみ分散配置をするという手法である。

表 1: データセットおよび最小効用

Dataset	# of items	# of transactions	*minutil*
Kosarak [2]	41270	990002	1500000
Chainstore [2]	46086	1112949	2000000
BMS [2]	3340	77512	2060000
Accidents [2]	468	340183	20000000

4 実　　験

分析的なデータ処理タスクとして高効用アイテムセットマイニングを取り上げる。また、高効用アイテムセットマイニング向けアルゴリズムの EFIM を効率化対象のアルゴリズムとして採用した。EFIM に提案手法を適用し、提案手法による性能改善を検証する実験を行った。

4.1 実 験 設 定

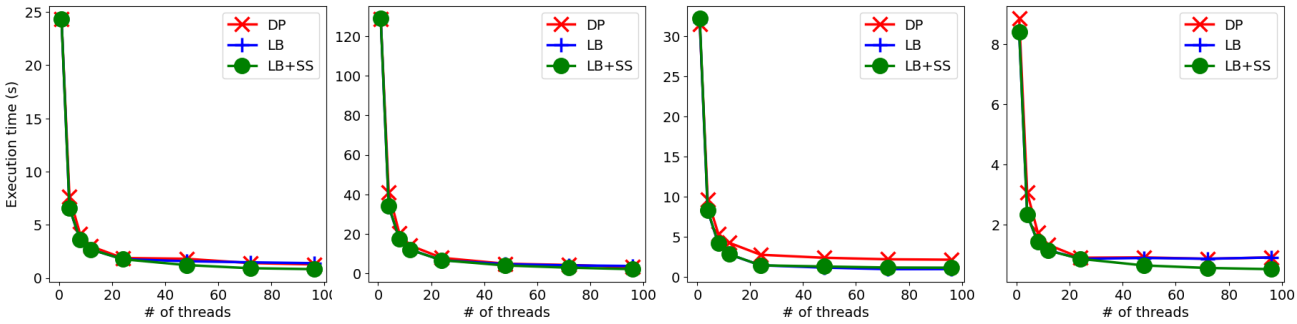
実験に用いた計算機は Xeon Gold 6252 processors (24 cores, 2.10 GHz) を備えており、各ソケットには 192 GB の DRAM が付属する。また、OS は Ubuntu20.04.3 である。表 1 には実験で用いた Kosarak, Chainstore, BMS, Accidents の 4 つのデータセットの基本的な性質について示した。これらは外部効用の無い実際のデータセットを元に、外部効用を追加したものである [7]。

提案手法の有効性を検証するために次の実装を用意した。

- **DP**: EFIM に動的並列化のみを適用したもの
- **LB**: **DP** にさらに局所拘束を適用したもの
- **LB+SS**: **LB** にさらに投機的分散制御を適用したもの

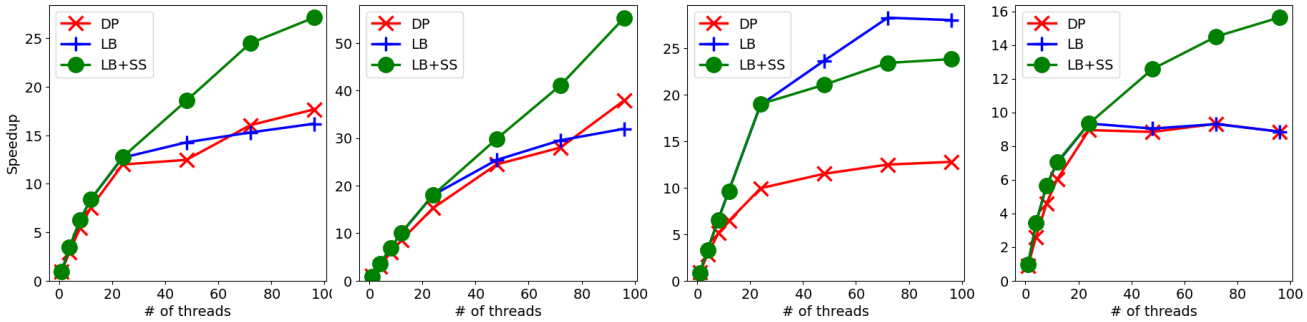
4.2 実行時間の比較

提案手法による性能向上を検証するために、スレッド数を増やしていったときの実行時間の変動を測定した。図 7(a) にデータセットに Kosarak を用いた時の実行時間を示す。スレッド数が 1 の時は実行時間が **DP, LB, LB+SS** の全てで 24.3 秒と同一であったが、スレッド数が 24、つまり 1 つの NUMA ノード分に相当するスレッド数に増加すると **DP** では 1.9 秒だったが、**LB** と **LB+SS** では 1.8 秒と実行時間が改善された。**DP** は NUMA アーキテクチャを考慮せずに動的並列化のみを行うため、実行すべきタスクが各 NUMA ノードに分散されメモリアクセスがローカルアクセスだけでなくリモートアクセスも発生するため実行時間が長くなる。一方、**LB, LB+SS** では局所拘束により実行すべきタスクの全てが単一の NUMA ノード内で実行されたために全メモリアクセスがローカルアクセスになり、実行時間が短縮されたと考えられる。さらにスレッド数



