修 士 論 文

並列ハッシュ結合における実行時のデータの偏りの扱いに関する研究

指導教官　横田 治夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

土屋 由美子

1997 年 2 月 14 日

目 次

第 1 章

序論

現在、データベース分野では並列計算機を用いたデータベース管理システムの実現が主流となっている。これは、データベースモデルの主流であるリレーショナルデータベースのリレーショナル演算と並列実行がうまく適合し、またディスクに蓄えられたデータの並列な入出力が可能となり、性能向上が得られるためである。このため、これまでに並列計算機を用いたデータベース構築に関する研究が数多く行われてきた[DWT90]。並列計算機でのリレーショナル演算の実現アルゴリズムも多く研究されてきた。中でも結合演算は、他の演算に比べ高価であるため並列計算機を用いて効率良く実行するために今日でも様々なアルゴリズムが提案されている。これには、ソートマージ結合、ハッシュ結合などがある。並列ハッシュ結合は、結合属性の値の分散が均一な場合、スピードアップ、スケールアップともに優れるアルゴリズムである。並列ハッシュ結合はその効率の良さから、等結合演算実現の主流となっている。

しかし、現実のデータベース内のデータには偏りが存在する[WAL91]。例えば、図書目録データベースの属性値の分散は Zipf 分散に似た不均一分散を示す。このような不均一分散データに対して通常のハッシュ結合アルゴリズムを適用すると、その性能は劇的に低下する。このため、データの偏りを考慮した並列結合アルゴリズムが近年の課題となっている。

多くのデータ偏りを扱う並列結合アルゴリズムは、結合属性における偏りの度合を調べるために、通常のアルゴリズムにサンプリングやスキャンを追加し、その結果を静的に解析し、ノードへの適切な処理分配を行う[KIT90][WOL91][DWT92]。この処理により、各ノードの負荷をほぼ均等にする事が可能となったが、1 つの（またはごく僅かな）高偏り値がある場合や、静的な結果見積りが誤る場合には負荷の不均衡が発生し、著しい性能低下を被る。

このため、各ノードの部分結合実行中に負荷を監視し、著しい過負荷を持つノードの処理を動的に他の軽負荷ノードへ移送する、動的な偏り制御アルゴリズムが提案されるようになった[SHA93][HAR95]。[HAR95]の提案アルゴリズムでは、各ノードの負荷の監視のためにコーディネータを置き、タイマを使って、各ノードの処理情報を一定時間毎に調べている。このため、たとえ静的な見積りが大きく誤ったり、高偏り値を割

り当てられたノードが他のノードに比べ著しく高負荷になってしまう場合でも、効果的に負荷を再均衡させる事ができる。しかし並列計算機のノード数が大幅に増えた場合、コーディネータへの情報および処理の集中が問題になる事が考えられる。上の提案アルゴリズムでは、コーディネータの置き方や数には触れていない。

本研究では、既存のデータ偏り制御並列ハッシュ結合アルゴリズムによるデータ偏りの影響の抑制について調べる。また、既存の偏り制御アルゴリズムに存在するコーディネータへの情報や処理の集中の影響についても調べる。さらに、各ノードでの部分結合処理中にローカルに過負荷の検出を行うための方法についても検討を行い、並列ハッシュ結合におけるデータの偏りを扱う効果的な負荷均衡手法について考察する。

このため、まずデータ偏り制御を行わない並列ハッシュ結合でのデータ偏りの影響を調べる。次に、既存のデータ偏り制御アルゴリズムとして、再分散偏りを扱うアルゴリズム、動的に結合生成偏りを扱うアルゴリズムを実験環境上へ実装し、データ偏り制御を行わない並列ハッシュ結合と比較してデータ偏り性能について検討する。さらに、動的な結合生成偏り制御アルゴリズムについて、コーディネータノードの置き方について比較・検討を行う。

論文の残りの部分の構成は次のようになっている。まず 2 章で並列ハッシュ結合アルゴリズムとデータ偏りについて述べる。3 章では、既存のデータ偏り制御アルゴリズムについて述べる。次に 4 章で本実験環境での既存の並列結合アルゴリズムの実現について述べる。各アルゴリズムの実験結果もこの章に記す。5 章ではコーディネータノードを分散配置した並列結合アルゴリズムについて、その方針、実現方法、実験結果について述べる。最後に 6 章で並列ハッシュ結合におけるデータ偏りの扱い方について結論をまとめる。

第 2 章

並列ハッシュ結合アルゴリズム

ここでは伝統的な並列ハッシュ結合アルゴリズムと結合処理におけるデータ偏りについて述べる。並列結合演算にハッシュを適用したアルゴリズムは [KIT83] で提案された。[KIT83] では、複数のメモリバンクと処理ノードからなる構成上で結合負荷を減少させる、ハッシュを用いた効果的な並列結合アルゴリズムについて述べている。また、この並列ハッシュアルゴリズムをディスクを用いて実行する際に、結合タップルを各処理ノードへ分散する処理に、あるバケットのハッシュテーブルの構築およびテーブルのプローブをオーバラップさせて並列ハッシュ結合を効率良く行う Hybrid ハッシュ結合アルゴリズムが [DWT90] で提案された。この 2 つのアルゴリズムは等結合処理の実現法として広く用いられている。ここではこの 2 つのアルゴリズムについて説明する。

並列ハッシュ結合は結合属性が均一に分散する際、並列実行で性能の指標となるスケールアップ、スピードアップ共に優れるアルゴリズムである。しかし、このアルゴリズムはハッシュ関数を用いた負荷の分割を行っている。従って、結合属性が均一に分散しない場合、特定のハッシュ値に属性値が集中する事がある。この場合、このハッシュ値に相当するパーティション（バケット）を処理するノードは著しい負荷を被り、結合処理全体の性能も劇的に低下する。このため、並列結合アルゴリズムの研究分野でもデータ偏りに関する研究が進められるようになった。この中で [WAL91] は、並列結合演算におけるデータ偏りの分類を行った。この分類では、データ偏りに関する類似した概念の区別が行われている。研究を進める上でこの分類は重要なため、これについてもここで説明する。

2.1 主メモリのみを使った並列ハッシュ結合

[KIT83] では、結合領域環境で高性能なデータベースマシン GRACE の開発について述べている。GRACE はハッシュとソートに基づく知的リレーショナル処理を行う。[KIT83] で述べられている、データベースマシンへのハッシュの応用を以下に示す。

ハッシュは動的なクラスタリングを行う事ができる。これを結合演算に適用すると、結合負荷そのものを減らす事ができる。すなわち、結合属性のハッシュ値によってタップルを別々の集合（これをバケットという）にグループ分けすると、異なるハッシュ値を持つバケット内のタップルとは結合処理 — 比較、適合タップルの結合処理 — を行わなくてよい。つまり、それぞれサイズ N, M のリレーションの結合演算の逐次実行の場合の総処理時間 T は次の様になる。

$$\begin{aligned} N &= \sum_{i=1}^{s} n_i, M = \sum_{i=1}^{s} m_i \\ T &\propto \sum_{i=1}^{s} n_i \times m_i \end{aligned}$$

ただし s はバケット数、n_i, m_i は i 番目のバケットのサイズとする。一方、クラスタリングを行わない方法では $T \propto N \times M$ となる。このようにクラスタアプローチは普通のノンクラスタアプローチに比べ劇的に負荷を減らす事ができる。

並列計算機でクラスタリングアプローチを処理する場合、あらかじめ複数のメモリバンクへデータベースをストアしておき、複数のノードで並列にバケット処理を行う。[KIT83] では、ハッシュしたバケットのマルチメモリバンクへの論理的なマッピング法により、次の 2 つのアプローチが提案されている。

- バケット集中アーキテクチャ(bucket converging architecture)
  あるバケットの全タップルは 1 つのメモリバンクだけにストアされる。
- バケットスプレッディングアーキテクチャ(bucket spreading architecture)
  一つのバケットのタップルが多くのメモリバンクへストアされる

ここではバケット集中アーキテクチャについて説明する。

結合演算実行前、リレーションは複数のソースメモリバンク中にストアされている。ここから、並列にタップルをリードし、その結合属性値にハッシュ関数を適用し、同じハッシュ値を持つ全タップルが一つのバンクに集まるようにする。あるメモリバンクのタップルは他のバンクのものとは結合されないので、各バンクを処理するプロセッサは独立に処理を行える。

しかし、このアーキテクチャではハッシュの非均一性によりメモリバンクオーバフローを発生させてしまう、つまり、あるバンクに集まったタップルがそのメモリバンクの容量を越えてしまう事がある。これはメモリ管理を難しくする。

2.2 ディスク利用を前提とした並列ハッシュ結合

[DWT90] では Hybrid ハッシュ結合アルゴリズムを提案している。Hybrid ハッシュ結合アルゴリズムはデータ偏りのない場合、最も効率の良いアルゴリズムとして知られている。集中型、並列型の Hybrid ハッ

シュ結合アルゴリズムについて以下に記述する。

集中型 Hybrid ハッシュ結合アルゴリズムは 3 つのフェーズを持つ。

1. インナーリレーション（小さい方のリレーション）R にハッシュ関数を適用して N 個のバケットに分割する。バケット 1 に属するタップルを使いメモリ上にハッシュテーブルを作る。残りの N - 1 個のバケットは、一時ファイルに記憶される。細かい (fine) ハッシュ 関数を用いて、各バケットのタップルがメインメモリ全体に収まるのに、ちょうど良い数のバケットを生成する。

2. フェーズ 1 のハッシュ関数を使って他方のリレーション（アウターリレーション）S を分割する。この時、バケット 1 に属するタップルでフェーズ 1 で作ったメモリ上のハッシュテーブルをプローブする。残りの N - 1 個のバケットは一時ファイルに記憶される。

3. 残りの N - 1 個のバケット対をそれぞれ結合する。

このように結合演算はより小さい結合の系列に分解される。この分割した各々はうまくいけば結合オーバーフローを調べなくても実行する事が可能である。インナーリレーションのサイズがバケットの数を決定する。この計算はアウターリレーションのサイズとは独立に行える。

一方、並列バージョンの Hybrid ハッシュ結合アルゴリズムも上で記述した集中型のアルゴリズムと同様に行うことができる。この場合、結合する 2 つのリレーションは “パーティション分割テーブル” を用いて N 個の論理バケットに分割される。バケットの数は、各論理バケットに対応するタップルが結合処理を行うプロセッサの集約メモリに収まるように選ばれる。集中型では一つのディスクに置かれていた N - 1 個のバケットは、利用可能なすべてのディスクサイトに分割して置かれる。

次に “結合分割テーブル” を用いて、あるバケットに属するタップルを各結合処理プロセッサ（これらのプロセッサはディスクを持っている必要はない）へ振り分ける。つまり、結合処理のフェーズを並列化している。更に、インナーリレーション R のバケットへの分割は、各結合処理ノードでの R のバケット 1 のタップルをメモリ常駐ハッシュテーブルへの挿入と同時に行われる。更に、アウターリレーション S のバケットへの分割は、S のバケット 1 と R のバケット 1 との結合処理と同時に行われる。

このために、バケット 1 のタップルは結合を行うプロセッサへ送られなければならないので、R と S のパーティション分割テーブルを結合分割テーブルも含むように拡張する必要がある。残りの N - 1 個のバケットを結合処理する時には、結合分割テーブルだけが必要である。

2.3 データの偏り

上で述べたような並列ハッシュ結合アルゴリズムは、結合属性値へのハッシュ関数の適用により結合処理コストを効果的に減らし、[DWT90] では Hybrid ハッシュ結合が線形のスピードアップ・スケールアップ

表 2.1: 属性値偏り

att1	att2	att3
1	1	1
1	5	3
1	9	5
10	13	7
15	17	9

特性を持つ事が示されている。しかし、これは均一仮定 (uniformity assumption) に基づいて示された性能である。

並列結合演算における均一仮定とは次の 2 つの仮定である。

- 結合のどのステージでもタップルは各処理ノードへ均一に分散される
- リレーションの結合属性値はどの値も同じ頻度でタップルに出現する

一般のデータにはデータ偏り (data skew) が存在する事が [WAL91] などで指摘されている。例えば、図書目録データベースの属性値の分散は Zipf 分散に似た不均一分散を示す [LYN88]。このような不均一分散データに対して通常のハッシュ結合アルゴリズムを適用すると、その性能は劇的に低下する [WAL91]。

以下に [WAL91] で述べられている、並列結合演算に生じる各種のデータ偏りについての分類を示す。

データ偏りは大別すると属性値偏り (attribute value skew) とパーティション偏り (partition skew) とになる。属性値偏りは、属性値が均一に分散していない事を指す。つまり、ある属性値において、特定の値の出現が他の値の出現に比べ突出して起こる事を指す。例えば、表 2.1のリレーションの att1 は 1 が繰り返し出現していて、属性値偏りが発生している。

属性値偏りは値の性質としてリレーションに内在する不均一であり、単一プロセッサの場合においても存在する。属性値偏りを持つリレーションは、均一な分散のリレーションに比べ、より高い結合選択率 (join selectivity) を持ち、より大きい結合結果を生成する。この結合結果増加分の負荷の増加は避けられない。従って、並列結合アルゴリズムでは、各ノードの負荷を均衡させる事しか行えない。

一方、パーティション偏りは並列実装において、各処理ノードの間で負荷が不均衡な場合発生する。従ってこの偏りは並列実装でのみ発生する。また、この偏りは結合演算の実装方式に依存して起こる。たとえ入力データが均一であっても、この偏りは発生する事がある。パーティション偏りは、結合アルゴリズムのどのフェーズで発生するかにより、更に次の 4 つに分けられる。

タップル配置偏り (tuple placement skew) タップルの初めの分散がノードにより異なる場合に起こる

選択率偏り (selectivity skew) 選択述語の選択率がノードにより異なる場合に起こる

再分散偏り (redistribution skew) 結合属性値の分散と再分散メカニズム（ハッシュ関数など）の予期する分散が異なる場合に起こる。

結合偏り (join product skew) 各ノードでの結合選択率 (join selectivity) が異なる場合に起こる。これはリレーションペアの特性であり、リレーションが結合されるまで現れない偏りである。

並列ハッシュ結合演算における各偏りの影響について検討する。以下では各種の偏りが単独に発生し、また、一つのノードに負荷が集中し他のノードは同じ負荷を持つ物として考える。

まず、タップル配置偏りについて考える。この場合、特定のノードの担当するディスクに含まれるリレーションの一部が、他のノードに比べ多くなる。このため、このノードはより多くのタップルをリードし、選択述語を実行しなければならない。高偏りノードの処理タップルが他のノードに比べ x タップル多いとするとこの処理のためのコスト c は $c \propto x$ となる。

次に、選択率偏りについて考える。ハッシュ結合では、各ノードがローカルなディスクからタップルをリードしそのタップルに選択述語を適用し、特定の範囲に属する属性値を持つタップルを取り出し、その取り出されたタップルに対してハッシュ関数を適用し、ハッシュ値に応じてそのタップルを適切なプロセッサへ転送しなければならない。高偏りノードの処理タップルが他のノードに比べ r 倍のタップルの選択率を持つとすると、そのノード選択後のタップル数は他のノードに比べ r 倍になる。他のノードの選択後のタップル数が y とすると、高偏りノードは他ノードに比べ $c \propto y(r-1)$ のコストを負う。

次に、再分散偏りについて考える。ハッシュ関数により分割されたそれぞれのバケットは各プロセッサに割り当てられる。再分散偏りにより特定のノードに多くのタップルが集中してしまうと、このノードはより多くのタップル受信処理、バケット結合処理を行わなければならない。また、再分散偏りが起こると、たとえ結合偏りが発生しなくてもより多くの結果タップルが生成され、このタップルのディスクへの出力処理も他のノードに比べ重くなる。高偏りノードの受けとったタップル数が他のノードに比べ x タップル多いとすると、このノードの余分なタップルの処理のためにコスト c は $c \propto x$ となる。

最後に結合偏りについて考える。結合演算ではある条件に当てはまる 2 つのタップルをマージして結果タップルを生成し、それを出力する。この際、どの位の確率でマッチタップルが生成されるかを表すのが結合選択率である。もしあるノードの結合選択率が他のノードの選択率の r 倍だとすると、そのノードで生成されるマッチタップル数は他のノードの r 倍になる。他のノードの生成マッチタップル数が y とすると、高偏りノードは他ノードに比べ $c \propto y(r-1)$ のコストを負う。

これらの偏りのうち、並列ハッシュ結合と特に強く結び付くのは再分散偏りと結合偏りである。このため、提案されている並列ハッシュ結合における偏り制御アルゴリズムの多くは、この 2 つの偏りの解決を扱う。次の章で、再分散偏りおよび結合偏りを扱う既存の並列ハッシュ結合アルゴリズムについて述べる。

