分散メモリ型並列計算機上での並列データキューブ計算における動的負荷分散

武藤 精吾　　喜連川 優

東京大学生産技術研究所

{muto,kitsure}@tkl.iis.u-tokyo.ac.jp

概要

近年新たなアプリケーションとして注目されている OLAP ではユーザが多次元のデータに対しさまざまな操作を高速に行うことができることを目的としている。[GBLP96] では現在最も広く使われている関係データベース上において、多次元データの処理が容易になるようにデータキューブ演算が提案され、現在までデータキューブ演算を効率よく計算するアルゴリズムが開発されてきている [AAD+96, BR99, DANR96, RS97, SAG96, ZDN97]。データキューブ演算は処理負荷が大きいため、これらのアルゴリズムを並列化することによる性能向上が行われている [GC97, FM98]。並列計算ではデータの分布に偏りがある場合にはプロセッサ間において負荷の偏りが生じ、負荷の集中した特定のプロセッサがボトルネックとなってしまう。並列化による性能向上をできるだけ大きくするためにはプロセッサ間の負荷ができるだけ均等になるように負荷分散を行わなければならないが、データキューブ計算では事前に処理負荷を正確に知ることはできないため、静的に負荷分散を行うには限界があるという問題がある。本稿では分散メモリ型並列計算機上における並列データキューブ計算に動的な負荷分散機構を取り入れることにより並列効果をより大きくする手法を提案し、シミュレーションによる実験を行うことによりその効果を示す。

Dynamic Load Balancing for Parallel Datacube Computation on Shared-Nothing Multiprocessors

Seigo Muto　　Masaru Kitsuregawa

Institute of Industrial Science, University of Tokyo

{muto,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

In recent years, OLAP technologies have become one of the important applications in the database industry. In particular, the datacube operation proposed in [GBLP96] receives strong attention among researchers as a fundamental research topic in the OLAP technologies. The datacube operation requires computation of aggregations on all possible combinations of each dimension attribute. Since the computation cost required for the datacube operation is very expensive, parallelization is very important factor for fast datacube computation. However, we cannot obtain sufficient performance gain in the presence of data skew even if the computation is parallelized. In this paper, we present a dynamic load balancing strategy, which enables us to extract the effectiveness of parallizing datacube computation sufficiently. We perform experiments based on simulations and show that our strategy performs well.

1 はじめに

近年新たなアプリケーションとして注目されているOLAPではユーザが多次元のデータに対しさまざまな操作を高速に行うことができることを目的としている。[GBLP96]では現在最も広く使われている関係データベース上において、多次元データの処理が容易になるようにデータキューブ演算が提案され、現在までデータキューブ演算を効率よく計算するアルゴリズムが開発されてきている[AAD+96, BR99, DANR96, RS97, SAG96, ZDN97]。データキューブ演算は処理負荷が大きいため、これらのアルゴリズムを並列化することによる性能向上が行われている[GC97, FM98]。並列計算ではデータの分布に偏りがある場合にはプロセッサ間において負荷の偏りが生じ、負荷の集中した特定のプロセッサがボトルネックとなってしまう。並列化による性能向上をできるだけ大きくするためにはプロセッサ間の負荷ができるだけ均等になるように負荷分散を行わなければならないが、データキューブ計算では事前に処理負荷を正確に知ることはできないため、静的に負荷分散を行うには限界があるという問題がある。本稿では分散メモリ型並列計算機上における並列データキューブ計算に動的な負荷分散機構を取り入れることにより並列効果をより大きくする手法を提案し、シミュレーションによる実験を行うことによりその効果を示す。

