平成 12 年度

学士学位論文

題目

多重集約演算のための
並列パイプライン処理方式

Parallel Pipelining Method
for Multiple Aggregate Queries

1010445　　前田庸亮

指導教員　　岩田誠

2000 年 2 月 5 日

高知工科大学 情報システム工学科

要 旨

題目

多重集約演算のための
並列パイプライン処理方式

前田庸亮

複数の集約演算を並列に処理するアルゴリズムとして、m2P、mRep、mBC などのアルゴリズムが提案されている。これらのアルゴリズムはすべてのノードが直接結ばれているモデルを想定すると、より大規模な並列計算機ではこの仮定が成り立たない。そこで本稿では、より拡張性に優れたリング構造向きのノード間通信アルゴリズムを提案し、並列パイプライン化を適用して通信時間の隠蔽による高速化を図る。提案するアルゴリズムは各ノードが、演算とノード間の通信を交互に繰り返し、集約処理を進める。通信は隣接した特定のノード間で行われ、衝突が起こることは無い。

既存のアルゴリズム評価に用いられているコストモデルに最小限の変更を加えて、mBC、mRep、並列パイプライン化適用前及び、適用後の本アルゴリズムを評価したところ、パイプライン化は全ての場合に効果があり、特にノード数が少ない場合に有力であった。次に、通信速度を変化させた場合、ノード間の通信速度が比較的遅い場合に改善の効果が大きかった。さらに、目標としていたノード数が大きい場合はいずれのアルゴリズムよりも高速であるという良好な見通しを得た。

キーワード　データマイニング, 集約演算，並列パイプライン，ノード間通信

Abstract

Parallel Pipelining Method for Multiple Aggregate Queries

YOUSUKE MAEDA

Existing parallel algorithms to do multiple aggregations–for instance m2P,mRep,and mBC–are based on a model in which every node is directly connected with each other. But,the model is inadequate for up-to-date super parallel computers. A novel inter-node communication algorithm adopting the ring structure and its parallel pipelined improving trial is described.In this paper using that every node processes aggregations repeating a set of computations and communications.Inter-node communications occur between the next node respectively,so that contentions would not be happen.The cost model used in evaluations of existing algorithms is modified to support the ring stracture.The modified cost model shows.

(1)Pipelining is always effective, especially in case of the number of nodes is small.

(2)A significant improvements of performance is obtained at relatively low band width in inter-node communication.

(3)As the number of nodes maximized, the new algorithm exceeds the other.

key words Data Mining, Aggregation, Parallel Pipelining, Inter-node Communication

目次

図目次

表目次

第１章

序論

巨大データベースからの知識獲得を高速に行い、意思決定を支援する技術としてデータマイニングや OLAP(On Line Analytic Processing) が注目されている。データマイニングとは、従来のデータベースのように単純な問い合わせに対する答を出力するのみでなく、データ属性間に成立するルールを導出するような、高度な処理を指す [3]。データマイニングや OLAP は、大規模データベース中の属性とそれらの間の関係を、さまざまな観点から分析するために、多次元データキューブとして扱っており、典型的なデータキューブの各セルには、総売上高などの集約演算の結果が格納されている。このようなシステムを実現するためには、多くのセルについてそれぞれ集約演算を行う必要がある。特に OLAP では、オンライン性が要求されるため、応答時間が非常に重要な要素となっている。現在、このような多くの集約演算を処理するために、高速ネットワークをもつ非共有型並列計算機上における複数の集約演算に対する並列アルゴリズムが提案されている [1]。

計算すべき集約演算が Q 個ある時、単一集約演算用のアルゴリズムを Q 回繰り返すことで、すべての集約演算を処理できる。しかしながら、同一のデータベース上においては、さまざまな集約演算が同じソースリレーションから計算されるので、複数の集約演算を並列に行うように拡張できる。単一集約演算用のアルゴリズムとして提案されている、２ P アルゴリズム、Rep アルゴリズム [4]、BC アルゴリズムを複数の集約演算用に拡張したものがそれぞれ、ｍ２ P アルゴリズム、ｍ Rep アルゴリズム、ｍ BC アルゴリズムである [1]。

最初に集約演算における、リレーションに対しての基本操作を示す。

- リード (Read)

  ローカルディスクからメモリまたはバッファへデータを読み込む。

- ハッシュ (Hash)

  読み込んだデータから、group-by 問い合わせで指定される属性のデータを分類する。

- 集計 (Aggregate)

  ハッシュの結果であるハッシュテーブルを用いて、集約演算を行う。ハッシュテーブルに必要なメモリサイズは、group-by 演算の結果得られるグループ数に比例する。仮にグループ数が多く、ハッシュの結果、そのデータが属するグループがメモリからあふれるような場合には、集約演算を行わず、データをメモリ上のパーティションファイルに書き出す。

- エクストラプロセス (Extra Process)

  集計でメモリあふれが生じた場合には、パーティションファイルから逐次、データを読み出して集約演算を行う。これをパーティションファイルの数だけ繰り返す。

- エクストラ IO(Extra IO)

  メモリがあふれて、パーティションファイルを作成する場合、これが余計な入出力となって、アルゴリズムのパフォーマンスに影響を及ぼす。この余計な入出力をエクストラ IO という。

基本的な集約演算の流れは (1) リード,(2) ハッシュ,(3) 集計, である。全ての集約演算のソースリレーションについて (1) から (3) を適用する。以上の基本操作を用いて、以下に 3 つのアルゴリズムの動作を説明する。

1. m2P アルゴリズム

   2P アルゴリズムでは 2 つのフェーズに分けて処理を行う。第一フェーズにおいて、各ノードは 1 回のリードに対して、Q 個のハッシュ及び集計を行う。第 2 フェーズでは、各ノードの部分的に得られた結果をさらに集計して最終結果を求める処理を行う。m2P アルゴリズムは、2P アルゴリズムの第 2 フェーズを並列に行うように拡張したもので

ある。m2P を用いると、Q 個の集約演算に対して、ソースデータを 1 度しか読み込む必要が無いため、読み込みコストは減少する。しかし、必要なメモリ量が 2P アルゴリズムの Q 倍になり、エクストラ IO のコストが増大するなどの問題点が挙げられる。

2. mRep アルゴリズム

Rep アルゴリズムは、1 度のリードに対して Q 個の集約演算に対するハッシュを行い、集計を担当するノードへデータの送信 (再配分) を行う。受信したノードは、それぞれのデータに対して集計を行う。mRep は後者の、Q 個の集約演算を並列に処理する。mRep を用いて複数の集約演算を並列に実行すると、大きなメモリを必要とするが、基本となる Rep アルゴリズムが、2P アルゴリズムよりも効率的にメモリを扱うことから、複数の集約演算を処理するのに有利である。しかし、Q 個の集約演算を処理する mRep アルゴリズムの通信コストは、単一アルゴリズムを処理する Rep アルゴリズムの Q 倍大きくなるなどの問題点が挙げられる。