第 3 章

既存のデータ偏り制御手法

これまでにいくつかの並列ハッシュ結合におけるデータ偏りの影響を取り除くためのアルゴリズムが提案されている。この章では、こういったアルゴリズムについて述べる。

初期の研究では、再分散偏りを防ぐことにより、データ偏りの影響を抑え、各プロセッサの負荷を均一にする試みが行われた [KIT90][HUA95]。これらの試みは、ハッシュ関数の適用によってできたバケットのサイズを調べ、これに基づいて動的にバケットを処理プロセッサへ割り当てることで、再分散偏りを抑えている。[HUA95] では Zipf-like 分散によるデータ偏りモデルを用いたシミュレーションで、提案アルゴリズムとデータ偏り制御を行わない並列ハッシュ結合（GRACE）との比較を行い、提案アルゴリズムが広い範囲のデータ偏りに有効であることを示している。

しかし、再分散偏りを取り除くことができても、結合偏りによりノード間の処理時間に偏りが発生することがある。この問題に対するアプローチとして、各バケットの結合によって生成されるマッチタップルの数を静的に見積り、その見積り値を用いて各バケットの処理時間を計算し、それに基づいてバケットのプロセッサへの割当を決定する並列結合アルゴリズムがいくつか提案された [DWT92][WOL91]。また [SHT93] では、共有仮想メモリ機構 (SVM) を用いた負荷の共有 (load sharing) により、パーティション偏りを解決している。[HAR95] では、各プロセッサがバケット結合処理の様子をコーディネータが監視してより柔軟な負荷の共有を行うアルゴリズムを提案している。

以下では再分散偏りを扱う結合アルゴリズムとして [KIT90] のバケットスプレッディングアルゴリズムと [HUA95] の適合アルゴリズムを、静的に結合偏りを扱うアルゴリズムとして [WOL91] の階層ハッシュを用いた結合アルゴリズムを、動的に結合偏りを扱う結合アルゴリズムとして [HAR95] の動的な結合偏り制御アルゴリズムを説明する。

3.1 再分散偏りの制御アルゴリズム

ここではまず、再分散偏りを扱う並列結合アルゴリズムとして [KIT90] のバケットスプレッディングアルゴリズムを説明する。通常の並列ハッシュ結合はプロセッサへのハッシュバケットの割当を静的に行うのに対し、このアルゴリズムではこれをバケットサイズに応じて動的に行って再分散偏りを取り除く。

動的にバケット割当を決める場合、各プロセッサが結合属性にハッシュ関数を適用してできたサブバケットを直接、処理プロセッサへ転送する事ができなくなり、一時的にそのタップルをどこかへ置かなければならない。このアルゴリズムでは、各バケットを全プロセッサへ分散させる事でこの問題を解決している。つまり、バケットはプロセッサ数と同じ数に分けられ（このそれぞれをサブバケットという）、各サブバケットは 1 つのプロセッサに置かれる。

このように全プロセッサがリレーションの分割処理を行い、その処理を終えると、各プロセッサ内でのサブバケットの分散は、システム全体でのバケットの分散の様子を反映している。従って、適当に一つのプロセッサをマスターに選び、ローカルなサブバケット分散からバケットサイズに基づいた、バケット割当を決定できる。

動的なバケット割り当てのためにこのアルゴリズムでは、バケットサイズ調整 (bucket size tuning)[KIT83] で複数のバケットを組み合わせて、ほぼ同じサイズのバケットを作り、それをサイクリックバケット割り当て法で各プロセッサへ割り当てていく。

再分散偏りを扱う並列ハッシュ結合アルゴリズムは [HUA95] でも述べられている。[HUA95] では、 3 種類の再分散偏り制御並列ハッシュアルゴリズムが提案されている。ここではそのうち適合アルゴリズム (Adaptive Load Balancing Parallel Hash Join, ABJ) を説明する。

ABJ では、各処理ノードが自分のローカルディスク内にあるリレーションの一部を並列にハッシュして、サブバケットに分け、それを再びローカルディスク内へライトする。この処理の後、各バケットの分散が計算され、その情報を基にバケットのノードへの割当が決められる。この割り当てに従い、各サブバケットは対応するノードへ転送される。最後に、こうして集められたバケットの結合を各プロセッサがローカルに行う。この詳細な流れを以下に示す。

1. **分割** 各プロセッサはローカルディスクから結合リレーション R, S の一部を読みだし、ハッシュを行い、これらをサブバケットに分ける。サブバケットは再びローカルディスクへストアされる。

2. **パーティション調整** 各プロセッサはある決められたコーディネータに自分の持つサブバケットのサイズを報告する。コーディネータは次の方針に従って、プロセッサへバケットを割り当てていく。

   a. バケットペア（R と S の対応するバッケトの対）をそのサイズで降順にソートする

b. このバケットペアがソートされた順にプロセッサへ割り当てられていく。各バケットペアは、一番大きいサブバケットを持つプロセッサへ割り当てられる。つまり、最大のサブバケットペアは今置かれているプロセッサ P_i に残る（他のプロセッサへ転送しなくてよい）。そして他のプロセッサからこのバケットペアに対応するサブバケットが P_i に集められる。この時、P_i のサイズ（割り当てタップル数）は新しいバケットペアの追加を反映して更新される。もしあるプロセッサ P_i が次の条件を満たすなら、そのプロセッサはバケット割り当ての対象外となる。

$$\sum_{j=1}^{n_i} |B_{ij}| \leq \frac{|R|+|S|}{N} \quad and \quad \sum_{j=1}^{n_i+1} |B_{ij}| > \frac{|R|+|S|}{N}$$

ここで n_i はプロセッサ P_i に割り当てられたバケットペアの数を表す。

この処理を全てのプロセッサが割り当て対象外になるまで繰り返す。この時、まだ残っているバケットがあればベストフィット減少法（best fit decreasing strategy）を使って、プロセッサへ割り当てる。プロセッサへのバケットの割当が決められると、割り当て情報は全プロセッサへブロードキャストされ、サブバケットがローカルバケットを作るためにそれぞれのバケットの割り当て先に集められる。

3. **バケット調整** 各プロセッサは小さいバケットを組み合わして、より最適なサイズの結合バケットを作る。

4. **結合フェーズ** 各プロセッサはそれぞれのバケットペアをローカルに結合する

上でベストフィット減少法とは [HUA95] で述べられているバケット調整 (bucket tuning) のためのアルゴリズムで、最大のバケットを最小のパーティションに追加していき、各パーティションのサイズをほぼ等しくするアルゴリズムである。

3.2 静的な結合偏り制御アルゴリズム

再分散偏りの制御法では、ハッシュバケットのサイズが均衡するようにしてプロセッサへの割当を決めた。しかし、この方法では結合偏りが発生する場合に対応できない。そこで、これを扱うために [WOL91] ではハッシュバケットの割当を各バケット結合処理時間の見積りに基づき行うアルゴリズムを提案した。このアルゴリズムの流れを以下で説明する。

1. **ハッシュフェーズ** 両リレーションをハッシュして、粗い (coarse) ハッシュパーティションを作り、細かい (fine) ハッシュパーティションの統計情報（サイズ）を集める。

   粗いハッシュパーティションの数はプロセッサ数の倍数とし、この粗いハッシュパーティション毎に、第二の細かいレベルのハッシュパーティションが作られる。各細かいハッシュパーティション毎にマップされるタップル数のカウントが取られる。また、ハッシュされたタップルは粗いパーティション毎にディスクへ書き込まれる。

2. **スケジューリングフェーズ** ここでは結合実行を部分タスクに分ける。この部分タスクが各プロセッサへ割り当てられる。割り当てはハッシュフェーズの細かいハッシュパーティションのカウントを用いて決められる。また、一つのパーティションが複数のプロセッサへマッピングされる事もある。

3. **転送フェーズ** 各ハッシュパーティションのタップルを割り当てプロセッサへ送る

4. **結合フェーズ** 各プロセッサがパーティションをローカルディスクから読みだし、ハッシュ結合を行う。結果はディスクへ出力される。

上のスケジューリングフェーズで各バケット結合（タスク）の処理時間が見積もられる。この詳細は 4.5 の静的な結合偏り制御アルゴリズムの実験内容で述べる。

3.3 動的な結合偏り制御アルゴリズム

結合偏りを扱うアルゴリズムとして [HAR95] で述べられている動的な結合偏り制御アルゴリズムを以下で説明する。

このアルゴリズムでは、再分散フェーズの前に各バケット（パーティション）の結合処理時間を静的に見積り、それを用いて各ノードの処理時間がほぼ等しくなるようにバケットの割当を決定する。更に、静的な見積りが誤る場合に備えて、コーディネータが各パーティションの結合処理を監視する。もし、性能低下を引き起こすような見積り誤りが検出されると、すなわち、特定のノードが他のノードに比べて著しい過負荷を被っている事が検出されると、この過負荷を他の軽負荷ノードへ移送して、各ノードの負荷を再均衡させ、静的な見積り誤りによる性能低下を抑える。

この結合偏り制御アルゴリズムの処理の流れは以下のようになる。

1. スキャン / サンプリング フェーズ
   コーディネータは結合属性の統計情報を集める

2. スケジューリング フェーズ
   集めた情報に基づいて、コーディネータは各パーティションの結合実行時間の見積りと、パーティションのプロセッサへの割当を行う

3. 再分散 フェーズ
   2 の結果に基づき、タップルの交換とパーティションの構築を行う

4. 結合 フェーズ
   各プロセッサでローカルパーティションペアをリードし、結合する。コーディネータは各プロセッサでのパーティション処理を監視し、見積り通りに結合が行われているか調べ、必要ならば動的に負荷補償方針（workload compensation strategy ）を呼び出し、各ノードの負荷を再均衡させる。

結合フェーズまでの処理の具体的な内容は、以前の研究に現れている [WOL91]。ここでは結合フェーズの詳細について述べる。N 台のプロセッサ $P_i\,(i = 1, ..., N)$ からなる無共有データベースシステムにおける、リレーション R と S の結合を考える。各プロセッサ P_i には、パーティション $(R_{ij}), (S_{ij})(j = 1, ..., m_i)$ が割り当てられているとする。ここで、(R_{ij}) はハッシュテーブルを構築（ build ）するのに使われ、(S_{ij}) は、そのテーブルをプローブ（ probe ）するのに使われる。以下では、多くのパーティションの結合はスケジューリング フェーズでの見積り通りに実行され、わずかなパーティションの結合でミスマッチが検出されると仮定する。

結合 フェーズの間、コーディネータは各パーティションの結合をモニタするために、プロセッサ毎にタイマを管理し、パーティション処理の統計情報を集める。パーティション処理の統計情報を得るためにコーディネータは負荷を調べるプロセッサにシグナルを送る。シグナルを受けとったプロセッサは、今までに処理し終わったプローブタップル (S_{ij}) の量 (ΔS_{ij}) を返す。コーディネータは (ΔS_{ij}) を受けとると、その値とタイマーの値から、1 プローブタップル当たりの処理時間（下式左辺第 1 項）を計算し、見積り値（下式左辺第 2 項）と比較する。もし以下の式を満たすならプロセッサ P_u は過負荷と見なされる。

$$\frac{T^*_{meas}(R_{uv}, \Delta S_{uv})}{Size(\Delta S_{uv})} - \frac{T^*_{est}(R_{uv}, S_{uv})}{Size(S_{uv})} > \alpha \tag{3.1}$$

ここで、$T^*_{est}(R_{uv}, S_{uv})$ は (S_{uv}) をリードし、ハッシュテーブルをプローブし、マッチタップルをライトする時間の見積り値である。$T^*_{meas}(R_{uv}, \Delta S_{uv})$ は ΔS_{uv} 個のタップルでこれを行うのにかかった実際の時間の測定値である。また $Size(S_{uv})$ は S_{uv} のタップル数である。パラメータ α の値はシステム構成と、許容する偏りの量に依存する。また、動的な負荷均衡はいくらかのオーバヘッドを起こすので α はこのオーバヘッドコストを考えて決める。

更に、(R_{uv}) と (S_{uv}) の結合の測定値と見積り値の誤差 (deviation) $dev(R_{uv}, S_{uv})$ を以下のように定義する。

$$dev(R_{uv}, S_{uv}) = T_{meas}(R_{uv}, S_{uv}) - T_{est}(R_{uv}, S_{uv})$$

ここで $T_{est}(R_{uv}, S_{uv})$ は (R_{ij}) をリードし、ハッシュテーブルを作り、(S_{ij}) をリードし、そのハッシュテーブルをプローブし、マッチタプルをライトする時間の見積り値である。$T_{meas}(R_{uv}, S_{uv})$ はこれを行うのに実際にかかった時間の測定値である。

(3.1) で検出されるには小さすぎる誤差の累積によりプロセッサ P_u でのパーティションペアの結合処理がスケジューリングフェーズで見積もられたより遅れて始まる場合も過負荷を検出する。すなわち次式を満たす場合もプロセッサ P_u は過負荷であると見なされる。

$$\sum_{j=1}^{v-1} dev(R_{uj}, S_{uj}) > \beta \tag{3.2}$$

パラメータβの値は αと同様にシステム構成と許容する偏りの量に依存する。

P_u の過負荷が検出されると、コーディネータは P_u における負荷の再見積りを行う。これに基づき移送過負荷量 $M(R_{uv}, S_{uv})$ を計算し、P_u の過負荷を軽負荷ノードに $1/N \cdot M(R_{uv}, S_{uv})$ ずつ移送する。

軽負荷ノードへの移送には次の 2 つの方法が提案されている。

- 結果再分散 (result redistribution)
- タスク処理移送 (task processing migration)

過負荷の移送を行う場合、まず P_u が $P_i(i \neq u)$ へ結果再分散を行うようにスケジューリングされる。一方、P_i は自身のローカルパーティション結合を終え、メモリが空いた状態になると、コーディネータにそのことを告げる。1 プローブタップル当たりの生成マッチタップル数を表す爆発率 (Browup ration) $B(R_{uv}, S_{uv})$ が十分大きい場合、このプロセッサへの移送はタスク処理移送へと移行する。もし、タスク処理移送を行うのに十分に爆発率が大きくない場合には P_i は結果再分散を続け、同時に次のローカルパーティションの結合を始める。

P_i へ $1/N \cdot M(R_{uv}, S_{uv})$ の過負荷の移送がなされると、そのことがコーディネータに報告され、P_i は自身のローカルパーティションの結合へ戻る。また、プロセッサが自分のローカル結合を全て終えると、そのことがコーディネータに報告され、まだ実行中のプロセスの負荷の一部を処理するようにスケジューリングされる

3.4 既存のデータ偏り制御法のまとめ

この章では既存のデータ偏り制御を行う並列ハッシュ結合アルゴリズムについて述べた。まず、再分散偏りを扱うアルゴリズムとして [HAR90] のバケットスプレディングアルゴリズムと [HUA95] の適合アルゴリズムを紹介した。これらは、各バケットをそのサイズに基づき再分散偏りが発生しないように、各処理プロセッサへ割り当てていく。これらアルゴリズムは結合リレーションの一方に偏りの発生する単一偏り (single skew) においては頑強な性質を示した。しかし、これらアルゴリズムは単に各プロセッサに均等な数のタップルが割り当てる事を試みるに過ぎない。たとえば、過剰に大きなサイズを持つバケットがある場合、特にそれが単一の属性値による場合には、どのような方法でもこのバケットによる再分散偏りは防げない。または結合偏りが発生する場合には対応する事ができない。

次に、このような結合偏りをバケットサイズ以外の情報も収集して見積り、それを反映した方法でバケット割当を決定する静的な結合偏り制御アルゴリズム [WOL91] について説明した。多くの場合、このアルゴリズムは各バケットに均等な負荷を与える事ができる。しかし、この静的な結合制御アルゴリズムは結合時間の見積り能力に大きく依存しているので、この見積りが誤る場合は結合生成偏りを制御する事が保証さ

れない。もし予想外の負荷が発生すれば、そのバケットの結合を行っているプロセッサは他のプロセッサに比べて重負荷となる。

最後に、動的な結合偏り制御アルゴリズムとして[HAR95] を紹介した。このアルゴリズムでは、結合時間の見積りに基づいてバケット割当を行い、なおかつ実際のバケット結合の処理状況を監視し、見積り誤りが発生していないか調べる事によって、静的な結合偏り制御アルゴリズムの問題を解決する。[HAR95] のアルゴリズムはこの処理を実現するために、コーディネータが各バケット処理およびプロセッサ状態の情報収集を行い、予測されなかった過負荷の検出を行い、それを移送先のプロセッサ状態に基づいた方法で移送し、負荷の再均衡を達成する。[HAR95] では静的な結合偏り制御アルゴリズムと提案アルゴリズムを比較し、予測されなかった過負荷が存在する場合でもそれを移送により再均衡できる事を示している。