2 データキューブ計算の並列化

2.1 データキューブ演算

データキューブ演算では対象となるリレーションの各次元アトリビュートのすべての組合せに対し集約(group-by)演算を適用することにより、対応するメジャーアトリビュートに対し集計処理を行う。例えば、A, B, C の3つの次元アトリビュートからなるリレーションに対しデータキューブ演算を実行した場合、$ABC, AB, AC, BC, A, B, C, \phi$ の組合せに対して集約演算を行うことになる。これらの集約演算により生成されるビューはキューボイドと呼ばれる。本稿では集約関数は[GBLP96]で定義されている分配的関数(count, sum, max, min 等)、幾何的関数(avg 等)を対象とする。これらの関数は部分的に計算した結果を再帰的に利用することができる、すなわち、タプルの集合 $X, Y (X \cap Y = \phi)$ に対し、$f(X,Y) = f(f(X), f(Y))$ が成り立つという性質をもっており、この結果、キューボイド間にもあるキューボイドが他のキューボイドから計算することができるという依存関係が成り立つ。これらの依存関係は[HRU96]により指摘されているように格子を用いて表現することができる(図1)[1]。キューボイド u からキューボイド v を計算するとき、u を親キューボイドと呼ぶ。親キューボイドとなりうるキューボイドが複数存在するときには計算コストが最も小さくなるように親キューボイドを選択することになる。

2.2 並列データキューブ演算

本節では前節で説明したデータキューブ演算の並列化方式について述べる。並列化方式は実装に用いる並列計算機の種類により違いが出てくるが、本研究では分散メモリ型並列計算機を対象とする。すなわち各ノードごとにプロセッサ、メインメモリ、ディスクが構成されており、ノード間はネットワークで結合されているものとする。

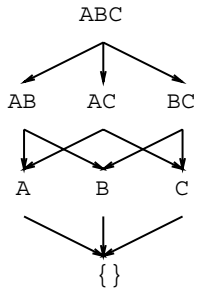

図 1: データキューブ格子

集約演算は同じ group-by アトリビュート値を持った複数のタプルからなるグループに分けられたリレーションに対し、各グループを1つのタプルに集約していく演算であるため、同じグループに属するタプルは同一のプロセッサ上に配置されるようにすれば異なるグループに属するタプル同士はプロセッサ間で独立に計算することができるため並列化は容易である。通常は特定数の group-by アトリビュートを用いてリレーションを複数のパーティションに分割し、それらのパーティションを各プロセッサに割り当てることにより並列処理を行う。

データキューブ演算ではさまざまな次元アトリビュートの組合せに関して集約演算を行わなければならないため、1つのタプルが複数のグループに存在することになり、並列に処理を行うためにはデータの再配置が必要となる。特定の次元アトリビュートによって分割されたリレーションを用いてすべてのキューボイドは計算できないため、計算できなかったキューボイドに対しては異なる次元アトリビュートを用いて再度分割を行ってから計算していくことになる。したがって、データキューブ演算の並列処理では図1の格子は図2に示すように分割に用いられる次元アトリビュートによって複数の部分格子に分けられることになる。この例では最初に原データに対し A を用いて分割を行っているため、この分割では A を含むキューボイドのみが計算可能である。その他のキューボイドを計算するためには今度は前回の分割で計算された ABC に対し B を用いて分割を行い、残りのキューボイドの中から B を含んでいるものを計算していく。最後の分割は BC に対し C を用いて行われ、C の計算結果を用いて最後に残ったキューボイド ϕ[2]の計算を行う。

分割後に各パーティションからキューボイドを計算していくには、すでに提案されているデータキューブ演算のアルゴリズム[AAD+96, BR99, DANR96, RS97, SAG96, ZDN97]を用いることが可能である。本方式には[AAD+96, SAG96]で提案されている「パイプハッシュ」や[RS97]で提案されている「メモリキューブ」を適用することができる。両方式ともパーティションベースで処理を行っていくため、元のアルゴリズムを変えることなくそのまま用いることが可能である。