3. m BC アルゴリズム

BC アルゴリズムは、各ノードが並列にリードしたデータをすべてのノードにブロードキャストし、集計を担当するノード以外は不要なデータを破棄するアルゴリズムである。mBC アルゴリズムとは、1 度のブロードキャストに対して、受信したノードが集約すべきグループならば Q 個の集約演算に対するハッシュと集計を並列に行う。mBC アルゴリズムのコストは BC アルゴリズムのコストと同じであるのに対して、他のアルゴリズムは Q 倍となっている。しかし、mBC アルゴリズムはソースリレーションと比べて、少数の集約演算を処理するのには適していないという問題点もある。

本稿では、データベースの巨大化が進んでいる現状を鑑みて、より大規模な並列計算機上で効率よく動作する多重集約演算の並列アルゴリズムを提案し、評価する。

第 2 章では、提案するアルゴリズムの手法、動作説明を行う。

第 3 章では第 2 章で提案したアルゴリズムに並列パイプライン化を適用した時の全体構成及び、利点について詳しく述べる。

第 4 章では、並列パイプライン化の有効性の評価を行った後、mBC アルゴリズム、mRep アルゴリズムとの性能比較を行い、提案方式の有効性を示す。

第 5 章では、本論文の結論ならびに今後の展望について考察する。

第 2 章

提案するアルゴリズム

2.1 既存のアルゴリズムの問題点

文献 [1] によると、m2P,mRep,mBC の 3 つのアルゴリズムについて、ブロードキャストバスをもつハードウェアモデル上で評価したところ、集約演算の対象となるデータ数が大きい時に最も高速にデータを処理するのは mBC アルゴリズムである。しかし、全てのノード間が直接結ばれているようなハードウェアモデルでは、より大規模な超並列計算機は実現不可能である。そこで本章では、基本処理ユニットであるノードを拡張性に優れたリング状の通信路で接続した構造に適した、ノード間通信アルゴリズムを提案する。

2.2 動作説明

提案するアルゴリズムの基本的なアイデアは、リング状に接続された多数のノードが演算と通信を交互に繰り返すことにより、システム全体をあたかも巨大なパイプラインとみなして処理を進行させるという点にある。図 2.1 に示すように、ローカルディスク (Localdisk)、バッファ(Buffer)、メモリ (Memory)、プロセッサ (Processor) からなる基本処理ユニットをノードと呼ぶ。

各ノードのディスクは、重複しないように分割された初期データを持っており、最終的な集約演算も、各ノードで異なるように割り当てる。これらのデータを並列にハッシュしていくと、各ノード毎に集計すべきデータの種類が異なり、自ノードで集計すべきデータと集計すべきでないデータが発生する。自ノードで集計すべきデータは各ノード毎に処理され、集

計すべきでないデータは次ノードのバッファへ送信される。本アルゴリズムではノード数に対して、m BC アルゴリズムでは単位時間あたり最大 $O(n^2)$ であった通信量が、$O(n)$ へと削減できる。

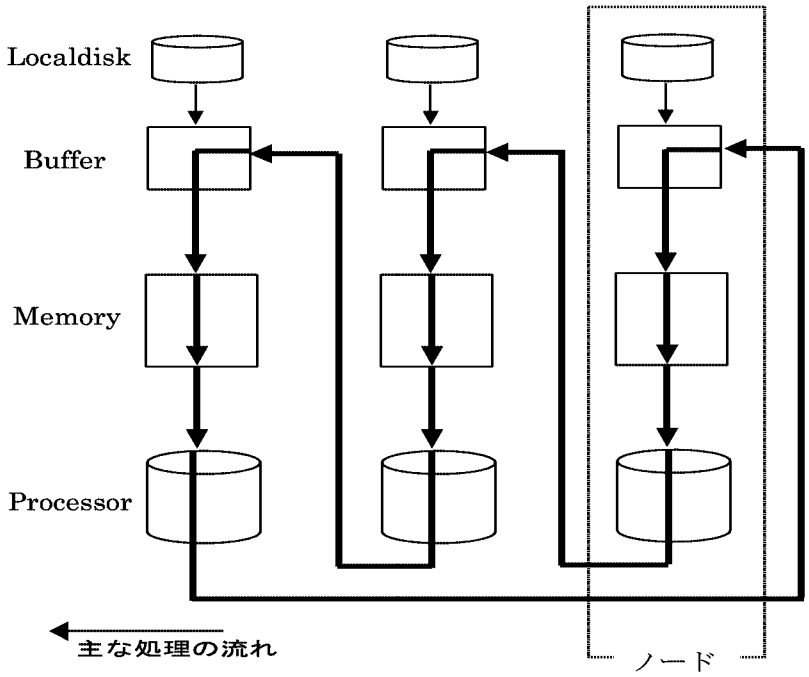

図 2.1　全体のハードウェアブロック図

次に本アルゴリズムの動作の手順を説明する。図 2.2 に動作手順をシーケンス図にて記す。

(1) ディスクからバッファにデータを送信する。

(2) バッファからメモリへデータを送信する。

(3) Q 個のデータに対して並列にハッシュを行う。

(4) 集約すべきデータは各ノード毎に集約し、集約すべきでないデータに関してはプロセッサから、次ノードのバッファへ送信する。以後、(1)～(3) を繰り返し、ディスクから送信されたデータすべてに関してハッシュを行う。