効果的な負荷移送を行うために、システム全体の状態を把握した単一のコーディネータを置く事は有効な方法である。しかしこの場合、大量の情報および処理の集中というリスクを単一のプロセッサが負わなければならない。ある程度の規模を越えると、そのリスクが効果的な移送による利点を上回る事が考えられる。そこで、本研究ではこのような場合により強力である事が予測される、コーディネータの分散配置による動的な結合偏り制御アルゴリズムを検討する。

第 4 章

予備実験

本研究は KLIC という並列論理型言語を用いてアルゴリズムを記述し、それを nCUBE/2 上で実装して、そのアルゴリズムの性能の評価を行う。KLIC の組み込みデータ型を用いた場合、大きなサイズのデータをプロセッサ間で移動させると異常動作により安定した実験が行えないため、かなり小さいサイズ（500 – 800 タップル）の実験リレーションを用いて実験を行う。リレーションが小さいと、偏りの影響が小さく、その制御を行うアルゴリズムの能力が十分に現れない可能性がある。

本章では、既存のデータ偏り制御アルゴリズムの実験環境での特徴を検討するために、前章までに述べた各偏り制御アルゴリズムについてその主要な箇所を nCUBE/2 上へ実装し、その能力について調査した結果を示す。始めに、予備実験における共通事項として実験環境、実験の前提について説明する。次に、並列ハッシュ結合に対するデータ偏りの影響について調べた結果について述べる。次に、最分散偏り制御に関する実験として各バケット割り当て方式による応答時間の変化を調べた結果を示す。最後に、静的な結合偏り制御に関する実験として各バケットの結合により生成されるマッチタップル数の見積り法と、その見積り能力について調べた結果を示す。

4.1 実験環境

本研究は並列計算機 nCUBE/2 を用いて実験を行った。また、アルゴリズムの記述には並列論理型言語 KL1 を用いた。また、データベース処理環境として本研究室では Parade という並列データベースシステムを開発している。これらについて以下で述べる。

4.1.1 nCUBE/2

本研究で用いた nCUBE/2 は 256 台の 64bit CPU を持つ並列計算機である。各 CPU は 16M(ノード 0-15) または 4M(ノード 16–255) のメモリを持ち、ハイパーキューブ結合で相互に結ばれている。ディス

クへのアクセスは I/O ノードを介して行われる。I/O ノードの内、ディスクアクセスを扱うのはディスクサーバと呼ばれる 8 台のプロセッサである。各ディスクサーバは 1G のディスクを 2 組持っており、全体では 16 台のディスクが使用可能である。

4.1.2 KL1

KL1 はガードホーン節に基づく、並行論理プログラミング言語である。KL1 の構文と意味は非常に単純で簡潔であるが、並行計算向けの非常に強力な機能を提供している。

KL1 は記号アトム機構や自動メモリ管理機構などの記号処理に必要とされる機能を持ち、またデータフロー同期機構による同期の自動化によりプログラムの並列動作を前提としている。このため、プログラマは、記号処理のための複雑なデータ構造の表現法やメモリ管理、そして並列処理のための並列実行部の指定や同期処理といった問題を言語にまかせる事ができる。従って、より本質的なプログラミングに集中できるようになる。

また並列処理に関しては、物理的な並列実行の指定にはプラグマと呼ばれる記述によって行なえる。同じプログラムのプラグマ部分を変更するだけでさまざまな並列実行の仕方を指定できる。プラグマはプログラムの正当性を変えないように設計されており、並列処理の仕方の変更によりデバッグを繰り返さなくてもよい。

KL1 の処理系として KLIC がある。KLIC は KL1 プログラムを C 言語にコンパイルし、さらにこの C 言語プログラムを C コンパイラで実行形式のコードへ変換する。C 言語は各種計算機上で広く使われている言語である。KLIC は KL1 を C 言語へ一度変換する事によって、C コンパイラを持つ各種計算機上での KL1 実行を可能にしている。

4.1.3 Parade

リレーショナルデータベース処理環境として本研究室では、Parade(Parallel Active Database Engine) というアクティブデータベースシステムを開発している。Parade はリレーショナル問い合わせ言語 SQL をサポートしている。Parade は上で述べた KL1 で記述されており、各種計算機への移植が容易である。

本研究で扱ったハッシュ結合をより一般的な状況での使用を仮定して実験を行うために、Parade へ追加する事を想定している。

4.2 実験の前提

4.2.1 プロセッサの割り当て

本論文での実験では特に断りがない限り以下の前提に基づいて行う。結合リレーションはあらかじめ 8 台のディスクに水平分割されている。i 番目のディスク D_i はプロセッサ P_j $(j = i \ mod \ N)$ がアクセスする。ただし $N(1 \leq N \leq 8)$ は結合処理を行うプロセッサの総数である。結果リレーション（マッチタップルの集合）は N 台のディスクに分割して格納される。この場合のプロセッサとディスクのマッピングも上と同様である。

実行プロセッサの指定には KLIC のゴール分散プラグマを用いた。ゴール分散プラグマは次のように指定する。

$$Goal@node(NodeID)$$

この記述により Goal の実行を（この Goal が展開されてできる SubGoal の実行も含め）NodeID で指定されたプロセッサで実行する事ができる。

また、タップルをバケットに分けるハッシュ関数 $h_1(x)$ にはモジュロを用いた。すなわち結合属性値 x を持つタップルはバケット $h_1(x) = x \ mod \ B$ に属する。ここで B はバケット数である。一方、各プロセッサがローカルに行うバケット結合の際にハッシュテーブルを構築するのに用いたハッシュ関数 $h_2(x)$ は $h_2(x) = (x/B) \ mod \ B$ とした。また、バケット数 B は 16 とした。

4.2.2 実験リレーション

実験では、各属性値は KLIC の整数アトムまたは文字列を用いて表現し、タップルはこれら属性値を要素とするリストで表現した。さらにリレーションはこれらタップルを要素とするリストとして表現した。

実験で用いたリレーションには属性値偏りを持たせている。属性によりその度合は異なり、属性値番号が大きい程、強い偏りを持つ。実験は次の 3 つの属性値偏りを持つリレーションを用いて行う。

- スカラー偏り : S_Skew
- zipf-like 偏り (バケットサイズ) : Z_Skew1
- zipf-like 偏り (属性値) : Z_Skew2

スカラー偏りは一般のリレーションのデータの分散を大げさにしたもので、より実験で扱いやすいように 1 つの結合属性値だけに大きな偏りを発生させる。zipf-like 偏りはより一般的なデータ偏りに近いモデルである。zipf-like 偏りによるデータ偏りモデルには、バケットサイズを zipf-like 分散で決めるものと各結合属性値の出現回数を zipf-like 分散で決めるものとがある。ここではその両方について述べる。

また、実験リレーションのサイズが小さいのは、実験プログラムを試作した分散版 KLIC では大きなサイズのデータを移動させると異常動作により安定せず、測定を行うリレーションサイズに制限を設けざるをえなかったためである。現在の実装ではタップルの移動にリストによるストリーム処理を利用しているが、今後、処理速度向上のためにジェネリックオブジェクトというユーザ定義型を用いて表現する事を計画しており、これと処理系の改善とによりこの制限がなくなる事を期待している。

スカラー偏り

$|R|$ タップルのリレーションを考える。このとき、各属性内のある固定された数のタップルを定数 1 とし、残りのタップルには 2 から $|R|$ を均等に分散させて作ったデータ偏りをスカラー偏り（scalar skew）[WDJ91] と呼ぶ。

このリレーションの使用には 3 つの利点がある。まず、何の実験が行われているのかを理解するのが容易である。第二に、データ偏りの度合を変化させても結果サイズを一定に保つのが容易である。最後に、Zipfian 分散の本質（多くの値は低頻度だが、僅かなデータが高い出現頻度を持つ）をとらえている。[Omi91]

本実験で用いるスカラ偏りリレーション S_Skew はサイズが 500 タップルで、各属性は x1, x10, x100, x200, x500 のスカラー偏りを持つ。“x” の後の値は、その結合属性値に値 1 の現れる回数である。残りのタップルは 2 から 500 の間でランダムに選ばれた値を持つ。例えば x10 属性はランダムに選ばれた 10 個のタップルに 1 が現れる事を意味する。残りの 490 個のタップルは 2 から 490 の間でランダムに選ばれた値を持つ。このリレーションの各属性をハッシュ関数によって分割した場合のバケット毎のサイズ（タップル数）を表 4.1 に示す。

zipf-like 分散偏り（バケットサイズ）

バケットサイズを zipf-like 分散で決めたリレーションは [HUA95] でシュミレーションモデルに用いられている。[HUA95] に述べられているこのリレーションの特徴を以下に述べる。

R を結合リレーションとする。R は B 個のバケット $R_1, R_2, ..., R_B$ にハッシュされる。 これらバケットのサイズは次のように zipf-like 分散によって決められる。

$$|R_i| = \frac{|R|}{i^z \sum_{j=1}^{B} \frac{1}{j^z}}$$

上式で z をバケット偏り (bucket skew) と呼ぶ。$z = 1$ なら上式は zipf 分散を表し、$z = 0$ なら均一分散になる。

本実験で用いる zipf-like 偏りリレーション Z_Skew1 は、$|R| = 500$ とし、上式でバケットサイズを決め、これに基づき各タップルの属性値を決めていった。この際、次の仮定を行った。

- $|R_i| \geq |R|/B$ ならバケット内の各値が均一に分散している

- $|R_i| > |R|/B$ なら 1 つの高偏り値があり、その他の値は均一に分散している

このようなデータ偏りを $z = 0, 0.2, 0.5, 0.8, 1$ について作成し、このそれぞれを各属性に対応させた。このリレーションの各属性をハッシュ関数によって分割した場合のバケット毎のサイズ（タップル数）を表 4.2 に示す。

zipf-like 分散偏り（属性値）

属性値に zipf-like 分散偏りをもつリレーションは [WOL91] のシミュレーションモデルに用いられていて、次の様な特徴を持つ。

この偏りモデルでは、各属性値の出現確率が zipf-like 分散に基づき決定され、この出現確率により各タップルの属性値が決まる。各値 $i(i = 1, 2, ..., D)$ の出現確率 p_i は次式になる。

$$p_i = \frac{1}{i^z \sum_{j=1}^{D} \frac{1}{j^z}}$$

本実験で用いる zipf-like 偏りリレーション Z_Skew2 はサイズが 500 タップルで、上の出現確率を用いて各タップルの値を決めた。具体的には、属性毎に次のようにな方法で属性値を決めた。まず、乱数 r を発生させる。それが値 i の出現確率 p_i よりも大きければその値をタップルに追加する。これを必要なタップル数がそろうまで $i(i = 1, 2, ..., D)$ について繰り返し行って、ある属性値に含まれる属性値集合を決定した。

このようなデータ偏りを $z = 0, 0.25, 0.5, 0.75, 1$ について作成し、このそれぞれを属性に対応させた。このリレーションの各属性をハッシュ関数によって分割した場合のバケット毎のサイズ（タップル数）、各バケット内にある属性値の種類の数、バケット内での最頻度値とその頻度を表 4.3,4.4 に示す。

4.3 並列ハッシュ結合へのデータ偏りの影響

4.3.1 実験内容

本実験環境上でデータ偏り（属性値偏り）のある場合の並列ハッシュ結合の能力の低下について調べた結果を示す。

ここでは、[KIT83] のバケット集中アルゴリズムを簡略化した並列ハッシュ結合アルゴリズムを用いた。まず、アルゴリズムの流れを簡単に述べる。

1. 各プロセッサは並列にディスク上から一方のリレーションをリードする。リードした各タップルの結合属性値にハッシュ関数 $H(x) = x \ mod \ B$ を適用し、対応するバケットを求める。ここで、B はあらかじめ決められたバケット数とする。

表 4.1: 実験リレーション S_Skew

バケット番号	属性 1 (x1)	属性 2 (x10)	属性 3 (x100)	属性 4 (x200)	属性 5 (x500)
0	31	30	25	18	0
1	32	40	125	218	500
2	32	31	25	19	0
3	32	31	25	19	0
4	32	31	25	19	0
5	31	31	25	19	0
6	31	31	25	19	0
7	31	31	25	19	0
8	31	31	25	19	0
9	31	31	25	19	0
10	31	31	25	19	0
11	31	31	25	19	0
12	31	30	25	19	0
13	31	30	25	19	0
14	31	30	25	18	0
15	31	30	25	18	0

表 4.2: 実験リレーション Z_Skew1

バケット番号	属性 1 (z=0)	属性 2 (z=0.2)	属性 3 (z=0.5)	属性 4 (z=0.8)	属性 5 (z=1)
0	50	72	120	185	237
1	50	63	85	106	118
2	50	58	69	77	79
3	50	55	60	61	59
4	50	52	54	51	47
5	50	51	49	44	39
6	50	49	45	39	34
7	50	48	43	35	30
8	50	47	40	32	26
9	50	46	38	29	24
10	50	45	36	27	21
11	50	44	35	26	20
12	50	43	33	24	18
13	50	43	32	23	17
14	50	42	31	21	16
15	50	42	30	20	15

表 4.3: 実験リレーション Z_Skew2（属性値 1–3）

バケット番号	属性 1(z=0)			属性 2(z=0.25)			属性 3(z=0.5)		
	サイズ	種類	最高頻度（値）	サイズ	種類	最高頻度（値）	サイズ	種類	最高頻度（値）
0	31	13	6 (32)	23	13	5 (48)	25	12	7 (32)
1	31	13	4 (273)	41	15	6 (17)	52	14	23 (1)
2	23	15	2 (18)	32	17	4 (66)	30	12	6 (2)
3	32	14	4 (51)	36	15	5 (51)	31	15	5 (3)
4	39	17	7 (68)	35	14	5 (52)	44	17	8 (4)
5	34	15	5 (101)	31	16	5 (5)	24	12	4 (5)
6	27	16	4 (38)	41	16	4 (22)	28	16	4 (22)
7	28	16	3 (295)	36	16	4 (103)	29	12	9 (7)
8	32	15	5 (8)	28	14	5 (88)	34	15	9 (8)
9	42	18	5 (249)	36	15	5 (25)	27	13	4 (9)
10	39	16	5 (106)	26	15	4 (10)	30	13	4 (42)
11	23	12	4 (139)	20	15	2 (299)	21	11	4 (91)
12	26	14	4 (204)	31	15	5 (92)	44	14	8 (12)
13	38	13	5 (109)	28	15	4 (205)	26	15	4 (141)
14	25	12	4 (286)	33	15	5 (46)	28	14	5 (14)
15	30	16	4 (207)	23	15	3 (159)	27	14	4 (47)

表 4.4: 実験リレーション Z_Skew2（属性値 4–5）

バケット番号	属性 4(z=0.75)			属性 5(z=1)		
	サイズ	種類	最高頻度（値）	サイズ	種類	最高頻度（値）
0	29	12	7 (16)	13	6	4 (32)
1	66	17	37 (1)	101	7	87 (1)
2	37	9	26 (2)	57	9	40 (2)
3	41	13	18 (3)	33	9	23 (3)
4	32	12	11 (4)	32	12	16 (4)
5	27	14	4 (5)	36	10	21 (5)
6	32	13	7 (6)	29	13	9 (6)
7	35	12	8 (7)	34	9	16 (7)
8	38	14	8 (8)	18	9	5 (8)
9	22	11	6 (9)	25	10	9 (9)
10	25	12	9 (10)	21	8	9 (10)
11	21	12	4 (11)	16	8	5 (11)
12	29	12	9 (12)	23	9	13 (12)
13	24	13	4 (29)	20	8	7 (13)
14	21	10	4 (14)	17	8	6 (14)
15	21	12	4 (111)	25	10	9 (15)

2. 静的に決められた各プロセッサとバケットの対応（バケット b を処理するノード p は $p = b \ mod \ B$）に基づき、各プロセッサは受け持ちバケットに属するタップルを他ノードから受けとり、マージを行ってバケットを生成し、それぞれをディスク上のファイルへストアする。

3. 他方のリレーションについても 1. のハッシュ関数を用いて、同様にバケットに分け、ディスク上へストアする。

4. 各プロセッサは受け持ちバケットを一つづつディスクからライトし、ハッシュテーブルを作り、それをプローブし、マッチタップルを生成し、ディスクへライトする。

4.3.2 実験結果

実験では、偏りのない属性値と偏りのある属性値の両方について、プロセッサ数を変化させた時の応答時間を調べた。実験リレーションには 4.2.2 の S_Skew を用いた。ただし、この実験ではバケット数は 8 としている。また、プロセッサ数によらず常に 8 台のディスクへ結果リレーションをライトする。