[1] 次元数の差が2以上であるキューボイド間の依存関係は省略されている。

[2] すべてのタプルを1つのタプルに集約することを意味する。図中では {} と表記。

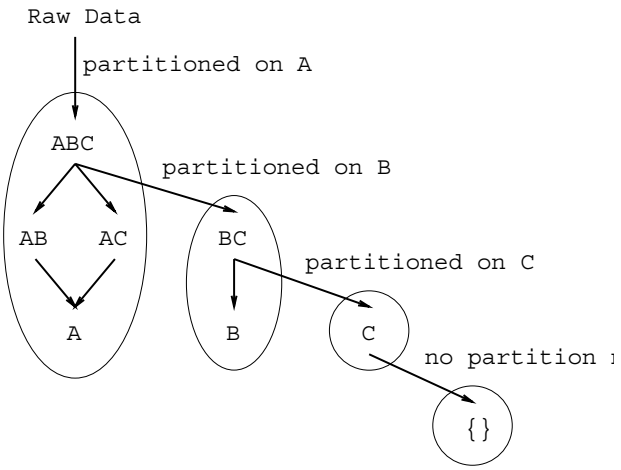

図 2: 並列データキューブ計算の実行木

パイプハッシュはハッシュを用いてキューボイドを計算していく方式で、図2の例を用いると、最初にメインメモリ上に読み込まれたパーティションの各タプルに対しハッシュ関数を適用していき、ハッシュテーブルを用いて該当するタプルが存在するかどうかを検索し、存在する場合にはそのタプルのメジャーアトリビュートに対して集約演算を行い、存在しない場合にはタプルをテーブルに加えていく。このようにして最初に ABC を計算し、次に生成された ABC を用いてこれらの各タプルに対して同様にハッシュを用いて AB、AC を計算していく。

メモリキューブはソートを用いた方式で、最初にメインメモリ上に読み込まれたパーティション内の全タプルを特定の次元アトリビュート順にソートする。この時用いられた次元アトリビュート順と同様のプリフィックスを持つキューボイドはソートされたタプルを最初から順に集約していくことにより計算することができる。例えば、あるパーティション内のタプルが ABC の順でソートされていれば ABC, AB, A のキューボイドを計算することができる。他のキューボイドの関しては異なる次元アトリビュート順を用いて再度ソートすることにより計算していく。

パイプハッシュとメモリキューブではパイプハッシュの方が複数のハッシュテーブルをメインメモリ上に作る必要がことからメインメモリの消費量が高いという特徴があるのに対し、メモリキューブの方がソートを複数回実行しなければいけないことから CPU コストが高いという違いがある。データキューブ演算のような CPU インテンシブな処理では CPU コストが大きく効いてくると考えられるため、 CPU コストの低いパイプハッシュの方がよい性能を示すことが予想される。また、パイプハッシュでは、メモリ消費量の高さからパーティションサイズの上限がメモリキューブよりも低くなってしまい、その結果、パーティションサイズが小さくなることによる部分格子の増加(すなわち、メインメモリ上で同時に計算できるキューボイド数の減少) が起こってしまうが、 CPU コストが高い場合には部分格子の増加による通信コストやディスク I/O コストの増加が全体の性能に与える影響は少ないと考えられ、また負荷分散の観点からもパーティションサイズは小さい方が望ましいため、メモリ消費量の高さは大きな問題にはならないと考えられる。このような理由から本方式ではパイプハッシュを採用している。

3 負荷分散方式

並列処理による性能向上をできるだけ大きくするにはプロセッサ間の処理負荷ができるだけ均等になるように各プロセッサに処理を割り当てなければならない。全節で述べられた並列処理方式に対して、どのようにプロセッサ間で処理負荷の均等化を行うかについて本節で検討する。

負荷分散の単位 負荷分散を行う場合にどのような単位で行うかという問題があるが、本方式ではパーティション単位で処理を進めていくため負荷分散もパーティション単位で行っていく方式を採っている。パーティション以外にはタプル単位での負荷分散などが考えられるが、パーティション単位では各パーティション同士はまったく独立に処理できるため、パーティションは移動先のプロセッサ内で処理を完結させることができるが、タプル単位でデータの移動を行うと同一パーティションに属するタプルが複数のプロセッサに分散されてしまうため、その制御はたいへん複雑になってしまうという問題があり、また、細かい単位で負荷分散を行うことによるオーバヘッドが大きいと考えられるため、本方式にはパーティション単位が適していると考えられる。

単位当たりの負荷の大きさ 前節でも述べたようにデータキューブ演算のコストは CPU コストが大きいと考えられるため、パイプハッシュのコストは CPU バウンドであるものとする。パイプハッシュの CPU コストは各キューボイドを計算するコストの合計ということになるが、ハッシュを用いた集約演算のコストは処理するタプル数に比例するためパイプハッシュを用いた場合の各キューボイドの計算コストは親キューボイドのサイズ (タプル数) によって決定される。したがって、各パーティション当たりの負荷はそのパーティションのサイズとそのパーティションから生成されるキューボイドの内、親キューボイドとして機能するもののサイズの合計で決まってくることになる。