(5) すべてのデータに対してのハッシュ終了後、Q 個のデータに対して、同時に集計を行う。

2.2 動作説明

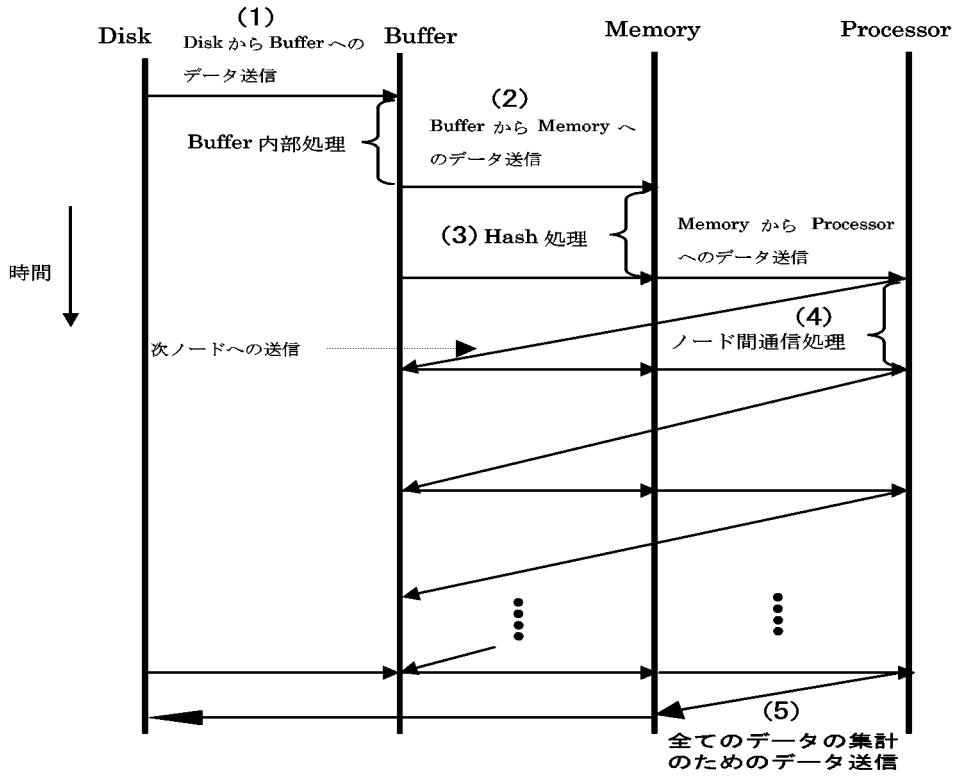

図 2.2　本アルゴリズムシーケンス図

以上のような流れで処理を行うのであるが、解決すべき問題点が 3 つ挙げられる。

まず 1 つ目の問題として、図 2.3 に示すようにバッファは基本的に前のノードからデータを受け取る役割と、メモリへのデータ送信する役割、ディスクからのデータの受信という処理を行わなければいけないが、これらすべてを同時にこなすことは不可能である。前のノードからのデータの受信とメモリへのデータの送信は同時にバッファは処理できる。またディスクからのデータの受信とメモリへのデータ送信も同時に処理を行うことができる。しかし、前のノードからのデータの受信とディスクからのデータの受信は同時に処理することはできない。したがって、次ノードへのデータ送信のタイミングと、ディスクからのデータ送信のタイミングの調整が重要となる。

2.2 動作説明

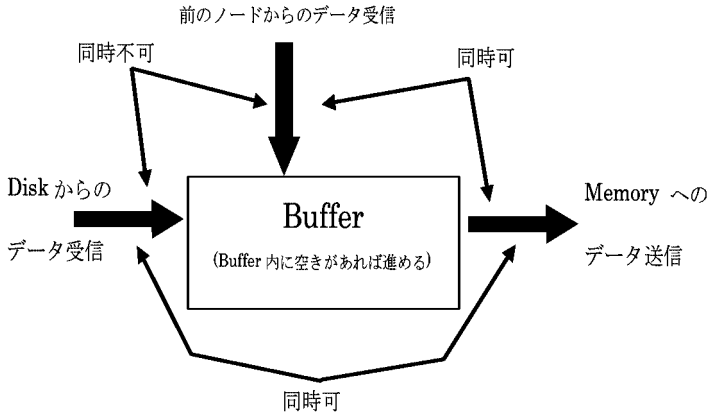

図 2.3　Buffer の役割

まず、ノード間通信のタイミングについて検討する。ノード間の通信はなるべくデータをまとめて通信したほうが通信回数が減り合理的である。しかし、一度に大量のデータを送信することで、バッファのオーバフローがおきやすくなる。そこで、バッファ内部を P フェーズに分けた図 2.4 のような流れの処理を提案する。

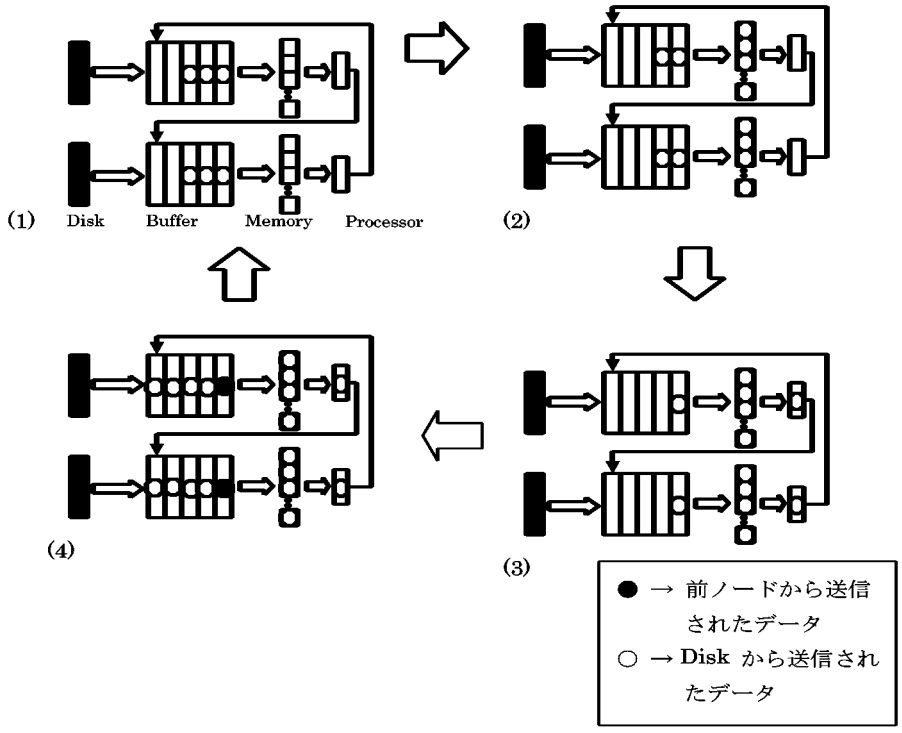

図 2.4　ノード間通信アルゴリズムの動作

2.2 動作説明

図 2.4 の動作の流れを下記に記す。

(1) データがディスクからバッファ内部に読み込まれて格納されている。この時、各ノード毎に並列にデータがバッファ内部に読み込まれる。図 2.4 のバッファ内部は P=5 フェーズに分かれており、ディスクから 1 度のリードで 3 フェーズ分のデータを読み込んだことを表す。図 2.5 には 2 つのノードでの構成図しか図示していないが、ノード数が増えた場合も同様に、どのノードのバッファ内部も P フェーズに分けられ、この処理は並列に行われる。

(2) バッファ内部に読み込まれたデータのうち 1 フェーズ分のデータをメモリへ送信し、メモリ内部で並列にハッシュする。この処理は各ノード毎に並列に行われる。

(3) (2) でハッシュされたデータは、自ノードで集約すべきデータに関しては自ノードのメモリ内に格納し、そうでないデータに関しては次のノードに送る。この時、並行してメモリ内では、再びバッファ内部から 1 フェーズ分のデータがメモリ内に送信され、ハッシュ処理が行われる。

(4) 1 回のディスクからバッファへの送信によって得られたデータのハッシュ処理を行い、バッファ内部に N フェーズ分の空きができたら再びディスクからバッファへとデータを送信していることを示している。図では N=4 となっている。空きフェーズが N に達しない場合、ディスクからのデータ読み込みは行われない。ノード毎に、(1)〜(4) までの処理を繰り返すことによってすべてのデータに対するハッシュが完了する。