実験 2-1 では S_Skew リレーションの属性 1（偏りなし）, 属性 3(x100), 属性 4(x200) の 3 種類の結合属性について再分散フェーズの実行時間（上のアルゴリズムの 1, 2 を行う時間）を調べた。実験 2-2 では、結合属性値が均一に分散しているリレーション同士の結合 (偏りなし : 属性 1 ⋈ 属性 1) と、一方のリレーションに偏りがある場合の結合 (偏り 1 : 属性 1 ⋈ 属性 3、偏り 2 : 属性 1 ⋈ 属性 4) の結合フェーズの実行時間（上のアルゴリズムの 4 を行う時間）を調べた。この実験結果を表 4.5, 4.6および図 4.1, 4.2に示す。

この結果より、リレーションにデータ偏りがあると、このアルゴリズムの応答時間は悪化し、偏りの度合が大きいほど、性能が低下することがわかる。また、プロセッサ台数が増えるほど偏りの影響は大きくなった。実験 2-1 では、プロセッサが 8 台の場合、偏り 1 の応答時間は偏りなしに比べて 1.4 倍に増えた。一方、偏り 2 では偏りなしに比べ 1.7 倍 に増えた。実験 2-2 では、プロセッサが 8 台の場合、偏り 1 の応答時間は偏りなしに比べて 1.8 倍に増えた。一方、偏り 2 では偏りなしの場合に比べて 3.0 倍に増えた。

4.4 再分散偏りの制御

4.4.1 実験内容

前章で述べた再分散偏りを扱うアルゴリズムでは、任意のリレーションに対応するために、バケットサイズを調べ、最適なプロセッサへのバケットの割当を決定している。しかし、ここでは単にプロセッサへのバケットの割り当てによりどの程度、偏りの影響を抑えられるのかについてのみ検討を行う。このため、すでに各バケットのサイズが分かっているリレーションの結合フェーズを様々な割当法で実行した場合の応答時間の変化を調べる。従って、ここでは最適なバケット割当を生成するためのコストに関する検討は行っていない。

表 4.5: データ偏りの影響（再分散処理時間）

Node 数	time(sec)		
	偏りなし	偏り 1	偏り 2
1	3.0	3.0	3.0
2	1.7	1.9	2.1
4	1.1	1.3	1.6
8	0.7	1.0	1.2

表 4.6: データ偏りの影響（結合処理時間）

Node 数	time(sec)		
	偏りなし	偏り 1	偏り 2
1	5.3	5.1	5.4
2	2.6	2.8	3.2
4	1.2	1.7	2.3
8	0.6	1.1	1.8

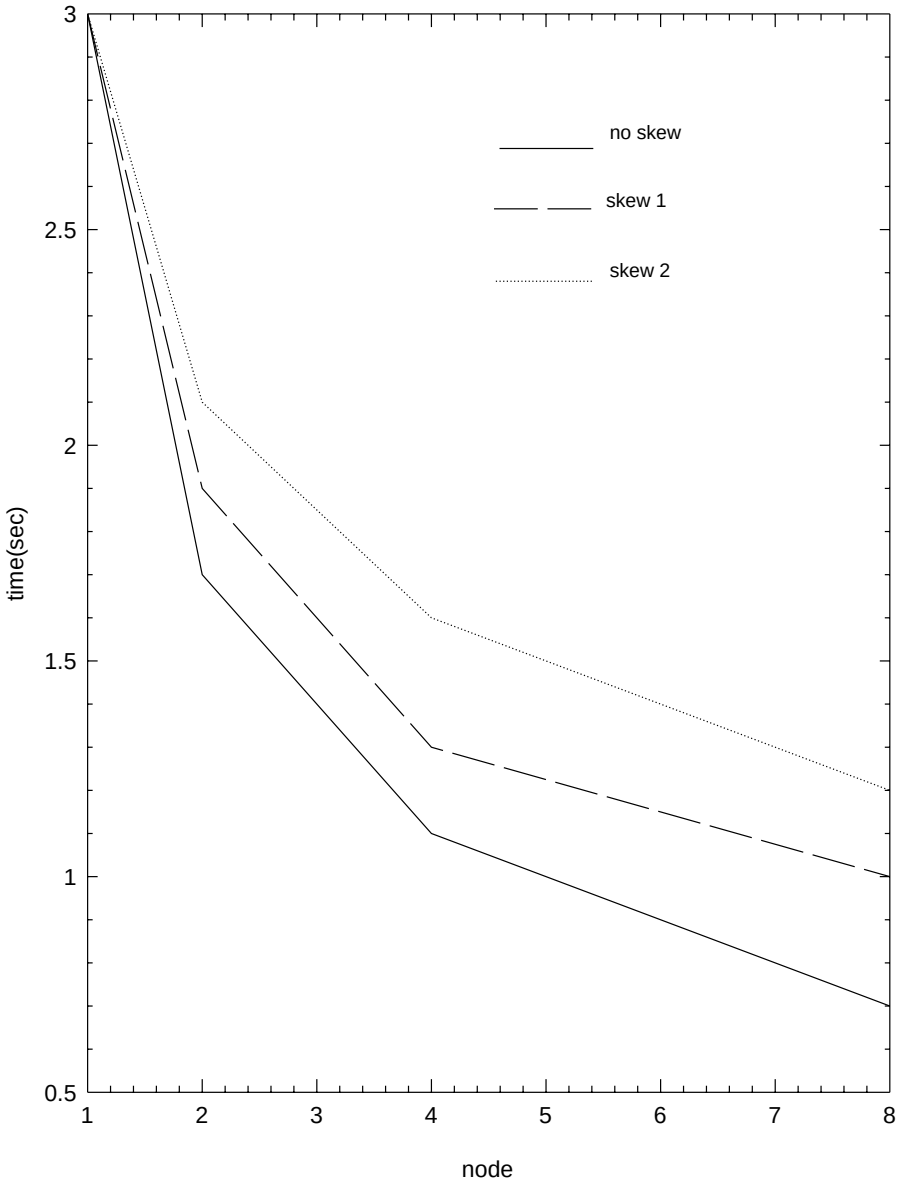

図 4.1: データ偏りの影響（再分散処理時間）

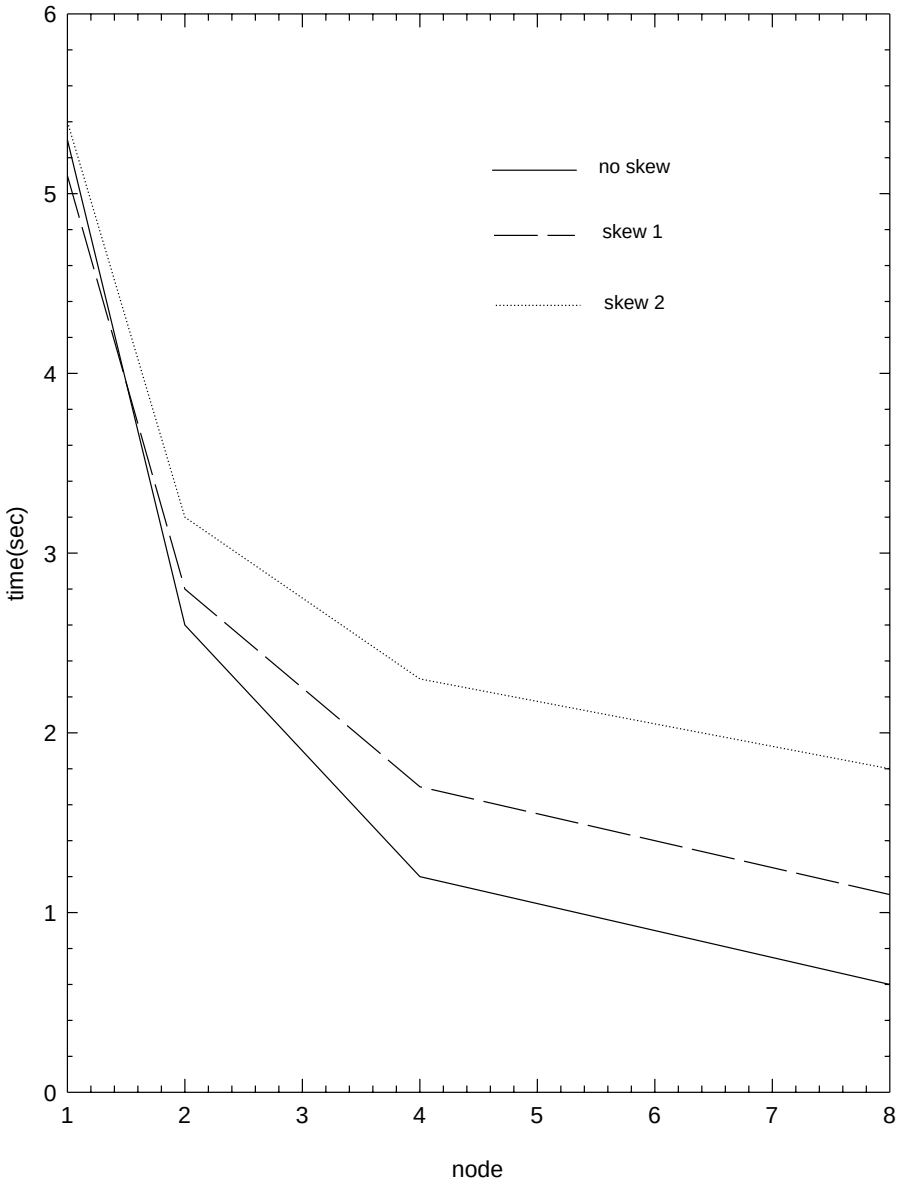

図 4.2: データ偏りの影響（結合処理時間）

次のような 3 種類のプロセッサへのバケット割当について結合フェーズの応答時間の変化を調べた。

1. 単純割り当て　　$p = i \ mod \ N$
2. 最悪割り当て　　$p = \lceil i/(B/N) \rceil$
3. 最適割り当て　　ベストフィット減少法を用いた割り当て

ただしここで、2. の割当が最悪割当となるのは、より小さいバケット番号を持つバケットがより大きなサイズを持つという性質がある Z_Skew1 を実験リレーションとして用いたためである。

4.4.2 実験結果

実験には 4.2.2で述べた偏りリレーション Z_Skew1 を用いた。また、この実験は 4 台のプロセッサを用いて行った。再分散偏り制御アルゴリズムの性能を調べるために、再分散偏りの発生する場合と、結合偏りの発生する場合について偏りの度合を強めながら、各割り当てによる応答時間を調べた。

再分散偏りは、ビルトリレーションの結合属性を属性 1 （偏りなし）に固定して、プローブリレーションの結合属性を属性 1–5 と変化させて発生させた。実験では、この時の各割り当ての応答時間を調べた。各割り当て法で処理プロセッサに割り当てられるタップル数を表 4.7, 4.8, 4.9に示す。また、最適割り当てによるノードへのバケットの割当を表 4.10に示す。

この割当を用いて実行した結合フェーズの応答時間を図 4.3, 表 4.11に示す。表で括弧内の数字は最も処理に時間がかかったプロセッサのプロセッサ番号を示す。- は処理に時間のかかるプロセッサが 1 つに同定できないケースである。

この結果から次のことがいえる。偏りの低い場合 (z $\leq$ 0.2）ではバケットの割り当て方による各プロセッサの処理するタップル数に大きな違いはなく、処理時間にもあまり変化は見られない。最悪割当がやや他の割り当て方式に比べ処理時間がかかっているのは、同一のディスクに各プロセッサが同時にアクセスする事が他の割当法に比べ多いため、その相互干渉によるものと考えられる。

偏りが大きくなるにつれて、単純割り当て法や最悪割り当て法は応答時間が著しく大きくなる一方、最適割り当て法は低い偏りの場合とほとんど変わらない応答時間を保持している。これは、偏りが大きくなるにつれて、各割当方式により各プロセッサへ割り当てられるタップル数が異なってくるためである。各プロセッサへ均等にタップルが分散される場合（ z = 0 ）、処理プロセッサは 4 台なので、各プロセッサは全体の 25% のタップルを受けとる。一方、最も高い偏り（ z = 1 ）の場合、単純割り当てでは全体の 33% のタップルが、最悪割り当てでは全体の 43 % のタップルが単一のプロセッサに集中する。しかし、最適割り当てでは最も多くのタップルが集中するプロセッサでも全体の 27% のタップルが集まるに過ぎない。この結果、各割当方式によって応答時間に変化が現れたと考えられる。

表 4.7: 割り当て 1（単純割り当て）

プロセッサ番号	属性 1	属性 2	属性 3	属性 4	属性 5
0	400	414	447	492	528
1	400	403	404	402	398
2	400	394	381	364	350
3	400	389	368	342	324

表 4.8: 割り当て 2（最悪割り当て）

プロセッサ番号	属性 1	属性 2	属性 3	属性 4	属性 5
0	400	448	534	629	693
1	400	400	391	369	350
2	400	382	349	314	291
3	400	370	326	288	266

表 4.9: 割り当て 3（最適割り当て）

プロセッサ番号	属性 1	属性 2	属性 3	属性 4	属性 5
0	400	406	391	405	437
1	400	399	397	398	384
2	400	398	420	400	397
3	400	397	392	397	382

表 4.10: 最適割り当てによるバケット割り当て（再分散偏り）

バケット番号	属性 1	属性 2	属性 3	属性 4	属性 5
0	P_0	P_0	P_0	P_0	P_0
1	P_1	P_1	P_1	P_1	P_1
2	P_2	P_2	P_2	P_2	P_2
3	P_3	P_3	P_3	P_3	P_3
4	P_0	P_3	P_3	P_3	P_3
5	P_1	P_2	P_2	P_2	P_2
6	P_2	P_1	P_1	P_1	P_3
7	P_3	P_0	P_3	P_3	P_1
8	P_0	P_3	P_2	P_2	P_2
9	P_1	P_2	P_0	P_1	P_3
10	P_2	P_1	P_1	P_3	P_2
11	P_3	P_0	P_3	P_2	P_1
12	P_0	P_3	P_0	P_1	P_3
13	P_1	P_2	P_2	P_3	P_2
14	P_2	P_1	P_1	P_2	P_1
15	P_3	P_0	P_2	P_0	P_2

しかし、最適割り当ての場合においても多少の最分散偏りが発生している事が表 4.9 から分かる。表 4.10 を見ると、偏りの発生しているプロセッサ 0 には 1 つのバケットしか割り当てられていない。しかし、このバケットのサイズが最分散偏りを起こすのに十分大きいためにこのプロセッサで最分散偏りが発生している。このような場合の最分散偏りを取り除く事は単純な最分散偏り制御アルゴリズムではできない。

次に、同様の実験を結合偏りが発生する場合について行った。この実験では、ビルトリレーションを Z_Skew1 の属性 2 に固定し、プローブリレーションの属性を 2–5 と変化させて結合偏りを発生させた。実験では、この時の各割当の応答時間を調べた。単純割り当て、最悪割り当て法によるバケット割り当ては、再分散偏りの実験と同様になる。最適割り当てによるバケット割り当ては表 4.12 に示すようになる。これらの割当を用いて実行した結合フェーズの応答時間を図 4.4, 表 4.13 に示す。

実験結果より、次の事が分かる。結合偏りが発生する場合には、再分散偏りアルゴリズムによって、各プロセッサの受けとるバケットのサイズが均等になるように調整されるかどうかによらず、その影響を強く受ける。図 4.4 を見ると、プローブリレーションの偏りが増加し、結合偏りの度合が大きくなるに従って、どの割り当て方式を用いた場合も著しく応答時間が増加している。この応答時間の増加の割合は、各割り当て方式ともほぼ同じになっている。

たとえば、属性 2 ⋈ 属性 2 に対する、属性 2 ⋈ 属性 5 の応答時間の増加の割合を見てみると、単純割り当てでは 5.58 倍、最悪割り当てでは 5.15 倍、最適割り当てでは 4.93 倍となり、各方式ともほぼ同じ割合で偏りの影響を受けている。一方、属性 2 ⋈ 属性 2 で生成されるマッチタップル数は 1598 タップルであり、そのうちバケット 0 によるマッチタップル数は 578 タップルである。また、属性 2 ⋈ 属性 5 で生成されるマッチタップル数は 6107 タップルである。そのうち、バケット 0 によるマッチタップル数は 4373 タップルである。両者を比較すると、前者に比べて後者は 3.82 倍にマッチタップル数が増加している。またバケット 0 に関しては、その数は 7.98 倍になる。このように高い結合偏りが発生すると、大量のマッチタップルが発生し、このマッチタップルの処理（主にライト処理）によりどの割り当て方式も著しい負荷の増加を被り、処理時間が悪化する。