集中 or 分散 負荷分散方式には一般に大きく分けて集中方式と分散方式とが考えられる。集中方式では特定のプロセッサが全体の負荷分散を管理するのに対し、分散方式では各プロセッサ同士で自律的に負荷分散を行っていく。集中方式では全体の負荷状況を考慮して負荷分散を行うことができるため、効率的な負荷分散を行うことができるという利点があるが、情報を管理するプロセッサがボトルネックとなるため、規模の大きな並列環境には向かないと考えられる。分散方式ではそのようなボトルネックは生じないが、負荷分散効率は集中方式よりも低くなると考えられる。また、デッドロックなどが起こる可能性も考慮しなければならない。また、この 2 方式を融合させた手法も考えられる。どの方式が適しているかは実装環境などにも依存すると考えられるが、今回のシミュレーションでは集中方式が用いられていることを仮定して実験を行っている。

静的 or 動的 また別の選択肢として負荷分散を静的に行うか動的に行うかと選択も考えられる。静的な負荷分散方式では事前に負荷ができるだけ均等になるようにプロセッサへの負荷の割り当てを行う。それに対し、動的な方式では処理が始まった後もその進行状況により動的に負荷の再割り当てを行う。事前に負荷が正確に予測できる場合には静的な方式を用いればよいが、データキューブ計算の場合には生成されるキューボイドのサイズは事前には知ることができないため、なんらかの予測手法(例えば [SDNR96]) を用いない限り負荷分散の効果には限界がある。ある程度のオーバヘッドをともなって予測手法を用いた効果よりも、動的な手法を用いて負荷分散を行った方がその効果は大きいと考えられ、したがって、本方式では動的なアプローチによる負荷分散方式を採っている。

3.1 静的負荷分散

本節では本方式で用いた静的負荷分散アルゴリズムについて説明する。本方式ではパーティション単位で各プロセッサに処理を割り当てるが、パーティション間には依存関係がないため、パーティションの割り当て問題は次のように定式化することができる。1 台のプロセッサに割り当てられているパーティションを P とし、それらの集合全体を PS とする。パーティション $p \in P$ の処理コストを $Cost(p)$ とすると、パーティションの割り当ては式 (1) で表される C_{max} が最小となるように行われた場合に最適となる。

$$C_{max} = \max_{P \in PS} \{ \sum_{p \in P} Cost(p) \} \quad (1)$$

最適な割り当てを求める問題は NP 問題である。この問題にはさまざまなヒューリスティクスが提案されているが、ヒューリスティクスを用いることによる最適解との誤差の広がりはパーティションサイズを十分小さくすることにより避けることができるため、用いるヒューリスティクスの種類による違いは少ないと考えられる。本方式ではすでに割り当てられているパーティションの合計サイズが最小であるプロセッサに、まだ割り当てられていないパーティションの中から最大サイズのものを割り当てていくことによりすべてのパーティションの割り当てを行っている。また、パーティションのコストには予測手法は用いず、各パーティションのサイズをコストとして割り当てを行う方式を用いた。

3.2 動的負荷分散

静的な負荷分散方式のみでもデータの分布や予測手法の精度によってはある程度の負荷分散を達成することができるが、各パーティションの処理コストの予測は高い精度を要求するほどそのオーバヘッドは大きくなってしまう。本方式で用いている静的負荷分散方式では単純にパーティションサイズをコストとして用いているため、オーバヘッドは存在しないが、この方式では最初に生成されるキューボイドのコストのみを考慮していることになるため、それより後に生成されるキューボイドのコストによっては静的負荷分散を用いても負荷の偏りが生じる可能性がある。図 3を例に用いてこのようなケースを説明する。この図ではリレーションを 2 つのパーティションに分割し、各プロセッサ上でデータキューブ演算を行っている。パーティションは静的負荷分散を用いてパーティションサイズが等しくなるように割り当てが行われている。このため、最初のキューボイドである ABC の計算に関しては各プロセッサで同じコストとなっている。しかしながら、その後の AB, AC, A の計算に関してはプロセッサ間でコストに大きな偏りができていることがわかる。動的負荷分散方式ではこのような負荷の偏りが生じてもプロセッサ間における負荷の均等化を図ることができるように、各プロセッサにおける処理の進行状況の情報をもとにして動的に負荷の再分配を行う。これにより分布に大きな偏りのあるデータを処理する場合にも高い並列効果を得ることが期待できる。