ノード間通信は 1 フェーズ分のデータをハッシュし、自ノードで処理すべきでないデータはそのつど次のノードへ送信するものとする。送信するタイミングは、各ノードのバッファがメモリへデータ送信を行う間に送信するものとする。

次にディスクから再度新しいデータを送信するタイミングについて検討する。まず、バッファ変数 P が大きいほど、システム全体の性能は向上する [5]。しかし、本アルゴリズムの実現に必要な高機能を実現するバッファのハードウェアコストは極めて高く、かつ O(p)

で増加する。本稿では、アルゴリズムの検討を中心に行うため、ハードウェアの条件が最も悪い状態を想定する。すなわち、ディスクからバッファへ送信するデータ量 M に対して、1 フェーズ分のデータしか格納されていない時に、ディスクからバッファへの読み込みを行うものとする。つまり、ノード間通信により、前のノードから送信されるデータ量は、1 回に高々1 フェーズ分のデータ量であり、バッファ内部に格納されているデータ量が残り、1 フェーズ分になった時にディスクからデータ送信をするので、内部を P フェーズに分けるとすると、バッファオーバフローを起こさないためには $P \geq 2 + M$(フェーズ) という式が成り立つ必要があり、第 4 章でのアルゴリズム評価では P=2+M と想定する。

2 つ目の問題点は、プロセッサ間通信速度と送信するタイミング、ハッシュ処理速度、ディスクからバッファへのデータ送信速度とタイミング、ディスクへのデータ送信速度とタイミングの問題からバッファオーバフローという問題である。本アルゴリズムのバッファでは、ディスクからの送信データ、前のノードからの送信データ、メモリへの送信データを制御し、効率の良い処理を行わせる役割を果たしている。もしある 1 つのノードのバッファがオーバフローしてしまうと連鎖的に各ノードのバッファがオーバフローしてしまい、動作不可能な状態が生まれてしまう。そこで、図 2.5 にて示すように、自ノードのバッファがオーバフローを起こしている時は、オーバフローを起こしているバッファ内部のデータを自ノードのディスクへ送信し、バッファ内部に空き容量をつくることでバッファのオーバフローの連鎖を防ぐ。ディスクへの送信はバッファ→メモリ→ディスクの順に行われる。この処理は、プロセッサ時間をかなり消費するが、後に述べる簡単なハッシュ処理プログラムの試作によれば、処理負荷を調整し、適当なバッファの空きを確保することで、バッファのあふれをほとんど避けることができる。このため、今回はソフトウェアによる対策をとるものとする。

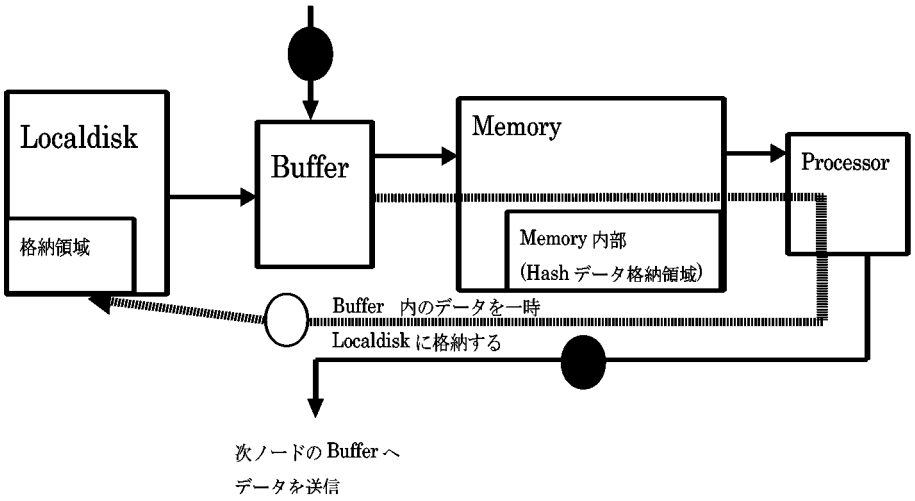

図 2.5　Buffer のオーバフロー

3 つ目の問題としては、ハッシュ処理した結果テーブルのテーブルをメモリ内部にて格納するのだが、メモリの空き領域が足りなくなった時のエクストラ IO の問題が挙げられる。エクストラ IO が生じるのは、(1) ハッシュテーブルあふれ,(2) パーティションファイルあふれの 2 種類がある。この時も、メモリ内のデータ格納領域が一杯になった時点で、メモリに格納されているデータを自ノードのディスクへ格納する。ディスクへの送信は図 2.6 に示すように、メモリ→プロセッサ→ディスクの順に行われる。

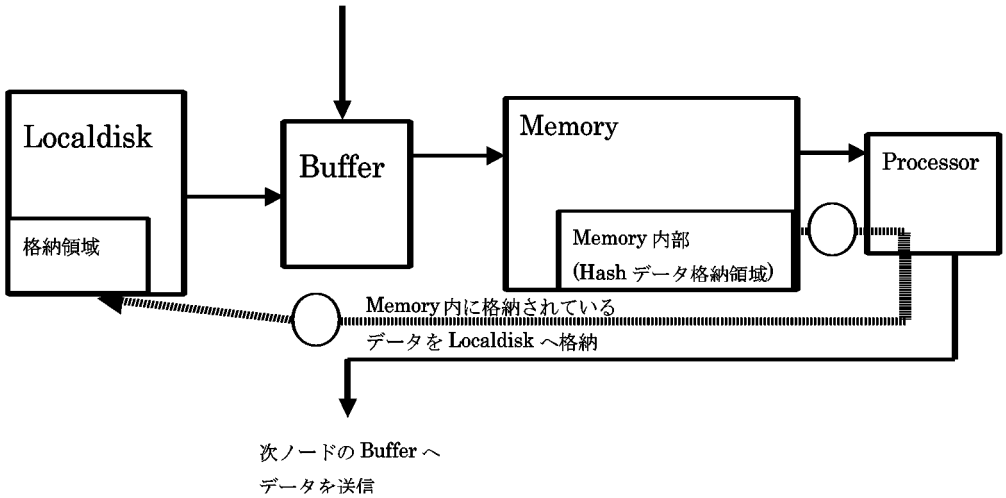

図 2.6　Memory のオーバフロー

(1) のエクストラ IO が生じた場合、最終的な集計の時に、すべてのデータに対してのハッシュが終了した後、メモリ内に格納されているハッシュ結果を一度すべてディスクへ送信し、グループ毎に最終的な集計を行う。

(2) のエクストラ IO が生じた場合は、一度ディスクに送信し、ディスク中のデータに割りこませて、再びハッシュを行う。

第 3 章

並列パイプライン化

3.1 並列パイプライン化構成

データマイニングでは、ソースリレーションや集約演算数が大きくなると入出力にかかる時間がボトルネックになりやすい。この入出力にかかる時間を最小化するために、本アルゴリズムのノード間通信時間を隠蔽するように、並列パイプライン化について検討する。並列パイプライン化した時のハードウェアは、第 2 章で提案したアルゴリズムと同様である。並列パイプライン化の適用前と適用後の違いは、ハードウェア毎の処理及び、各ハードウェア間の通信時間を同一にすることで、各ノード間が並列にパイプライン処理をおこない、入出力にかかる時間を最小化できるところである。また提案したアルゴリズムは、単一通信により全体の構造がリング構造となっているため、他のアルゴリズムと比べると並列パイプライン化の適用が容易である。