表 4.11: 各割当法の応答時間 (単位 msec)（再分散偏り）

割り当て	属性 1	属性 2	属性 3	属性 4	属性 5
単純割り当て	1763(-)	1753(P_0)	1872(P_0)	2138(P_0)	2352(P_0)
最悪割り当て	1817(-)	1986(P_0)	2352(P_0)	2938(P_0)	3295(P_0)
最適割り当て	1726(P_2)	1707(P_0)	1817(P_2)	1808(P_2)	1848(P_2)

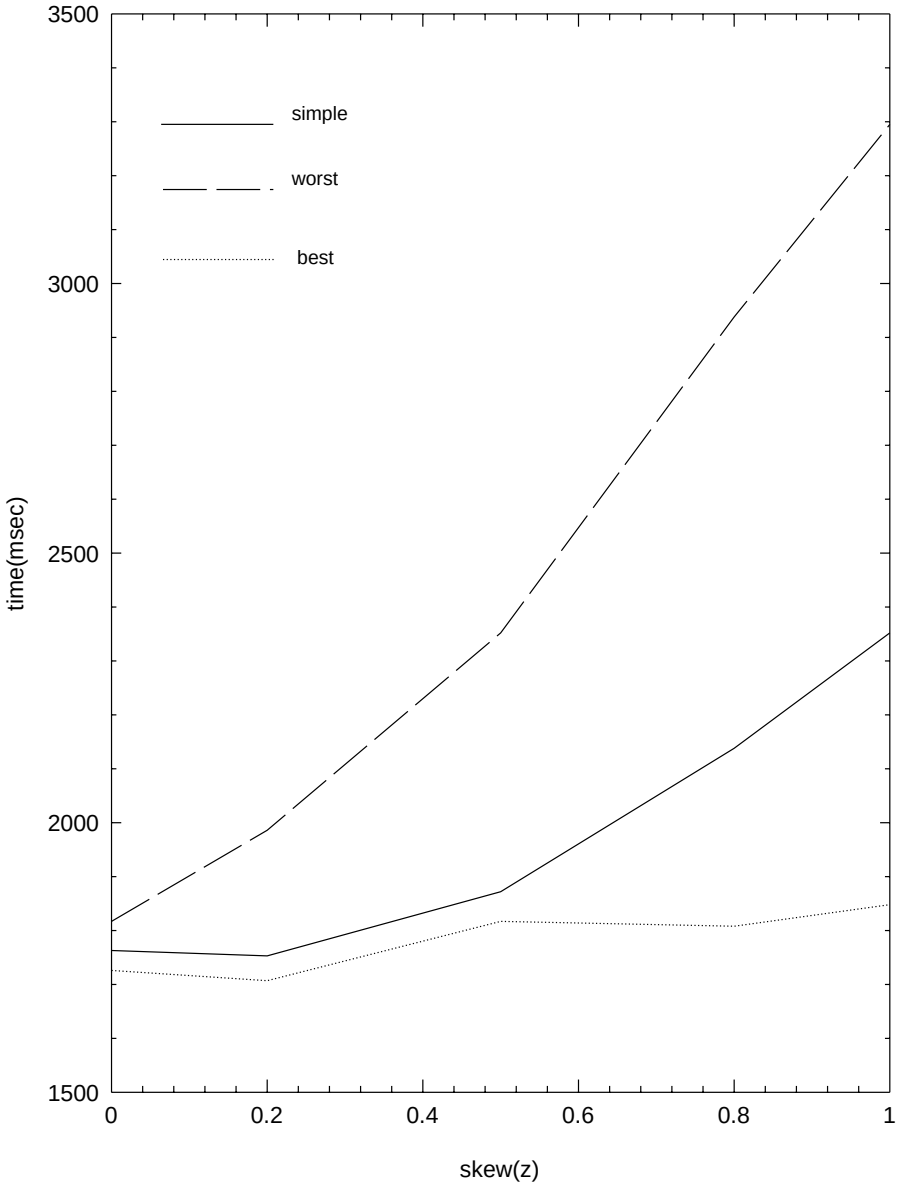

図 4.3: 各割当法の応答時間（再分散偏り）

表 4.12: 最適割り当てによるバケット割り当て（結合偏り）

バケット番号	属性 2	属性 3	属性 4	属性 5
0	P_0	P_0	P_0	P_0
1	P_1	P_1	P_1	P_1
2	P_2	P_2	P_2	P_2
3	P_3	P_3	P_3	P_3
4	P_3	P_3	P_3	P_3
5	P_2	P_2	P_2	P_2
6	P_1	P_1	P_1	P_1
7	P_0	P_0	P_3	P_3
8	P_3	P_3	P_2	P_2
9	P_2	P_2	P_0	P_1
10	P_1	P_1	P_1	P_3
11	P_0	P_0	P_3	P_2
12	P_3	P_3	P_2	P_0
13	P_2	P_2	P_1	P_1
14	P_1	P_1	P_0	P_3
15	P_0	P_0	P_3	P_2

表 4.13: 各割当法の応答時間 (単位 msec)（結合偏り）

割り当て	属性 2	属性 3	属性 4	属性 5
単純割り当て	3353	7167	12380	18725
最悪割り当て	4570	10038	16736	23549
最適割り当て	3356	7198	12017	16555

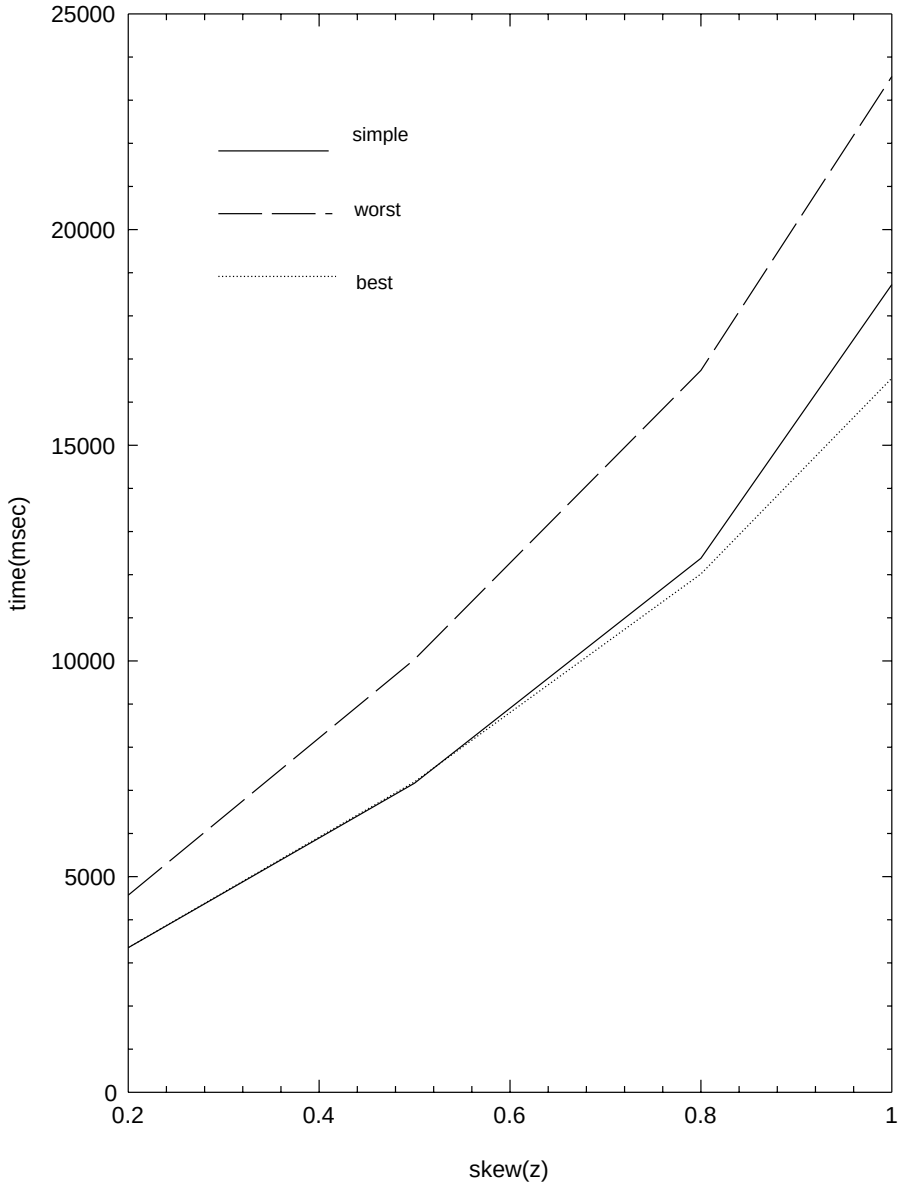

図 4.4: 各割当法の応答時間（結合偏り）

4.5 静的な結合生成偏りの制御

4.5.1 実験内容

ここでは [WOL91] のスケジューリングフェーズで行われる、部分結合処理時間の見積り能力を調べる。ただし、ここでは部分結合処理時間ではなく、単に部分結合処理によって生成されるマッチタップル数を [WOL91] で述べられている方法で計算し、これを実際の生成マッチタップル数と比較する。まず、実験で用いた [WOL91] に基づく生成マッチタップル見積り法を以下で述べる。

この見積りには次の入力情報が必要である。

1. リレーションの区別可能な属性値の数 ($Distinct$)
2. リレーションのサイズ ($Relation_r$)
3. 各バケットのサイズ ($BucketNum$)

1. はリレーションカタログから知る事ができるとする。または、単にリレーションサイズで近似するものとする。2, 3 はハッシュフェーズで入手する。

この見積りでは、まず、これらの入力情報から各バケットが高い偏りを持つ (これを high skew という) か偏りが少ない (これを low skew という) かを予測する。そして、このそれぞれの場合について以下のような仮定に基づいて、バケット内の属性値の濃度 (頻度) $SMALL, HIGH$ を求める。$SMALL$ はバケット内で均一に分散している値の濃度である。$HIGH$ は high skew に存在する高偏り値の濃度である。以下で $BucketNum$ はリレーションが分割されるバケット数である。

- 各バケット内の区別可能な属性値の数は平均 ($Distinct/BucketNum$) と同じ
- low skew バケットは各値が均一に分散している
- high skew バケットは 1 つの高偏り値があり、その他の値は均一に分散している

バケットペア R_i, S_i の $SMALL, HIGH$ からそのバケットの結合による生成マッチタップル数 OUT を見積もる。

この結果見積り処理の流れは以下のようになる。

0. 入力 : $Distinct, BucketNum$
1. 両結合リレーションの各バケットのサイズ (B_{ir}) を求める。またこの結果よりリレーションサイズ ($Relation_r$) を求める。
2. 各リレーション r ($r = 1, 2$) の各バケット i ($0 \leq i < BucketNum$) 毎に以下を行う

2-1 どの偏りクラスに入るかを決める

1. high skew : $Relation_r/BucketNum < B_{ir}$ の場合
2. low skew : 1. 以外の場合

2-2 $SMALL, HIGH$を求める

$ADist = Distinct/BucketNum$ とする

1. low skew : $SMALL = HIGH = B_{ir}/ADist$
2. high skew :

$ADist - 1$ 個の均一に分散している値の濃度

$$SMALL = Relation_r/Distinct$$

1 個の高偏り値の濃度

$$HIGH = BucketNum - (ADist - 1) * SMALL$$

3. 次式を用いて各バケット毎にマッチタップル数 OUT を求める

$$OUT = HIGH_{i1} * HIGH_{i2} + (ADist - 1) * SMALL_{i1} * SMALL_{i2}$$

4.5.2 実験結果

4.2.2で述べたすべての偏りリレーション、すなわち S_Skew, Z_Skew1, Z_Skew2 について、それぞれ属性値 2 と 3 の結合結果をこの結果見積り法でマッチタップル数を見積もった結果と、実際の生成マッチタップル数を表 4.14, 4.15, 4.16 に示す。

属性値 2, 3 は共に偏りを持っているため、この 2 つの属性値の結合では結合偏りが発生する。最分散偏りは前に述べた最分散偏り制御アルゴリズムで扱う事ができるが、結合偏りは扱えないため、結果見積りではこの結合偏りを見積もる事ができるかを検討する必要がある。結合偏りは表 4.14 のバケット 1、表 4.15 のバケット 0, 1, 2 に発生している。これらバケットの見積り値と実際に生成されたマッチタップル数を比較すると、S_Skew のバケット 1 に発生する結合偏りは、実際の生成タップル数の 2.7% のタップルを見積もる事ができないに過ぎず、また Z_Zipf1 のバケット 0, 1, 2 に関しては誤差なしで結合偏りのある場合の生成タップル数を見積れる事が分かる。また、結合偏りの発生しないその他のバケットについても同様に高い精度で生成タップル数を見積もる事ができている。

このように高い精度で見積りが行えたのは、S_Skew や Z_Skew1 がバケット内のある特定の値だけを高い頻度で出現させて偏りを作っているためである。これは、上で述べた結果見積り法の仮定である high skew バケットは 1 つの高偏り値があり、その他の値は均一に分散しているという仮定と適合する。

一方、この様な仮定に反するようにして作成したリレーション Z_Skew2 では、複数の値が偏りを起こす事もあり、また、この見積り法で high skew と判定されないバケットに高い頻度で出現するタップルも存在する。このリレーションを用いた見積り結果では、S_Skew や Z_Skew1 の様な高い精度での見積りが行

えていない事が分かる。例えば、表 4.16 のバケット 0 では実際の生成タップル数の 31% に相当する量のタップルを見積もる事ができない。

4.6 予備実験のまとめ

本章では、並列ハッシュ結合におけるデータ偏りに関する既存の研究の本実験環境における性能について評価した。まず、並列ハッシュ結合におけるデータ偏りの影響の実験として、最分散偏りの度合を変化させた場合の応答時間の変化を調べた。この結果、偏りの度合が強いほど応答時間は悪化し、また、処理プロセッサ台数が増えるほど、偏りの影響は強くなる事が分かった。この事から、作成した実験リレーションがデータ偏りの影響を被る事が確認できた。

次に、最分散偏り制御に関する実験として、3 種類の割り当て方式による、各プロセッサへの割り当てタップル数の違いと、応答時間の変化を調べた。再分散偏りだけが発生する結合においては、バケットサイズに基づいて最適なバケット割当を生成する方法（最適割り当て法）が、他の割当法に比べ最分散偏りの影響を受けにくく、ほぼ均等な数のタップルを各処理プロセッサへ分けられる事が分かった。また、この割当により受けとりタップル数を均衡させた場合、応答時間も偏りの影響を受けにくい事が確認された。一方で、最適割り当て法でも特定のバケットが大きなサイズを持つ場合には、そのバケットの処理を複数のプロセッサへ分配する事はできない。

また、結合偏りが発生する結合においては、各プロセッサへ割り当てられるバケットのサイズを均衡させても、結合選択率の違いによる生成マッチタップル数の違いが生じ、その生成マッチタップルの出力処理によって大幅な性能低下を被る事が分かった。

最後に静的な結合偏り制御に関する実験として、部分結合処理の見積り能力の検討を行った。この実験では部分結合処理によって実際に生成されるマッチタップル数を、[WOL91] で述べられている見積り法で予測した結果と比較した。その結果、見積りの仮定 — あるバケットに属性値偏りがある場合、それは単一の値による偏りである — に従うようなリレーションについては、高い精度で見積りを行える事が分かった。しかし、見積りの仮定に反するリレーションについては、乏しい見積り能力しか持たない事が分かった。

このように予備実験により本実験環境においても偏り制御アルゴリズムの検討が行える事が分かった。また、各偏り制御アルゴリズムの問題点についても検討する事ができた。

表 4.14: 静的な結果見積り 1（スカラー偏り）

バケット番号	見積り値	測定値	バケット番号	見積り値	測定値
0	23	25	8	24	25
1	953	980	9	24	25
2	24	25	10	24	25
3	24	25	11	24	25
4	24	25	12	23	25
5	24	25	13	23	25
6	24	25	14	23	25
7	24	25	15	23	25

表 4.15: 静的な結果見積り 2（zipf-like 偏り (バケットサイズ)）

バケット番号	見積り値	測定値	バケット番号	見積り値	測定値
0	1682	1682	8	37	38
1	553	553	9	34	37
2	229	229	10	32	33
3	115	115	11	30	32
4	64	64	12	28	29
5	49	49	13	27	26
6	44	45	14	26	26
7	41	42	15	25	26

表 4.16: 静的な結果見積り 3（zipf-like 偏り (属性値)）

バケット番号	見積り値	測定値	バケット番号	見積り値	測定値
0	29	42	8	48	53
1	303	159	9	50	53
2	49	62	10	39	44
3	57	66	11	21	16
4	125	106	12	70	89
5	38	46	13	37	42
6	59	74	14	47	38
7	53	50	15	31	39