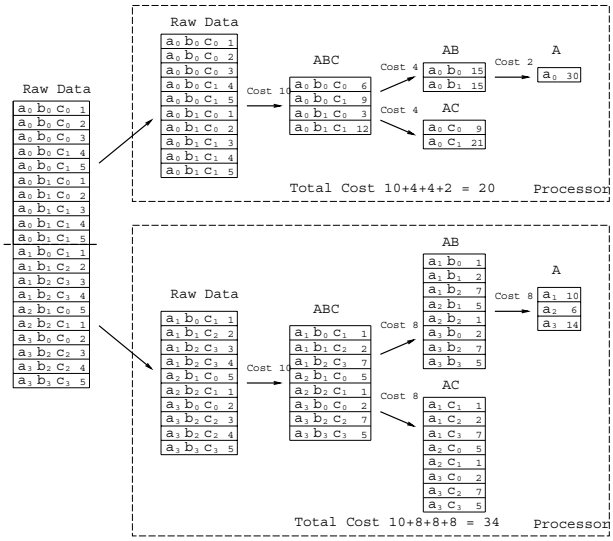

図 3: 並列データキューブ計算の例

前述のように今回のシミュレーションでは集中方式を仮定しており、1 台のプロセッサが全プロセッサの処理状況を管理しており、全体の負荷分散の制御を行うことを仮定している。制御プロセッサは割り当てられたすべてのパーティションの処理が終了したプロセッサに対しては、未処理のパーティションの合計コストが最大である他のプロセッサから最小コストのパーティションの再割り当てを行うことにより負荷分散を行っていく。ここでもやはりコストの予測は困難であるため、静的負荷分散方式と同様パーティションサイズをコストとして用いている。

4 実験

本稿で提案した負荷分散方式の効果を示すため、シミュレーションにもとづく実験を行った。本節においてその結果を示す。

4.1 シミュレーションの設定

原データを分割しパーティションを生成するフェーズはディスク I/O バウンド、データキューブの計算は CPU バウンドであるとし、また、 CPU、ディスク I/O、通信は完全にオーバラップすると仮定している。集約演算の処理コストに関してはタプル長とタプル数により決定されるコストモデルを用いている。各タプルは次元数分の次元アトリ

パラメータ	値
次元数	10
タプル数	10^8
ドメインサイズ	100
ディスク転送速度 (リード)	6.45(MBytes/sec)
ディスク転送速度 (ライト)	3.5(MBytes/sec)
集約演算の処理時間	(5.82+0.53d)t(msec) d: 次元数、 t: タプル数
アトリビュートサイズ	4(Byte)
パーティションサイズ	64(KByte)

表 1: パラメータ値

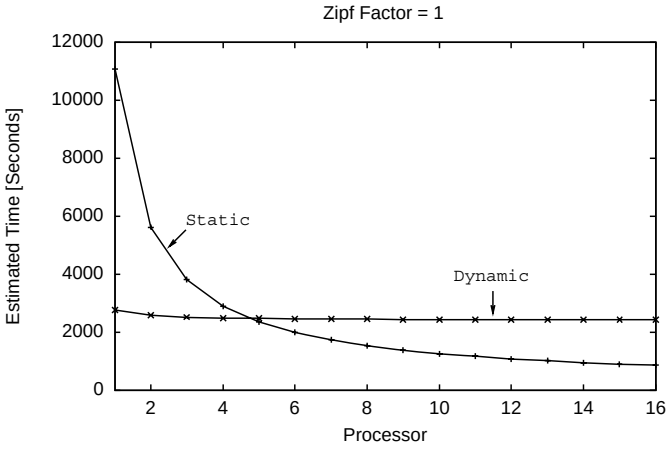

図 4: 各プロセッサにおける実行時間

ビュートと 1 つのメジャーアトリビュートからなっているものとする。表 1にシミュレーションに用いたパラメータの値を示す。ディスクの転送速度、集約演算の処理時間は IBM SP2 上にて計測した値にもとづいている。プロセッサ間における負荷の分布には Zipf 分布を用いた。