3.2 プロセッサ間通信コストの隠蔽

本節では、プロセッサ間通信コストの隠蔽について図 3.1 に示すシーケンス図を用いて説明する。図 3.1 に記されているシーケンス図内の矢印はデータの流れを表している。上から順にディスクからバッファへの矢印は図 3.1 における (1) のリードを、バッファからメモリへの矢印は (2) のデータ送信を表す。3 つ目のメモリからプロセッサへの矢印は、図 3.1 における (3) の処理すべきでないデータを次ノードへ送るための、メモリからプロセッサへのデータ送信を表し、最後のプロセッサからバッファへの矢印は (4) のノード間のデータ送信

を表している。

図 3.1 の、(a) はバッファ内部でのデータをフェーズ毎に分ける処理時間である。(b) ではメモリ内部でのハッシュ処理時間を表している。(c) はプロセッサ間通信時間を表している。この図を見ると、並列パイプライン化を適用することで、(c) のノード間通信部分に重複している、(a)、(b) の時間を隠蔽できることがわかる。したがって、はじめのプロセッサ間通信から最後のハッシュ処理をするまでの (a)、(b)、(c) の部分の隠蔽が可能となる。そこで重要となるのが隠蔽率の最適化である。隠蔽率とは、隠蔽される時間の割合である。例えば (c) の時間が (a)、(b) の時間より大きい場合、(c) の時間により (a)、(b) の時間が隠蔽されるが、この時 (c) の時間と (a)、(b) の時間との差が小さいほど隠蔽率が高いといえる。パイプライン化するにあたって隠蔽率の最適化が重要な要素となる。

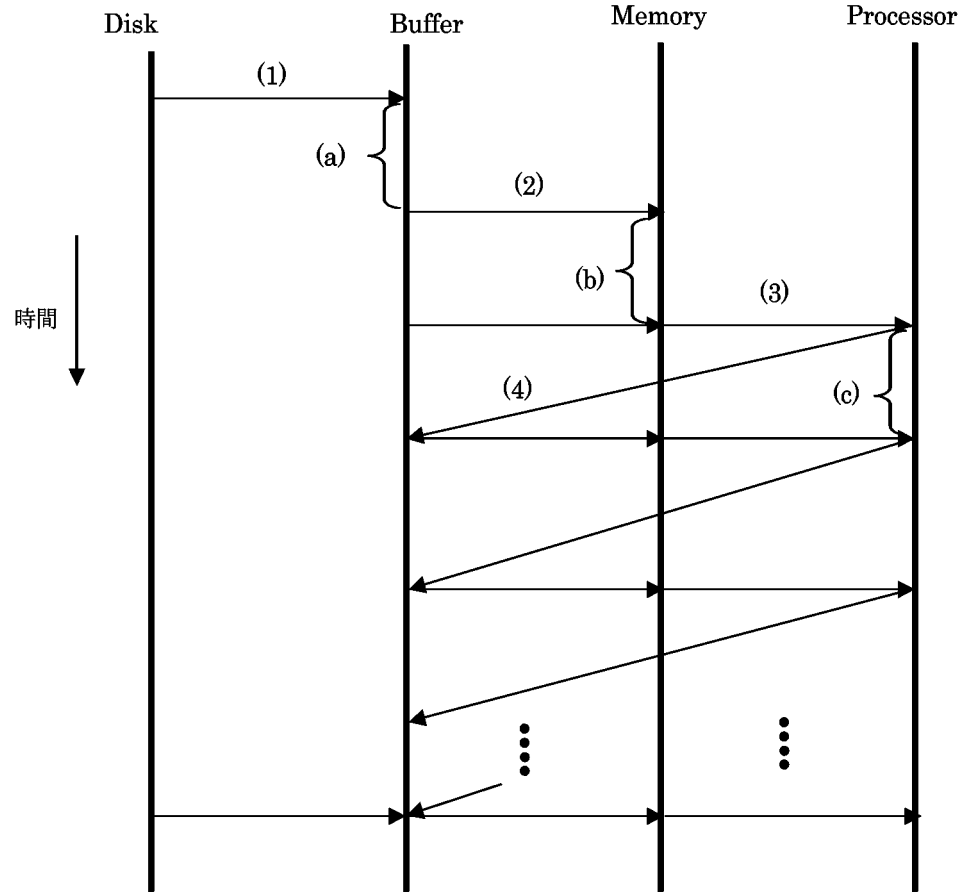

図 3.1　プロセッサ間通信アルゴリズムシーケンス図

3.3 Buffer 内部構成及び動作説明

ここでは、本アルゴリズムの処理の流れで、重要な役割をもたらすバッファ内部の処理説明を行う。図 3.2 にバッファ構成を示す。

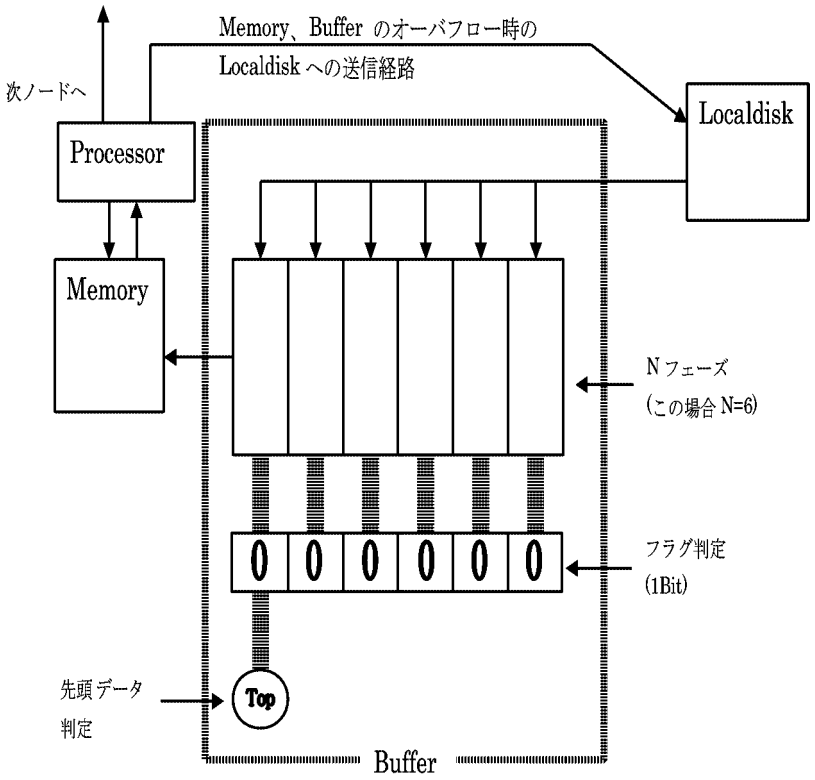

図 3.2 Buffer 構成

まず、バッファの内部構成について説明していく。バッファ本体は N フェーズに区分けされており、それぞれのフェーズ毎にデータの有効性を判定するための 1Bit 分のハードウェアフラグが取り付けられているものとする。またバッファ内にデータが格納された順番どおりにメモリへ読み込まれるため、どのフェーズにあるデータが先頭のデータであるかという判別を行う必要性がある。これもフラグ判定部分と同様にハードウェアにて実装するものとし図 3.2 では Top として表す。以上のようなハードウェアを用いてバッファ内部の処理を行う。