第 5 章

コーディネータの分散配置による動的な結合偏りの制御

ここでは [HAR95] に基づいた方針で、各プロセッサでの部分結合実行中に、処理の状態を監視し、必要であれば重負荷ノードの負荷を他の軽負荷ノードへ移送する並列ハッシュ結合について述べる。[HAR95] ではコーディネータによる過負荷検出が提案されているが、本研究ではこの機能を各処理ノードに置き、特定のノードに処理および情報が集中しないような方法について考える。また、このコーディネータの分散配置による動的な結合偏りの制御の性能を評価するために実験環境上へアルゴリズムを実装し実験を行った。その結果についても示す。

5.1 コーディネータの分散配置の動機

[HAR95] では、コーディネータが負荷均衡のために各プロセッサの状態やバケット処理の様子を管理している。[HAR95] ではプロセッサの状態は次の 3 つのいずれかである。

- バケット結合中（メモリにハッシュテーブルがある）
- バケット処理終了（メモリが空いている）
- 全担当バケット処理終了（割り当てられた仕事を終えている）

各プロセッサはバケット処理の終了毎にコーディネータに通知を行う。このため、コーディネータは全処理ノードの状態を把握する事ができる。また、各プロセッサの負荷についても一定時間毎に監視し、把握しているため、適切な負荷移送方針を決定する事ができる。例えば、全担当バケット処理を終了したノードが、まだ処理中のノードの仕事の一部を手伝う事を決定できる。また、重負荷ノードの過負荷を移送する際

も、移送先のプロセッサのメモリ状況についての情報を知っているので、これに基づき、3.3 で述べたような 2 種類の移送（結果再分散とタスク処理移送）を選択でき、より効果的な負荷均衡が行える。

コーディネータがシステム全体の状態を監視する事により効果的な負荷均衡方針が立てられる一方、コーディネータには情報と各プロセッサの負荷監視のための処理が集中してしまう。もし、並列計算機の規模が今後も拡大を続けるとすると、集中した情報や処理がコーディネータの処理能力を上回り、コーディネータがボトルネックとなる事が考えられる。そこで、本研究では動的な偏り制御を扱う並列ハッシュ結合の別の負荷分散方法として、過負荷の検出、移送を各ノードが独自に行う、すなわち、コーディネータを分散配置した動的な偏り制御アルゴリズムを検討した。

5.2 コーディネータの分散配置のための方針

コーディネータを分散配置するために、次のような方法で動的な偏り制御を行う。

1. 結果見積りアルゴリズムにより各バケットの生成マッチタップル数を予測する。また、この見積りの際に用いたバケットサイズも記憶しておく。
2. 各プロセッサへ割り当てバケットと割り当てバケットの静的な結果見積り結果、プローブタップル数を報告する
3. 各プロセッサは、担当バケット毎に以下を行う
   (a) 負荷のチェックポイントを求める
   (b) 担当バケットのハッシュテーブルを作る
   (c) 担当バケットのプローブ処理を開始し、チェックポイントに到達するまでプローブ処理とマッチタップルの生成、生成タップルのライト処理を行う。
   (d) チェックポイントに到達したら、これまでにプローブを終えたタップル数と生成されたマッチタップル数から、そのバケット処理の負荷を調べる
   (e) (d) の調査の結果、そのバケット処理が重いタスクであると判断したプロセッサはマッチタップルのライトの一部を他ノードへ移送する。その他のノードは重負荷ノードからのライトと担当バケットの結合処理を並行して行う。

5.3 実現方法

前節で述べたアルゴリズムをどのように実現したかについて以下で説明する。

5.3.1 結果見積り法

各バケット結合により生成されるマッチタップル数の見積り法は 4.5 で既に述べているが、 4.5で示した様に偏りリレーション S_Skew, Z_Skew1 については見積り誤りを起こさない。しかし、前節で述べた動的な結合偏り制御法は見積り誤りがある場合の処理が重要なため、ここではより結合偏りに対する見積り能力の低い結果見積り法を用いる。

この見積り法は古典的な選択演算の選択率の見積り法に基づいた方法で、各バケットのサイズの他に各バケットに属する属性値の種類も調べ、その情報からマッチタップル数を見積もる。

バケット R_i, S_i の結合によって生成されるマッチタップル数の見積りは次の手順で行う。

1. R_i をリードして、タップル数 $|R_i|$、区別可能な属性値の数 $UNIQUE(R_i)$ を求める
2. S_i をリードして、タップル数 $|S_i|$、区別可能な属性値の数 $UNIQUE(S_i)$ を求める
3. R_i, S_i の結合によって生成されるマッチタップル数 $|R_i \bowtie S_i|$ を次式で見積もる。

$$|R_i \bowtie S_i| = \frac{|R_i||S_i|}{UNIQUE(B_i)}$$

ただし

$$UNIQUE(B_i) = \begin{cases} UNIQUE(R_i) & |R_i| \geq |S_i| \\ UNIQUE(S_i) & |R_i| < |S_i| \end{cases}$$

ただし、この見積りは次の仮定に基づいて行われている。

- バケット内で各属性値は均一に分散している
- $|R_i| \geq |S_i|$ ならば $\{R_i\} \supseteq \{S_i\}$

[HAR95] では、再分散フェーズの前にリレーションのスキャンまたはサンプリングを行って結合実行時間を見積もるが、ここでは簡単のために、既にバケットに分けられたリレーションをスキャンして、各バケットのマッチタップル数を見積もる事とする。以下では、結果見積り処理のオーバヘッドについての検討は特に行わない事にする。

5.3.2 過負荷の検出法

[HAR95] では、あるノードの過負荷は、そのノードの処理するバケットの処理時間の見積りが誤る場合に検出される。それは、アルゴリズムがこの見積りに基づいてスケジューリングフェーズで各ノードにバケットを割り当てていくためである。本研究でもこれに従って、見積りと実際の処理の状態の違いから過負荷を検出する事にする。

[HAR95] では、コーディネータが各ノードの過負荷の検出のためにタイマを管理して一定時間の間に処理したプローブタップルの数を調べて、1 プローブタップル当たりの処理時間を求める。これを見積りの結果と比較し、負荷の均衡が必要と見なした場合に負荷の再分散を行う。

一方、ここでは各ノードがプローブを終えたタップル数と、そのプローブタップルにより生成されたマッチタップル数をカウントし、それを見積りの結果と比較して過負荷を検出する。つまり、コーディネータではなく各ノードが、時間ではなくタップル数をトリガーに用いて過負荷を検出するところが [HAR95] と異なる。

具体的には、次のように見積り誤りを検出する。あるバケットのプローブタップル数が $|S_i|$ であり、このバケットの静的な結果見積り値が $est(\ |R_i \bowtie S_i|\ )$ であるとする。バケット結合中に、$|S_i|/3$ 個のプローブタップルを処理し終えた時点（これをチェックポイントとする）での生成マッチタップル数 $Match(S_i/3)$ を調べ、これが次式を満たすならそのバケットの結果見積りは誤っており、そのバケットを処理しているノードは過負荷であると判断する事にした。

$$\frac{est(\ |R_i \bowtie S_i|\ )}{|S_i|} < \frac{Match(S_i/3)}{|S_i|/3}$$

上式の左辺は結果見積りで予測されたこのバケット結合の爆発率（1 プローブタップル当たりの生成マッチタップル数）である。一方、右辺は実行中の計測により得られたこのバケット結合の爆発率である。

この方法による過負荷検出は、プローブバケット内での結合属性値の出現の仕方に大きく依存する。もし、高い結合選択率を持つタップルがリレーションの始めに固まっていると、チェックポイントで測定した爆発率は実際の爆発率に比べ過剰に大きくなってしまう。逆に、高い結合選択率を持つタップルがリレーションの最後に固まっていると、たとえ見積り誤りが発生する場合でもこれを検出する事ができない。

別の過負荷検出法として、結果見積りに基づくバケットのプロセッサへのスケジューリングを行わず、全結合バケットの見積り爆発率から平均の爆発率を計算し、上式の左辺にこの平均の爆発率を用いる方法も考えられる。この場合、各バケットの見積りとは独立に他のノードと比べて負荷が重いノードの負荷を分散するができる。

5.3.3 過負荷の移送

過負荷が検出されると、そのバケット処理の一部を他の処理ノードへ移送して、そのノードに発生した処理の偏りを取り除く。[HAR95] では移送内容として、実際の結合処理とマッチタップルのライト処理の 2 つを挙げているが、ここではマッチタップルのライト処理のみを移送対象とする。また、移送先については現時点では全ノードとしている。しかし、複数の重負荷ノードが発生する事が考えられるため、より選択的に移送先を決める事も検討している。たとえば、過負荷の移送を受ける際に、どのノードからその過負荷が送られて来たかを把握し、自ノードが過負荷になってもそのノードへは過負荷の移送を行わず、自分への移送を中止するように依頼できる機能を持つ事が望ましい。

また、移送する処理の量は次のように決めた。S_i のあるタップル t でハッシュテーブルをプローブした時に生成されるマッチタップル数を $Match(t)$ とする。また、結果見積りによる爆発率を $BrowUp$ 、この結合処理に参加するノード数を N とすると、t の生成マッチタップルのうち移送するタップル数 Mig は次の様になる。

$$
\begin{aligned}
Match(t) &< BrowUp \rightarrow Mig = 0 \\
Match(t) &\geq BrowUp\ and \\
&\quad Match(t) < N \rightarrow Mig = Match(t) \\
&\quad Match(t) \geq N \rightarrow Mig = Match(t)(1 - 1/N)
\end{aligned}
$$

上式で $Match(t)$ が見積り爆発率 $BrowUp$ よりも大きいか調べているのは、プローブタップルの値の出現順序がランダムでない場合に、過剰な過負荷検出をしてしまい不要な負荷移送が行われるのを防ぐためである。

5.4 実験結果

実験環境上で実現したデータ偏りを扱う並列ハッシュ結合アルゴリズムの性能評価として次の様な実験を行った。

1. 過負荷の検出と実行時再見積りに関する実験
2. 通常の並列ハッシュ結合との比較実験
   (a) 各プロセッサのライトタップル数の偏りの解消に関する実験
   (b) 各プロセッサの結合処理時間の偏りの解消に関する実験
3. コーディネータの配置に関する実験

このそれぞれについて以下で述べる。

5.4.1 過負荷の検出と実行時再見積りに関する実験

この実験は 4 台のプロセッサを用いて行った。結合リレーションは 4.2.2 の S_Skew, Z_Skew1 を用いた。表 5.1, 5.2, 5.3 に、過負荷の検出と生成タップル数の実行時再見積りを行った結果を示す。実行時再見積りの結果は本アルゴリズムでは使用しないが、過負荷の検出が妥当であるかの目安にはなるため共に示す。

表 5.1: S_Skew 属性 1 ⋈ 属性 3

バケット	$\|S_i\|$	生成数	見積り	検出	再見積り
0	24	24	24	-	-
1	126	126	149	-	-
2	26	26	26	-	-

表 5.1 はリレーション S_Skew の属性値 1 と 3 を結合した結果である。表 5.2 はリレーション S_Skew の属性値 2 と 3 を結合した結果である。この場合には、バケット 1 の結合により結合偏りが発生する。また、この 2 つの S_Skew リレーションを用いた実験結果にはバケット 0 – 2 についてのみ結果を示す。他のバケットもバケット 0, 2 と同様な結果となる。表 5.3 はリレーション Z_Skew1 の属性値 2 と 3 を結合した結果である。この場合には、バケット 0 – 4 の結合により結合偏りが発生する。

表の $|S_i|$ 欄はプローブタップル数である。生成数欄はそのバケットの結合処理によって生成されたマッチタップル数を表す。見積り欄はそのバケットの生成マッチタップル数の見積りを表す。検出欄は過負荷、すなわち見積り誤りが検出されたかどうかを表す。この欄が + であるバケットは過負荷が検出された。- であるバケットは過負荷が検出されなかった。また、再見積り欄は、過負荷を検出したバケットについて、実行時に得られた情報から、最終的な生成マッチタップル数を見積もった結果である。

この過負荷検出では結果見積りが誤る場合に過負荷を検出するので、表の生成数と見積り欄が大きく異なる場合に過負荷を検出できているかが、問題となる。結果をみると結合生成偏りによって見積りが大きく誤る場合について、そのことを検出できていることが分かる。

過負荷を検出したバケットに対する実行時のマッチタップル数の再見積り値 $|R_i \bowtie S_i|_{reest}$は、次式で求めた。

$$|R_i \bowtie S_i|_{reest} = |S_i| \cdot \frac{Match(S_i/3)}{|S_i|/3}$$

実験結果より再評価値と最終的な結果を比較すると、十分に近い値とはいえないが、結果見積りに比べかなりよい見積り値になっており、結合生成偏りが発生している事を示すのに十分な見積りが行えている事が分かる。

また、ここで 5.3.1 の結果見積り法の見積り能力についても検討する。表 5.1 のバケット 1 には再分散偏りが発生しているが、結合偏りは起こっていない。この場合は、実際の生成タップル数に近い値を見積もることができる。また、偏り値を含まない 1 以外のバケットについても正しい見積りが行える。しかし、表 5.2, 5.3 のように生成偏りが発生すると、その見積りは大きく誤る。また表 5.3 では、まだバケット内での値の不均一さが発生していないバケット（6–15）でも見積り誤りが発生している。これは、この見積り法の見積りのための仮定のうち、$|R_i| \geq |S_i|$ ならば $\{R_i\} \supseteq \{S_i\}$ という仮定が満たされていないためである。

表 5.2: S_Skew 属性 2 ⋈ 属性 3

バケット	$\|S_i\|$	生成数	見積り	検出	再見積り
0	24	24	24	-	-
1	126	1026	191	+	963
2	26	26	26	-	-

表 5.3: Z_Skew1 属性 2 ⋈ 属性 3

バケット	$\|S_i\|$	生成数	見積り	検出	再見積り	バケット	$\|S_i\|$	生成数	見積り	検出	再見積り
0	120	1682	172	+	1836	8	40	38	40	-	-
1	85	553	107	+	637	9	38	37	38	-	-
2	49	229	80	+	237	10	36	33	36	-	-
3	60	115	66	-	-	11	35	32	35	-	-
4	54	64	56	-	-	12	33	29	33	-	-
5	49	49	49	-	-	13	32	26	32	-	-
6	45	45	45	-	-	14	31	26	31	-	-
7	43	42	43	-	-	15	30	26	30	-	-

表 5.4: 各プロセッサのライトタップル数の分散

ノード数	平均	normal			migrate		
		最大	最小	σ	最大	最小	σ
2	678	1155	200	478	835	520	158
4	339	1055	100	414	558	260	127
8	169	1005	50	316	449	129	88

5.4.2 各プロセッサのライトタップル数の偏りの解消に関する実験

この実験ではリレーション S_Skew の属性 2 と 3 の結合（結果は 1355 タップル）について、プロセッサ数を変えながら各プロセッサのライトタップル数の分散について調べた。この結果を表 5.4 に示す。表で normal 欄は 2.2 で述べた偏り制御を行わない通常のハッシュ結合について、migrate 欄は本節で述べたコーディネータの分散配置による動的な結合偏り制御アルゴリズムについての記述である。また、ノード数はその結合処理を行ったプロセッサの数、平均は全マッチタップルを各プロセッサに均等に分けた場合のプロセッサ当たりのライトタップル数、最大（最小）は全ノード内での最大（最小）ライト数、σ は各プロセッサのライトタップル数の標準偏差を表す。

この結合ではバケット 1 で結合偏りが発生する。従って、このバケットの処理を担当するプロセッサにはこの結合偏りによって大量のライト処理が発生する。表 5.4 の最大欄のライト数が結合偏りの影響を受けたプロセッサのライトタップル数を表す。また、S_Skew ではバケット 1 以外では偏りが発生せず、均一な負荷が分散しているので、他のプロセッサは表 5.4 の最小欄のライト数のライト処理を行う。

偏り制御を行うハッシュ結合アルゴリズムはバケット 1 の結合結果のライト処理の一部を他のプロセッサへ移送するので、偏り制御を行わない場合に比べ、最大のライトタップル数が減少し、なおかつ最小のライトタップル数が増加し、両者の差が小さくなる。この結果、偏り制御を行うアルゴリズムは、偏り制御を行わないものに比べ、約 1/3 の標準偏差を持ち、各ディスクにライトされるタップルの数をより均衡させている事が分かる。

5.4.3 各プロセッサの結合処理時間の偏りの解消に関する実験

結合偏りの発生する結合において、動的な偏り制御を行った場合と行わない場合の応答時間の変化をプロセッサ数 1–8 台について調べた。偏りリレーションは S_Skew を用いた。結合条件は属性 2 $\bowtie$ 属性 2 （実験 1）と属性 2 $\bowtie$ 属性 3 （実験 2）と属性 2 $\bowtie$ 属性 4 （実験 3） の 3 つについて調べた。どの場合もバケット 1 に結合偏りが発生する。実験では各処理プロセッサがすべての担当バケットの結合処理および移送されたタップルのライトを終える時間を調べた。