4.2 シミュレーション結果

図 4に Zipf 係数が 1 である場合においてシミュレーションを行った時の各プロセッサの実行時間を示す。静的な負荷分散方式ではパーティションサイズのみを考慮しているため実際の処理負荷の偏りがそのまま各プロセッサの実行時間の分布に表れているのに対し、動的な負荷分散方式では処理負荷に動的に対応し負荷分散を行っているためどのプロセッサでもほぼ等しい実行時間となっている。

次のシミュレーションでは 16 台のプロセッサ上で Zipf 係数の値を変化させていった時の効果について調べた。図 5にその結果を示す。性能比は次の式にもとづいて算出されている。

$$\text{性能比} = \frac{16\text{プロセッサでの実行時間} \times 16}{1\text{プロセッサでの実行時間}}$$

性能比はプロセッサの使用効率が高いほど 1 に近付き、低くなるほど 1 よりも大きな値を示すことになる。この結果から静的な負荷分散方式では最も低速なプロセッサがボトルネックとなるため Zipf 係数の増加にともない性能比が増

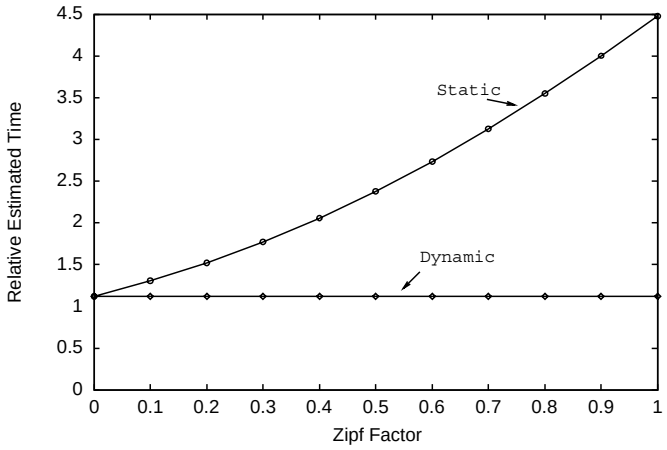

図 5: 偏りの変化に対する性能比

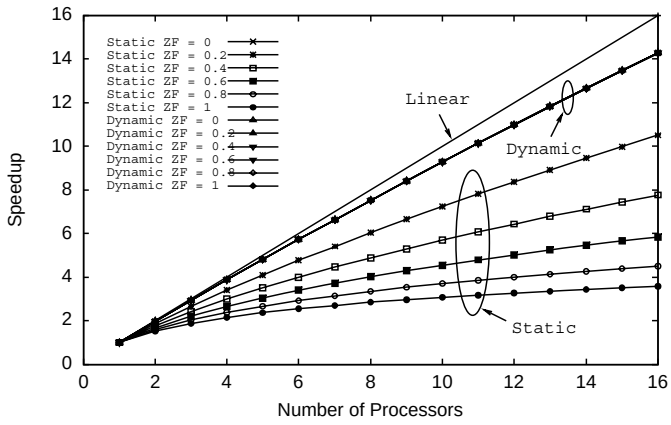

図 6: 台数効果

大しているのに対し、動的な負荷分散方式では Zipf 係数が増加しても効果的に負荷分散されており、全プロセッサが効率的に使用されているのが分かる。

最後に負荷分散方式による台数効果への影響を見るためプロセッサ数を変化させた時の 1 台のプロセッサでの実行時間に対する速度向上率を調べるシミュレーションを行った。図 6にその結果を示す。この結果から静的な負荷分散方式では負荷の偏りが大きくなるほど台数効果の低下が著しいのに対し、動的な負荷分散方式では負荷の偏りに関係なくほぼ理想的な台数効果が得られているのが分かる。