3.3 Buffer 内部構成及び動作説明

図 3.3 にバッファの処理の流れを示し、図の流れに沿ってバッファの動作を説明する。

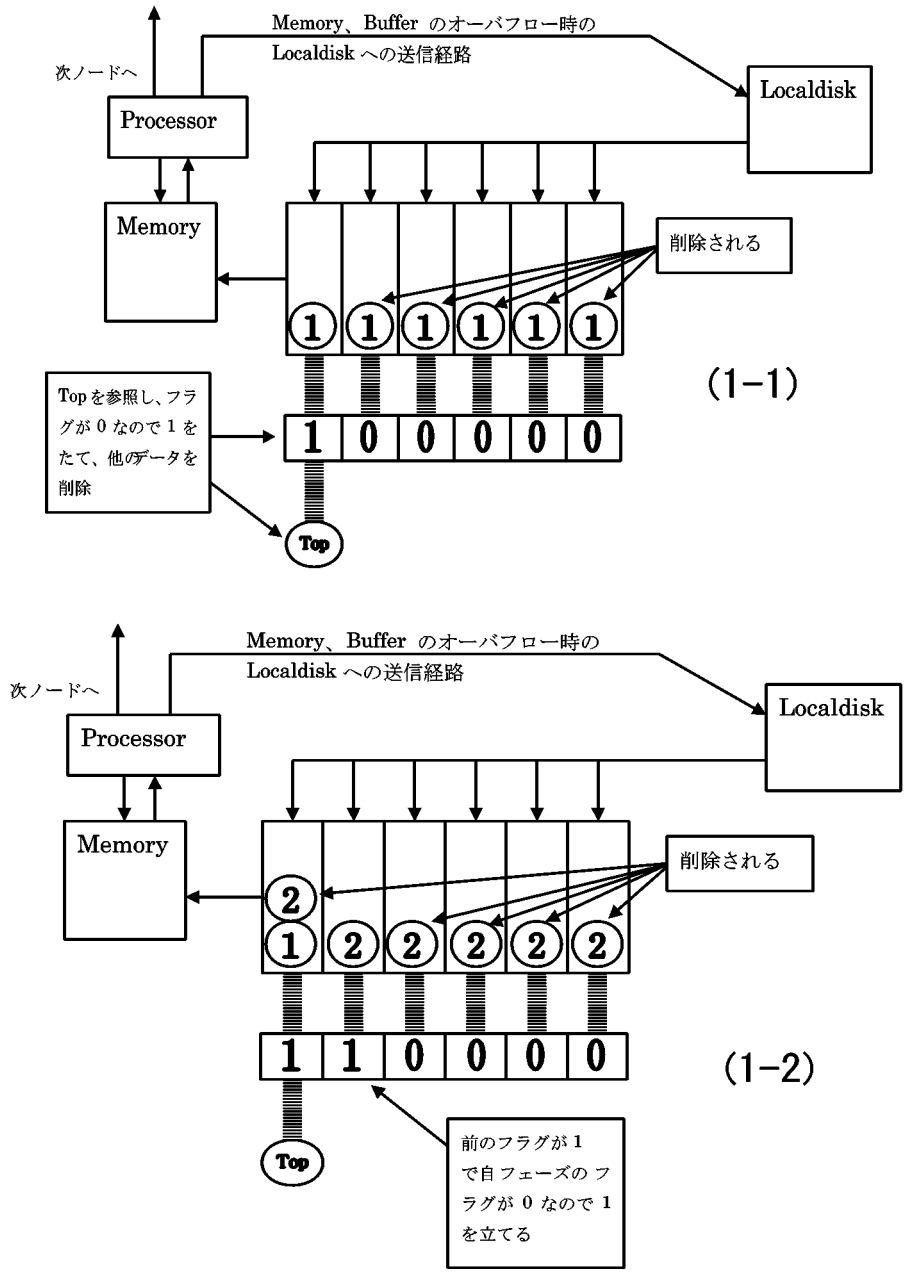

図 3.3 Buffer へのデータ送信

(1-1) ここではまずディスクから 1 ページ分のデータを読み込み、それらを 1 フェーズ分のデータに区分けする流れを記している。まず 1 フェーズ分の先頭のデータをすべてのフェー

ズに入力する。この時、各フェーズ毎にフラグ判定が行われる。0 のフラグがたっている場合は、そのフェーズには有効なデータが格納されていないということを表し、1 のフラグがたっている場合は自データの前に他のデータがあることを表す。しかし、ここで入力されているデータは最初のデータ (バッファ内部が空の状態の時) なので、すべてのフラグは 0 となっている。そこで top を参照し、そのフェーズ部分のフラグ判定が 0 である場合、そのフェーズに入力されたデータを先頭データとし、そのフェーズのフラグに 1 をたてる。また、そのフェーズ以外に入力されているデータはすべて削除される。

(1-2) 次に入力されたデータも、(1-1) と同様にまず全フェーズに入力される。この時、(1-1) にて入力されたフェーズでは 1 のフラグがたっており、それ以外のフェーズでは 0 のフラグがたっている。そこで、自フェーズのフラグが 0 で、1 つ前のフェーズのフラグに 1 がたっているフェーズを探し、0 のフラグを 1 にして、0 のフラグがたっているフェーズのデータを削除する。以後この (1-2) と同様の処理をすることで、1 ページ分のデータがバッファ内においてフェーズ単位での区分けが可能となる。

上記の流れでバッファ内にデータを格納していく。次にメモリへデータを出力する時のバッファ内の処理について述べる。図 3.4 にメモリへのデータ出力の流れを記す。

(2-1) まずメモリへデータを出力するわけであるが、どのデータを送信するかを述べると、top のデータをメモリ内に出力する。出力されたデータが格納されていたフェーズは空となっているため、0 のフラグを立てる。また top は次フェーズに移行される。したがって top のデータが順にメモリへ送信されていく。

(2-2) ここでは、前のノードからの送信データについてのバッファ処理を述べる。前のノードから送信されたデータも (1-2) と同様に処理されバッファ内に格納される。