バケットのプロセッサへの割り当ては、本来であれば結果見積りの結果から、各プロセッサの処理時間が均衡するように決定しなければならないが、今回は単に 4.4 の単純割り当てを用いている。従って、結合偏りを持つバケット 1 の処理を行うのはプロセッサ 1 である。他のプロセッサは生成偏りの発生しない結合処理を行う。表5.5, 5.6, 5.7は、過負荷が検出されるプロセッサ 1 (p1) と過負荷を受けとるプロセッサの代表としてプロセッサ 0 (p0) の処理時間を調べた結果である。この処理時間とは、再分散フェーズ、静的な結果見積りが終り、各ノードが割り当てバケットの結合処理を開始し、結果タップルのライト、移送タップルのライトを終えるまでの時間である。

表で N は結合処理を行うプロセッサ数であり、 normal 欄は偏り制御を行わない並列ハッシュ結合による各プロセッサの処理時間である。migrate 欄は動的な偏り制御を行う並列ハッシュ結合による各プロセッサの処理時間である。また、migrate では klic の転送を効率良く行うためのオプション指定を行った場合の処理時間となっている。一方、migrate2 欄は動的な偏り制御を行っているが、転送を効率良く行うためのオプション指定をしない場合のプロセッサの処理時間である。Migrate tuple 欄はプロセッサ 1 が他のプロセッサへ移送したタップル数である。図 5.1, 5.2, 5.3 は normal と migrate でのプロセッサ 0 とプロセッサ 1 の処理時間をグラフで表した物である。

この結果より次の事が分かる。過負荷処理ノードであるプロセッサ 1 の処理時間を normal と migrate で比較すると、migrate の方がより短い処理時間になる。この度合は処理プロセッサ数が大きいほど強くなる。処理プロセッサが 8 台の時、migrate は normal に比べて、実験 1 では 0.84 倍、実験 2 では 0.67 倍、実験 3 では 0.68 倍の処理時間を持つ。一方、移送を受けとるプロセッサ 0 の処理時間は、移送タップルのライト処理により migrate の方が長くかかっている。この時間はプロセッサ 1 による移送タップルの到着を待つため、プロセッサ 1 の処理時間にほぼ等しくなっている。

また、normal と migrate2 を比較すると、migrate2 は偏りの解消による処理時間の減少の度合よりも移送時間による処理時間の増加の度合の方が強く、normal と比べて著しく処理時間が増加している。migrate と migrate2 は同じアルゴリズムにより偏り制御をしているにも関わらず、移送タップルの転送法の違いによりその性能が著しく異なってしまう。このように、動的な偏り制御は過負荷の移送を行うための通信の性能に強く依存するので、動的な偏り制御を行う場合にはこの事に十分な注意を払う必要がある事がわかる。

5.4.4 コーディネータの配置に関する実験

コーディネータの配置に関する実験として、ここでは各ノードで動的な偏り制御のための処理を行った場合と、特定のノードでその処理を行った場合の各プロセッサの処理時間を調べる。ここで、動的な偏り制御のための処理とは、バケットの結合の際に最初に起動され、プローブタップル数から負荷のチェックポイントを決め、それを結合処理部へ伝え、チェックポイントに到達した時に結合処理部からこれまでに生成されたタップル数を受けとり、あらかじめ算出された結果見積りの結果と比較し、移送を行うか、ローカルに

表 5.5: プロセッサ処理時間 : 属性 2 ⋈ 属性 2

N	normal(msec)		migrate(msec)		migrate2(msec)		migrate
	p0	p1	p0	p1	p0	p1	tuple
1	4474	-	4447	-	4503	-	-
2	2009	2513	2446	2449	3248	2663	31
4	995	1486	1365	1378	1463	1660	34
8	535	970	814	819	1234	1124	35

表 5.6: プロセッサ処理時間 : 属性 2 ⋈ 属性 3

N	normal(msec)		migrate(msec)		migrate2(msec)		migrate
	p0	p1	p0	p1	p0	p1	tuple
1	6825	-	6846	-	6818	-	-
2	1731	5174	4792	4796	11971	5176	340
4	891	4618	4022	3269	7195	4092	532
8	496	3961	2669	2643	6720	6787	596

表 5.7: プロセッサ処理時間 : 属性 2 ⋈ 属性 4

N	normal(msec)		migrate(msec)		migrate2(msec)		migrate
	p0	p1	p0	p1	p0	p1	tuple
1	10762	-	10920	-	10832	-	-
2	1545	9370	8080	7293	20864	7732	642
4	843	7659	5926	5592	12953	6862	1014
8	471	7287	6312	4981	12842	12843	1138

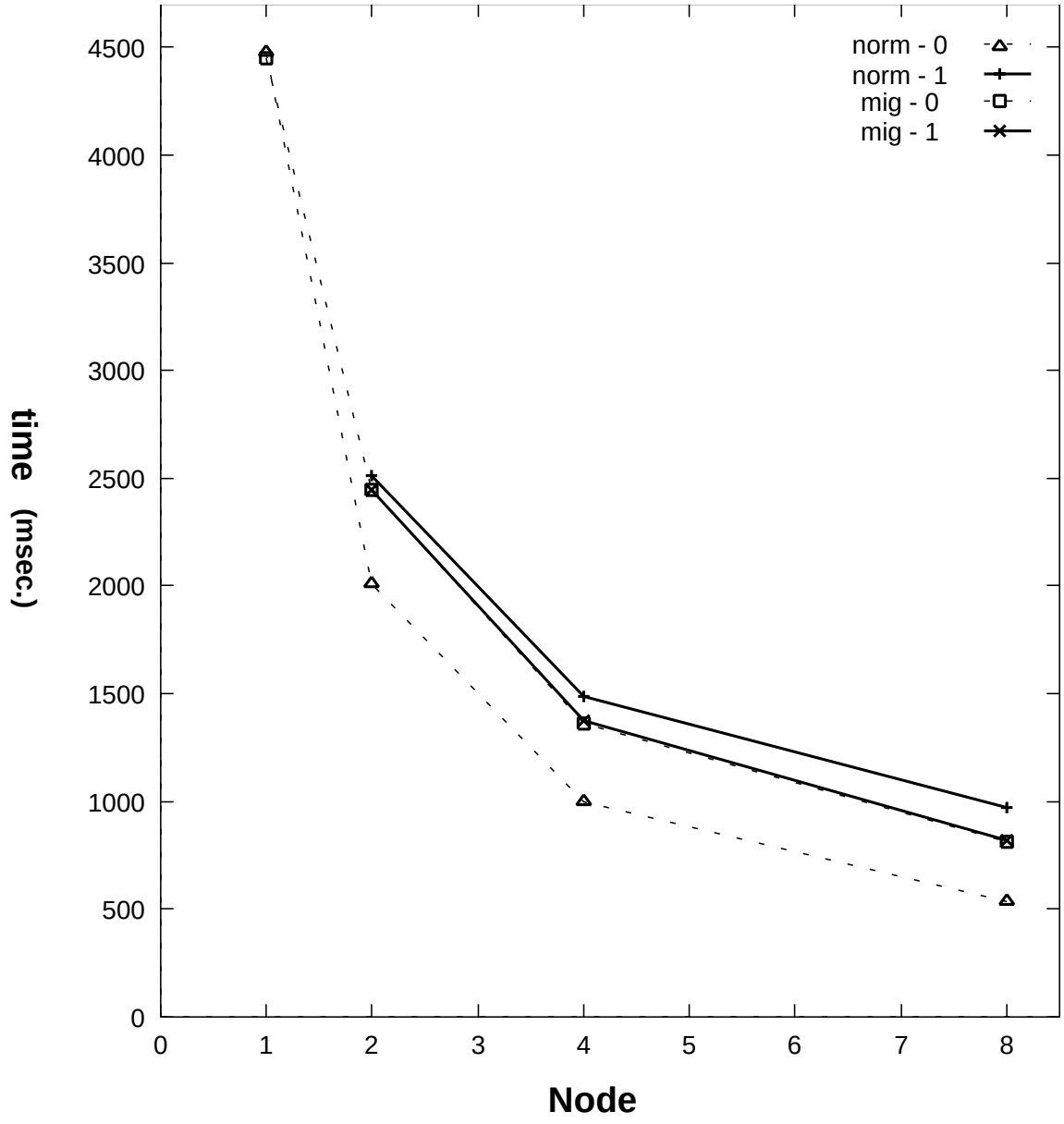

図 5.1: プロセッサ処理時間 : 属性 2 ⋈ 属性 2

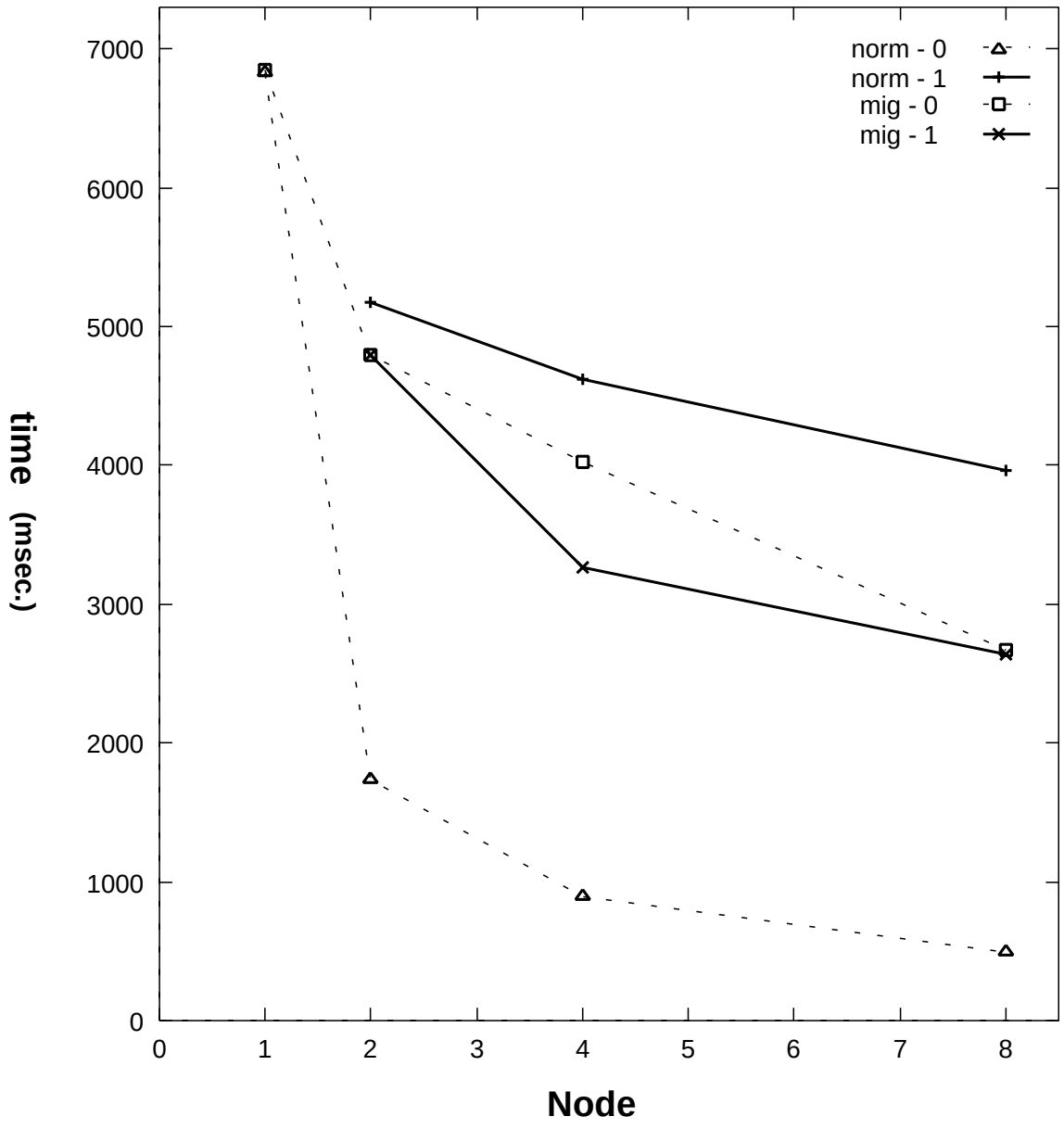

図 5.2: プロセッサ処理時間 : 属性 2 ⋈ 属性 3

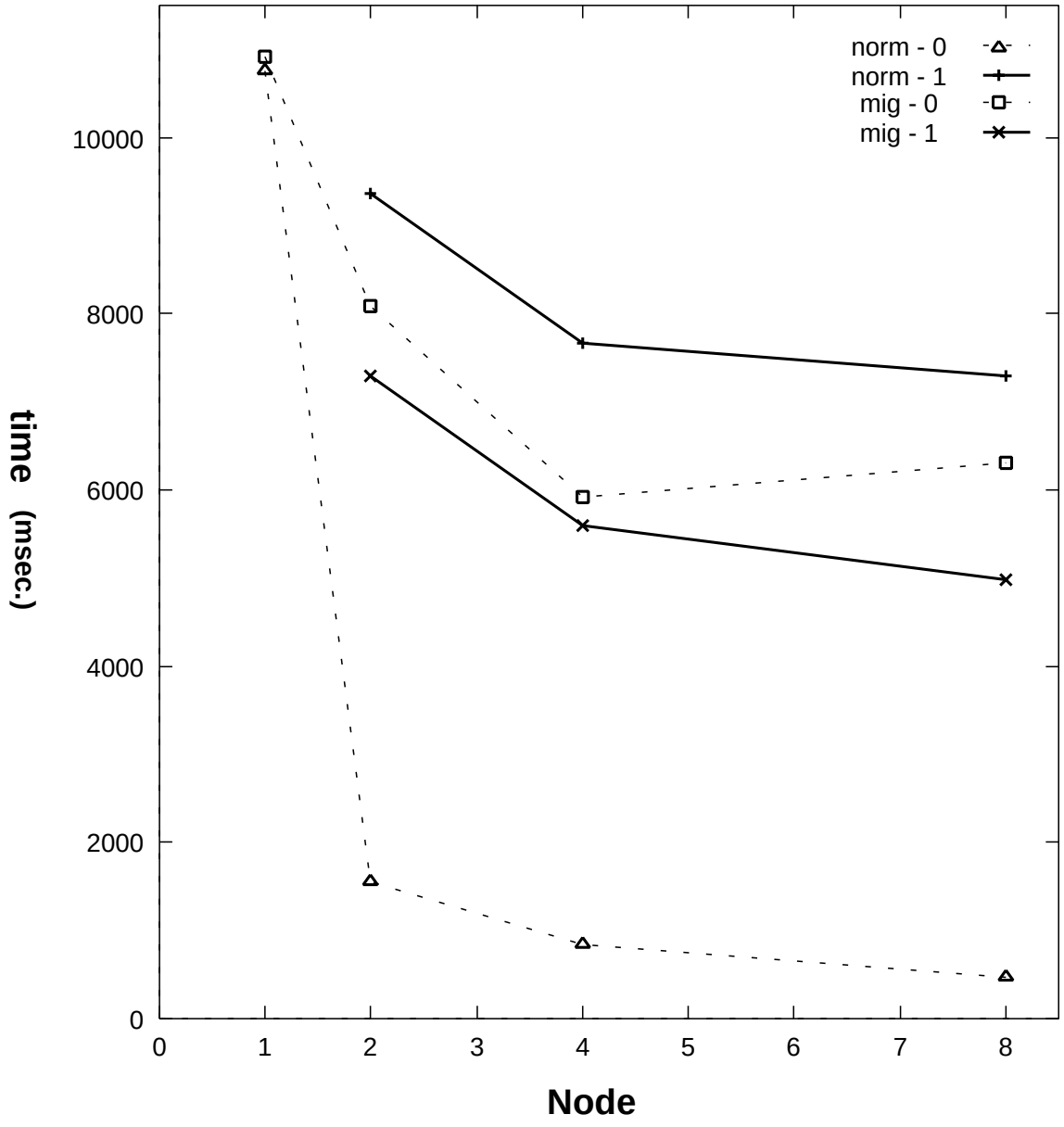

図 5.3: プロセッサ処理時間 : 属性 2 ⋈ 属性 4

処理を続けるかを決定し、それを結合処理部へ伝え、全プローブタップルの処理が終ったという報告を受けとった時に終了する処理である。

この実験は各プロセッサの処理時間に関する実験と同様な方法で行った。ただし、偏りリレーションには Z_Skew1 を用いた。結合条件は属性 2 ⋈ 属性 2 と属性 2 ⋈ 属性 3 の 2 つについて調べた。この結合では、バケット 0, 1, 2 で見積り誤りが検出され、タップルの移送が行われる。偏りの度合はバケット 0 が最も強く、次はバケット 1、その次はバケット 2 となっている。また、バケットのプロセッサへの割り当ては 4.4 の単純割り当てを用いている。このため、複数のプロセッサが移送を行う。また、動的な偏り制御のための処理を行う特定のノードはプロセッサ 0 とした。この実験リレーションでは単純割当を用いた場合、プロセッサ 0 の処理が最も重くなるので、この結果は最悪の場合を想定したものである。