5 関連研究

並列処理における負荷分散手法は数多く提案されているが、データベースの研究において比較的本研究に近いものとして結合演算の並列処理における負荷分散が挙げられる [DNSS92, HL91, KO90]。集約演算に関しては [SN95] で並列処理方式について検討されている。 [SN95] は負荷分散に重点をおいた論文ではないが、提案されている処理方式は偏りのある負荷に対しても従来の手法よりも高性能であることを示している。

データキューブ演算の並列化に関しては [GC97] や [FM98] において取り上げられている。本方式が ROLAP であるのに対して [GC97] では MOLAP のアプローチからデータ

キューブ計算の並列処理を扱っている。この論文も負荷分散に関する論文ではないが、最初にデータを分割する際にサンプリングを行い、各プロセッサに均等にデータが分配されるようにしており、本稿で述べた静的負荷分散に該当する手法を採り入れている。[FM98] ではROLAP のアプローチからデータキューブ計算の並列化を行っているが、複数の集約演算を実行する場合における処理方式に重点をおいており、負荷分散に関しては記述されていない。

6 まとめ

本稿では分散メモリ型並列計算機上におけるデータキューブ計算の並列処理方式に関してまず述べ、その後並列処理における負荷の偏りによる性能低下を防ぐための負荷分散方式について検討を行った。静的に負荷を均等化するだけでなく、動的に負荷を再分配していくことにより負荷を予測する必要なく均等化する方式に関して提案を行い、シミュレーションにもとづく実験によりその効果を示した。現在 IBM SP2 上において本方式の実装が行われており、今後は実装による実験にもとづきより現実的な性能比較を行っていく予定である。

参考文献

[AAD+96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan and S. Sarawagi, "On the Computation of Multidimentional Aggregates", In *Proceedings of the International Conference on Very Large Databases*, pages 506–521, 1996.

[BR99] K. S. Beyer and R. Ramakrishnan, "Bottom-Up Computation of Sparse and Iceberg CUBEs", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 359–370, 1999.

[DANR96] P. M. Deshpande, S. Agarwal, J. F. Naughton and R. Ramakrishnan, "Computation of Multidimensional Aggregates", Technical Report 1314, University of Wisconsin, Madison, 1996.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins", In *Proceedings of the International Conference on Very Large Databases*, pages 27–40, 1992.

[FM98] T. Fukuda and H. Matsuzawa, "Parallel Processing of Multiple Aggregate Queries on Shared-Nothing Multiprocessors", In *Proceedings of International Conference on Extending Database Technology*, pages 278–292, 1998.

[GBLP96] J. Gray, A. Bosworth, A. Layman and H. Pirahesh, "A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", In *Proceedings of the IEEE International Conference on Data Engineering*, pages 152–159, 1996.

[GC97] S. Goil and A. Choudhary, "High Performance OLAP and Data Mining on Parallel Computers", *Journal of Data Mining and Knowledge Discovery*, 1(4):391–417, 1997.

[HL91] K. A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", In *Proceedings of the International Conference on Very Large Databases*, pages 525–535, 1991.

[HRU96] V. Harinarayan, A. Rajaraman and J. D. Ullman, "Implementing Data Cubes Efficiently", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205-216, 1996.

[KO90] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", In *Proceedings of the International Conference on Very Large Databases*, pages 210–221, 1990.

[RS97] K. A. Ross and D. Srivastava, "Fast Computation of Sparse Datacubes", In *Proceedings of the International Conference on Very Large Databases*, pages 116–125, 1997.

[SAG96] S. Sarawagi, R. Agrawal and A. Gupta, "On Computing the Data Cube", Research Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[SN95] A. Shatdal and J. F. Naughton, "Adaptive Parallel Aggregation Algorithms", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 104–114, 1995.

[SDNR96] A. Shukla, P. Deshpande, J. F. Naughton and K. Ramasamy, "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies", In *Proceedings of the International Conference on Very Large Databases*, pages 522–531, 1996.

[WDJ91] C. B. Walton, A. G. Dale and R. M. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", In *Proceedings of the International Conference on Very Large Databases*, pages 537–548, 1991.

[ZDN97] Y. Zhao, P. M. Deshpande and J. F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159–170, 1997.