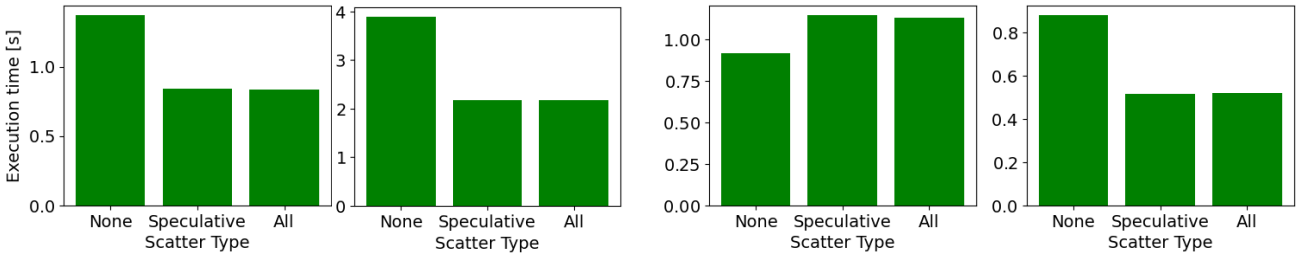
(a) Kosarak (b) Chainstore (c) BMS (d) Accidents

図 7: 実行時間の比較.



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

図 8: スレッドスケーラビリティ.



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

図 9: 分散制御手法ごとの実行時間

が 96、つまり全てのプロセッサを利用して実行した場合、**DP** では 1.3 秒、**LB** では 1.4 秒、**LB+SS** では 0.83 秒であった。これは投機的分散制御により実行性能が改善されたことを示している。

同様の実験を他のデータセット (Chainstore, BMS, Accidents) に対して行った結果を図 7(b)-(d) に示す。スレッド数が 1 の時は実行時間が各データセットについてそれぞれ約 129 秒、約 32 秒、約 8.5 秒でほぼ同じである。スレッド数が 24 では Kosarak の時と同様に **DP** は **LB** 及び **LB+SS** よりも実行時間がやや長いが、**LB** と **LB+SS** の実行時間はほぼ同じである。スレッド数が 96 になると全ての実装で実行時間に差が出るが、Chainstore 及び Accidents では **LB+SS** が最も実行時間が短いのに対して、BMS では **Lb** の実行時間が最も短い。これは BMS において子タスクの発生数のばらつきが大きいため、コスト推定がより正確であれば本来はより実行時間を短縮できるはずが投機的な分散制御に失敗し、実行コストが増加したためと考えられる。

4.3 スレッドスケーラビリティ

図 8(a) にはデータセットに Kosarak を用いたときに逐次実行時の実行時間を 1 としてスレッド数を増やしていった時にどの程度実行速度が増加するのかを示した。スレッド数が 24 以下の範囲内では全実装が似たような振る舞いを見せており、スレッド数が 24 で速度向上率は 12.0 倍 12.8 倍である。しかし、スレッド数が 24 を超えると **DP,LB** は速度向上率があまり改善されていないのに対して **LB+SS** ではさらなる改善を示しており、スレッド数が 98 での速度向上率は **DP,LB,LB+SS** のそれぞれについて 17.7 倍、16.2 倍、 27.1 倍である。これは通常 NUMA アーキテクチャを考慮しなかったり、考慮するにしても単なる局所拘束では計算環境の並列度を十分活かすことができないが、投機的分散制御により十分並列度を活かした実行が可能になるということを示している。

図 8(b)-(d) には同様の実験を他のデータセット (Chainstore, BMS, Accidents) に対して行ったときの結果を示した。**LB+SS** はいずれに対しても十分な速度向上率を示し、最大でそれぞれ 55.3 倍、23.8 倍、15.6 倍であった。Chainstore と Accidents については **DP,LB** がスレッド数 24 以上で速度向上率の改善が望めなくなっているのに対して **LB+SS** のみが投機的分散制御により改善を続けて計算環境のハードウェア性能を十分に活かした実行を可能にしている。データセットに BMS を用いた図 8(c) ではスレッド数が 24 よりも大きい範囲において **LB** が **LB+SS** よりも良い速度向上率を示している。これは投機的分散制御におけるコスト推定のミスにより実際は行わない方がよい分散制御が行われてしまったということが要因であると考えられる。データセットに BMS を用いると実行時間はあまり他のデータセットと変わらないが、生成されるタスク数が数百倍のオーダーで増加する。そのため、タスク生成やタスク制御自体等のコスト推定が重要になってくるが、本実験ではメモリアクセスコスト推定により分散制御の投機的実行を行ったため、全体としてのコスト推定に誤りが生じてしまったと考えられる。