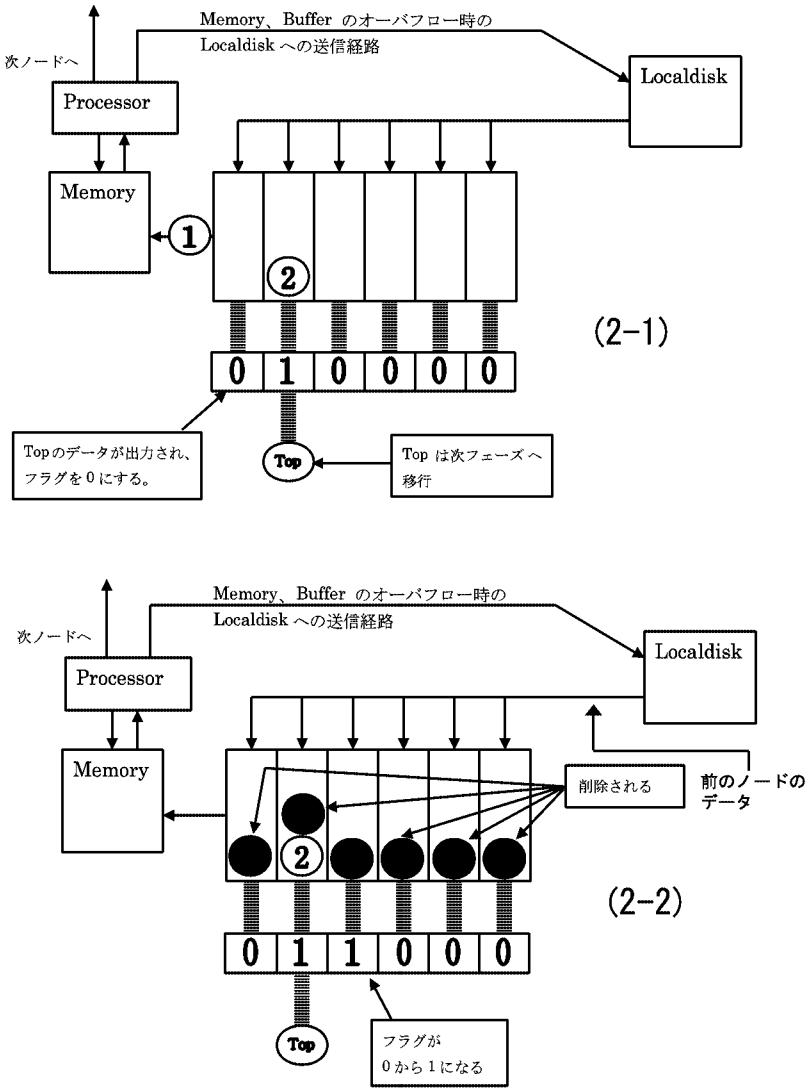

図 3.4 Memory へのデータ送信と前のノードからのデータ受信

以上の (1-1),(1-2),(2-1),(2-2) の流れでバッファの処理は行われる。しかし、この一連の流れはバッファがオーバフローを起こしていない時の流れである。そこでオーバフローが発生した時のバッファ内部の処理を次に記す。図 3.5 はバッファのオーバフローが起きた時のバッファ内の状態を示している。

この場合フラグはすべてのフェーズに 1 がたっており、前のノードから送信されてきたデータを格納する場所が無く、バッファはオーバフローを起こし、この状況が連鎖的に各ノードに飛火してしまう。そこで、すべてのフラグに 1 たっている場合には、バッファ内にあるデータ (top 部分のデータ) を一時的に自ノードのディスクに格納することでバッファ

内部に空きフェーズを設ける。この時、前のノードはバッファからオーバフローであるという信号を受け取り、一時的に次ノードへの送信をとめておく。バッファ内に空きフェーズができると、前のノードは次ノードへのデータ送信を開始する。また、ディスクに送信されたデータは以後、処理される。以上の流れでオーバフローの問題を解決する。

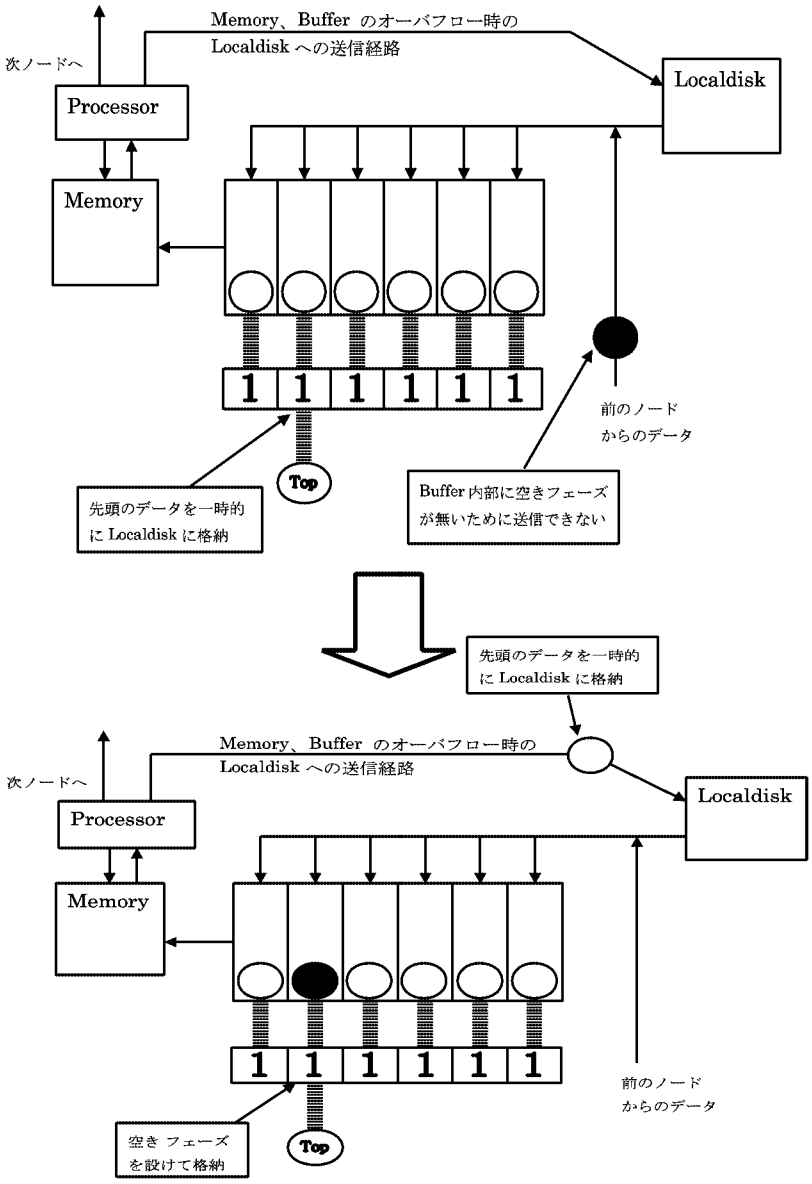

図 3.5 Buffer のオーバフロー

第 4 章

性能評価

4.1 コストモデル

文献 [1] のコストモデルパラメータを表 4.1 に示す。この表では、1 つのリレーションに対して、Q 個の集約演算が存在し、各集約演算は A 個の集約演算を含んでいると仮定している。これらはディスク上に格納されたリレーションに対して直接、働くものとして、事前に各ノードが分配されているものとする。