表 5.8, 5.9では、最も重い負荷を持つプロセッサ 0 (p0) と、その次に重い負荷を持つプロセッサ 1 (p1) の処理時間を調べた結果である。表で N, normal, migrate は表 5.5と同様である。coordinator 欄は動的な偏り制御のための処理をプロセッサ 0 で実行した場合の処理時間である。図 5.4, 5.5 はプロセッサ 0 の normal と migrate と coordinator の処理時間をグラフで表した物である。また、バケット 0, 1, 2 の処理において他のプロセッサへ移送されるタップル数を表 5.10, 5.11 に示す。

この結果で migrate と coordinator を比較しても、あまり大きな違いは見られない。通信数は migrate よりも coordinator の方が、他ノードで実行されるバケット処理の情報収集のために 1 バケット当たり 4 メッセージかかる。したがって、この分の通信増加がプロセッサ 0 の処理時間に影響を与える事を予測していた。しかし、今回の実験ではバケット数も処理プロセッサ数もあまり大きくないため大きな影響として処理時間に反映されず、このような結果になったと考える。

また、この実験では複数のプロセッサが負荷の移送を行う。その様な場合のための特別の機能は現時点ではアルゴリズムに含まれていないので、たとえ自ノードが負荷移送を行っている最中でも、他ノードからの負荷の移送を受けてしまう。しかし今回の実験結果では、この様な状況であるにもかかわらず、最も処理に時間がかかるバケット 0 の負荷の多くが移送されるため、このバケットの処理時間がより短くなり normal に比べ migrate の処理時間はより良くなっている。しかし、多くの無駄な負荷移送が行なっているので、より効率の良い処理のためにはこのような移送を取り除く事を検討する必要がある。

表 5.8: プロセッサ処理時間 : 属性 2 ⋈ 属性 2

N	normal(msec)		migrate(msec)		coordinator(msec)	
	p0	p1	p0	p1	p0	p1
1	8614	-	8566	-	8599	-
2	5488	3606	4843	4847	4838	4842
4	3395	2602	2887	2888	2892	2895
8	2667	1679	1950	1953	2203	2203

表 5.9: プロセッサ処理時間 : 属性 2 ⋈ 属性 3

N	normal(msec)		migrate(msec)		coordinator(msec)	
	p0	p1	p0	p1	p0	p1
1	13566	-	15881	-	15097	-
2	10118	4762	8518	8522	9298	9302
4	8136	3226	6393	6394	6349	6352
8	6582	2771	4486	4628	4771	4781

表 5.10: 移送タップル数 : 属性 2 ⋈ 属性 2

N	バケット 0	バケット 1	バケット 2
1	-	-	-
2	213	96	63
4	303	132	75
8	348	150	81

表 5.11: 移送タップル数 : 属性 2 ⋈ 属性 3

N	バケット 0	バケット 1	バケット 2
1	-	-	-
2	575	189	98
4	845	277	124
8	980	321	137

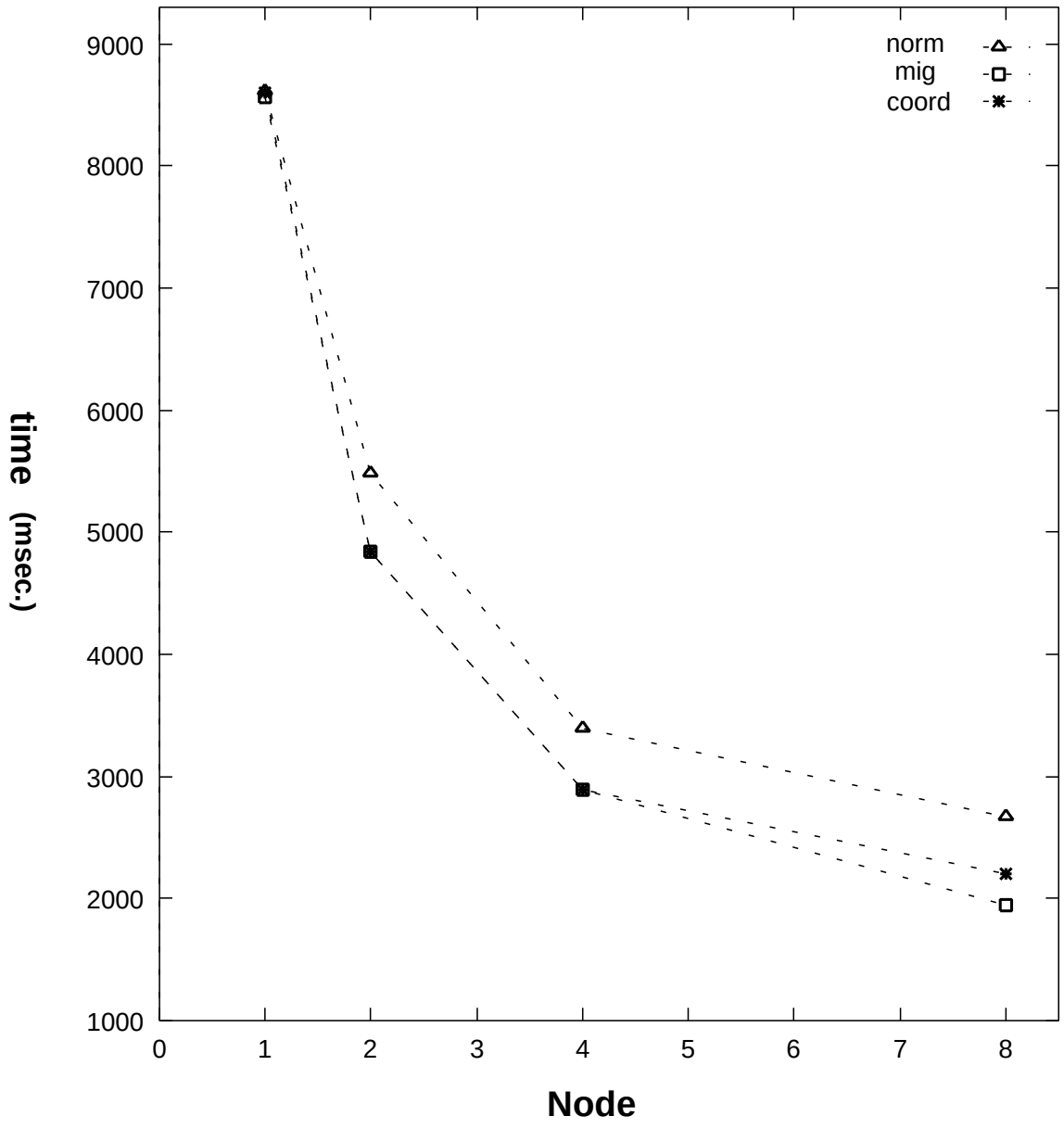

図 5.4: プロセッサ処理時間 : 属性 2 ⋈ 属性 2

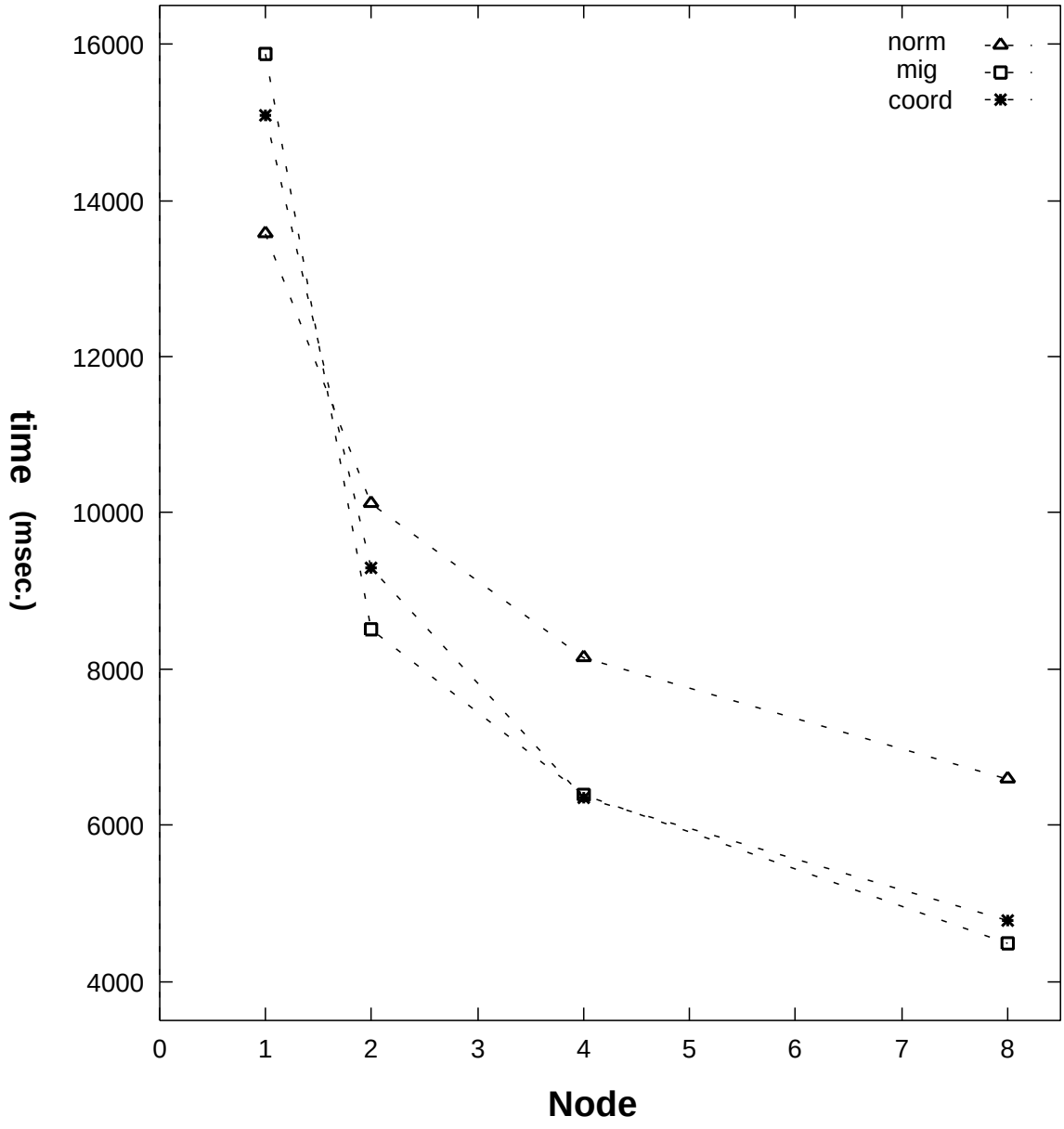

図 5.5: プロセッサ処理時間 : 属性 2 ⋈ 属性 2

第 6 章

まとめと今後の課題

本論文は並列ハッシュ結合におけるデータの偏りについて、その影響の調査、既存のデータ偏り制御アルゴリズムの検討、コーディネータの分散配置による動的な結合偏り制御アルゴリズムの検討を行った結果を述べた。

まず、始めに既存の並列ハッシュ結合アルゴリズムについて説明した。次に並列ハッシュ結合において性能の妨げとなるデータ偏りについて、その分類と並列ハッシュ結合への影響について述べた。次に、既存のデータ偏り制御アルゴリズムについて説明し、これまでにどのようなデータ偏りによる並列ハッシュ結合の性能低下を防ぐ試みが行われてきたかを示した。

次に、実験環境における並列ハッシュ結合とデータ偏り、既存のデータ偏り制御アルゴリズムの動作を確認するため、また、既存のデータ偏り制御アルゴリズムの能力と欠点を検討するために予備実験として次の実験を行った。

- 並列ハッシュ結合へのデータ偏りの影響の調査
- 再分散偏りの制御アルゴリズムに関する実験
- 静的な結合偏り制御アルゴリズムに関する実験

これらの予備実験により本実験環境においても偏り制御アルゴリズムの検討が行える事が分かった。また、各偏り制御アルゴリズムの問題点についても検討する事ができた。

最後にコーディネータの分散配置による動的な結合偏り制御アルゴリズムについて、コーディネータの分散配置を行う動機、その方針、実現方法について説明した。基本的なアルゴリズムは [HAR95] の動的な結合偏り制御アルゴリズムと同様であるが、本研究では移送先の状態の管理を省略し、移送の方式を生成マッチタップルのライト処理という単一の方法で行い、各ノードがローカルに自ノードの過負荷を検出できるようにしている。

この偏り制御アルゴリズムを実験環境上で実装し、その性能を検討するために次の実験を行った。

- 各プロセッサのライトタップル数に関する実験
- 各プロセッサの結合処理時間に関する実験

この結果、各プロセッサのライトタップル数のばらつきは偏り制御を行わない並列ハッシュ結合に比べて 1/3 の標準偏差を持ち、高い偏りを持つバケットの処理を最良の場合 0.67 倍の時間で行える事が分かった。

また、この方式は、移送法決定のための情報を単純化しているので、移送先の状態を考慮した負荷の移送を行う事は困難であり、場合によっては動的な偏り制御のオーバヘッドにより、処理能力が低下する事が懸念されたが、今回の実験ではそのようなオーバヘッドは発生しなかった。

本研究ではコーディネータを分散配置した動的な結合偏り制御アルゴリズムによって、特定のプロセッサに高い偏りがある場合に、その偏りの一部を負荷の移送により他のプロセッサへ分散させ、処理時間を改善できる事を示した。しかし、その改善の度合は十分に大きくなく以下のような改善が必要である。

- 過負荷の検出法の改善
  これにはチェックポイントの検討、平均爆発率を用いた過負荷ノードの検出が含まれる
- 移送量の改善
  過負荷の移送量の決定法を変え、移送する過負荷の量を増減してより良い移送量の決定法を検討する
- 通信方式の改善
  KLIC のジェネリックオブジェクトを用いたノード間転送の検討を行う

また、本研究では多くの簡略を行ったのでこの点も改善し、コーディネータの分散配置による動的な結合偏り制御法の能力を調査する必要がある。その上で、[HAR95] で述べられているような、単一のノードで偏り制御を行う事によって可能となる効率の良い偏り制御を行う場合の偏り制御性能と、コーディネータを分散配置して偏り制御を行う場合の偏り制御能力について比較・検討する必要がある。

謝辞

研究を進めるにあたり、御指導、御助言を頂きました北陸先端科学技術大学院大学情報科学研究科 計算機アーキテクチャ講座の横田 治夫 助教授に感謝申し上げます。

また、研究を行う上で、御指導、御助言を頂きました北陸先端科学技術大学院大学情報科学研究科 計算機アーキテクチャ講座の日比野 靖 教授に御礼申しあげます。

最後になりましたが、博士過程の杉野 栄二氏ならびに宮崎 純氏、並びに横田・日比野研究室の皆様には大変お世話になりました。ここで感謝致します。

参考文献

[DWT90] D.J. DeWitt, S.Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, R. rasmussen "The Gamma Database Machine Project" IEEE Trans. Knowledge and Data Engineering. vol. 2, No. 1, Mar. 1990

[DWT92] D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri "Practical Skew Handling in Parallel Joins" Proc. 18th VLDB Conf. 1992

[HUA95] K.A. Hua, C. Lee, C.M. Hua "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning" IEEE Trans. Knowledge and Data Engineering. vol. 7, No. 6, Dec. 1995

[KIT83] M. Kitsuregawa, H.Tanaka, T. Moto-oka "Application of Hash to Data Base machine and Its Architecture" New Generation Computing, 1 1983

[KIT90] M. Kitsuregawa, Y. Ogawa "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer" Proc. 16th Conf. VLDB 1990

[HAR95] L. Harada, M. Kitsuregawa "Dynamic Join Product Skew Handling for Hash-Joins in Shared-Noting Database Systems" Proc. 4th Inter. Conf. on DASFAA'95, Apr. 1995

[SHA93] A. Shatdal, J.F. Naughton "Using Shared Virtual Memory for Parallel Join Processing" ACM SIGMOD Conf. 1993

[WAL91] C.B. Walton, A.G. Dale, R.M. Jenevein "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins" Proc. 17th Inter. Conf. on VLDB Sep. 1991

[WOL91] J.L. Wolf, D.M. Dias, P.S. Yu, J. Turek "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew" Proc. 7th Int. Conf. DE 1991

[LYN88] C.A. Lynch "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values" Proc. 14th Conf. VLDB 1988