4.4 分散制御手法ごとの実行時間

本稿では投機的分散制御を提案したが、分散制御を投機的に実行することによる効果を検証するために、分散制御を行わなかった場合、投機的に行った場合、常に行った場合の 3 通りの実行で実行時間を比較した。なお、スレッド数は 96 である。まずはデータセットに Kosarak を用いた場合の結果を図 9(a) に示す。分散制御を行わなかった場合 (**None**) は実行時間が 1.4 秒であり、投機的に分散制御を行った場合 (**Speculative**) は 0.84 秒である。分散制御を常に行った場合は (**All**) は 0.84 秒と投機的に行った場合と同じである。この場合は **Speculative** でもほぼ全てのメモリ確保において分散制御されるため、この結果は妥当である。この傾向は Chainstore や Accidents でも見られる。

データセットに BMS を用いると投機的分散制御を行った場合が最も実行時間が長く (1.1 秒)、一切分散制御を行わなかった場合が最も実行時間が短い (0.92 秒)。つまり投機的実行のためのコスト推定に誤りがあり、分散制御をすべきときに行わず、分散制御をすべきでないときに行ってしまったと考えられる。この結果から分散制御による実行速度の向上のためには正確なコスト推定が不可欠であることがわかる。

5 おわりに

本稿では、再帰的演算を含む分析的処理を効率的に並列実行するための手法として動的並列化及び投機的分散制御を提案した。動的並列化では、再帰的構造を利用してタスクを実行時に細粒度で生成していくことにより、プロセッサあたりの負荷の不均等を解消し、計算環境のハードウェアを最大限活用できるようにした。投機的分散制御では、局所性のないアクセスパターンを持つ処理に対してもデータを分散配置し、それにアクセスするタスクを動的に分割し各 NUMA ノードに移動させることで、リモートアクセスを減らしメモリアクセスの平均的なレイテンシを下げることを可能にした。また、高効用アイテムセットマイニング問題向けアルゴリズムである EFIM にこれらの手法を適用することで、これらの手法による実行速度向上を実験的に検証し、その結果逐次実行に対して最大で 55.3 倍の大幅な高速化を達成した。本手法は再帰的演算を含む分析的処理一般に対して適応可能な手法であるため、今後は EFIM 以外の処理に対しても本手法を適用していきたい。

謝　辞

本研究の一部は，日本学術振興会科学研究費補助金基盤研究 (B) JP20H04191 の助成を受けたものである。

文　献

[1] Raymond Chan, Qiang Yang, and Yi-Dong Shen. Mining high utility itemsets. In *Proc. ICDM'03*, pp. 19–26, 2003.

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

[3] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Tin Truong-Chi, and Roger Nkambou. *A Survey of High Utility Itemset Mining*, pp. 1–45. 2019.

[4] Philippe Fournier-Viger, Cheng-Wei Wu, Souleymane Zida, and Vincent S. Tseng. FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Proc. ISMIS'14*, pp. 83–92, 2014.

[5] Matt Godbolt. Optimizations in C++ compilers. *Communications of the ACM*, Vol. 63, No. 2, pp. 41–49, 2020.

[6] Genki Kimura, Yuto Hayamizu, Rage Uday Kiran, Masaru Kitsuregawa, and Kazuo Goda. Efficient Parallel Mining of High-utility Itemsets on Multi-core Processors. In *Proc. ICDE'23*.

[7] Mengchi Liu and Jun-Feng Qu. Mining high utility itemsets without candidate generation. In *Proc. CIKM'12*, pp. 55–64, 2012.

[8] Ying Liu, Wei-keng Liao, and Alok N. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Proc. PAKDD'05*, pp. 689–695, 2005.

[9] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proc. SYSTOR'11*, p. 12, 2011.

[10] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. SIGMOD'15*, pp. 1493–1508, 2015.

[11] Hong Yao, Howard J. Hamilton, and Liqiang Geng. A unified framework for utility based measures for mining itemsets. In *Proc. UBDM'06*, pp. 28–37, 2006.

[12] Souleymane Zida, Philippe Fournier-Viger, Jerry Chun-Wei Lin, Cheng-Wei Wu, and Vincent S. Tseng. EFIM: A highly efficient algorithm for high-utility itemset mining. In *Proc. MICAI'15*, pp. 530–546, 2015.

[13] 木村元紀, 合田和生, Rage Uday Kiran, 喜連川優. 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. In *DEIM2022*, pp. J34–2, 2022.

[14] 木村元紀, 早水悠登, ラゲ ウダイキラン, 合田和生, 喜連川優. NUMA 環境に於ける高効用アイテムセットマイニングの並列実行方式の検討と予備実験. WebDB 夏のワークショップ 2022, pp. DE2022–9, 13–18, 2022.