文献 [1] のコストモデルは、ネットワークの調停、CPU、I/O などの重複を含んでおらず、ノード間にブロードキャストバスを持つ IBM SP2 を対象としている。この結果、本稿で想定するようなデータ量が大きい場合には、mDC が最も高速に動作するという証言がなされている。これに対して、本稿で用いるコストモデルは、ノード間の接続がリングにより行われていると仮定している点が異なるが、他の点は文献 [1] のモデル及びパラメータをそのまま用いる。また、すべてのプロセッサが完全に並列に働くものとしている。このような前提条件により単一ノードにおける部分的な I/O と CPU のコストの合計から全体のパフォーマンスを計算することができる。

4.1 コストモデル

表 4.1 コストモデルのパラメータ

記号	説明	値
R	リレーションのサイズ [byte]	5 GB
$\|R\|$	R のタプル数	10 million
$\|R_i\|$	ノード i に分配されたタプル数	$\|R\|/N$
$\|R_p\|$	再配分されたタプルの数 (mBC)	$max(\|Ri\|, 1/SQ)$
Q	集約演算の数	変数
T	ソースリレーションを読み込む回数	変数
A	1 つの集約演算中の集約演算の数	100
S	グルーピングセレクティビティ	変数
p	プロジェクティビティ	0.5
P	ページサイズ	4KB
M	ハッシュテーブルサイズ	50MB
IO	ディスクから 1 ページをバッファに 読み込む/書き込む時間	0.8ms
t_r	1 タプルを Buffer へ書き込む時間	200/mips
t_w	1 タプルを Buffer へ書き込む時間	200/mips
t_h	Hash を計算する時間	200/mips
t_a	集約演算を処理する時間	40/mips
t_m	ノード間で 1 ページを送信する時間	0.1ms
t_b	1 ページをブロードキャストする時間（m BC)	$(N-1)*N*tm$
N	ノード数	変数
G	結果のタプルサイズ	$\|R\|S$

次に表 4.1 に表記されているコストモデルパラメータを用いて、m BC アルゴリズムと mRep アルゴリズムと並列パイプライン化適用前と適用後の提案アルゴリズムの 4 つのコストモデルを示す。

(mBC アルゴリズム)

(1)Disk から Buffer への読み込みコスト: $(Ri/P) * IO * T$

(2) 通信 Buffer からタプルを取り込むコスト: $|R| * tr * T$

(3)Hash のコスト: $|R| * th * T$

(4) 集約演算のコスト: $|Rp| * ta * A * Q$

(5)extra IO のコスト: $(Rp * p * Q - M/S * T)/P * 2 * IO$

(6) 結果タプルを生成するコスト: $|Rp| * S * tw * Q$

(7) ローカルディスクに格納するコスト: $Rp * S/P * p * IO * Q$

(8) ブロードキャストするコスト: $(Ri/P) * tb * T$

(mRep アルゴリズム)

(1)Disk から Buffer への読み込みコスト: $(Ri/P) * IO * T$

(2)Buffer からセレクトするコスト: $|Ri| * tr * T$

(3)Hash するコスト: $|Ri| * th * Q$

(4) 通信 Buffer に書き込むコスト: $|Ri| * tw * Q$

(5) 再分配のための送受信コスト: $Rp/P * p * tm * Q * (N-1) * N$

(6) 受信したデータを通信 Buffer から読み込み、
集約演算を計算するコスト: $|Rp| * (tr + ta * A) * Q$

(7)Extra IO コスト: $(Rp * p * Q - M/S * T)/P * 2 * IO$

(8) 結果タプルを生成するコスト: $|Rp| * S * tw * Q$

(9) ローカルディスクに格納するコスト: $Rp * S/P * p * IO * Q$

4.1 コストモデル

(並列パイプライン化適用前の提案アルゴリズム)

(1)Disk から Buffer への読み込みコスト: $(Ri/P) * IO * T$

(2) 通信 Buffer からタプルを取り込むコスト: $(Ri + Ri * (N-1)/2) * th * Q$

(3)Hash のコスト: $(Ri + Ri * (N-1)/2) * th * Q$

(4) ローカルディスクへの格納コスト: $((Ri*(N+1)/2*Q*p*1/N) - (M/S*T))/P*2*IO$

(5) 結果タプルを生成するコスト: $(Ri * (N+1)/2 * Q * 1/N) * S * tw$

(6) プロセッサ間通信コスト: $Ri/P * T * tm * Q * (N-1)/2$

(7) 最終結果を Disk へ格納するコスト: $(Ri * (N+1)/2 * Q * S * p * 1/N)/P * IO$

(並列パイプライン化適用後の提案アルゴリズム)

(1)Disk から Buffer への読み込みコスト: $(Ri/P) * IO * T$

(2) 通信 Buffer からタプルを取り込むコスト: $(Ri + Ri * (N-1)/2) * th * Q$

(3)Hash のコスト: $(Ri + Ri * (N-1)/2) * th * Q$

(4) ローカルディスクへの格納コスト: $((Ri*(N+1)/2*Q*p*1/N) - (M/S*T))/P*2*IO$

(5) 結果タプルを生成するコスト: $(Ri * (N+1)/2 * Q * 1/N) * S * tw$

(6) プロセッサ間通信コスト: $Ri/P * T * tm * Q * (N-1)/2$

(7) 最終結果を Disk へ格納するコスト: $(Ri * (N+1)/2 * Q * S * p * 1/N)/P * IO$

(注) 並列パイプライン化を適用することで、(2)、(3)、(4)、(6) のコストの間で隠蔽が可能となる。

上記の並列パイプライン化適用前と適用後の本アルゴリズムのコストモデルでは、各ノードごとにデータが平均的に分かれており Hash 処理では均等に処理されるものと考えているので、バッファのオーバフローはおきないものと考えている。また、上記の提案アルゴリズムのコストモデルはm BC アルゴリズムとハードウェアの量を同じにしているため、バッファを追加した時のコストをは新たに発生しない

ものとしている。以上のことを踏まえて次にm BC アルゴリズム、mRep アルゴリズム、並列パイプライン化適用前と適用後の提案アルゴリズムのコストモデルでの評価を行う。

4.2　並列パイプライン化の効果

はじめに並列パイプライン化の適用前と、適用後の本アルゴリズムの性能向上率を調べる。性能向上率とは、この場合ノード数の増加に伴う性能の変化を表す。図 4.1 に並列パイプライン化の適用前と適用後の性能向上率のグラフを示す。このグラフはただし (Q=100,G=50000,T=100) である。また、ノード間の通信速度 tm が 10ms と比較的遅い場合である。グラフを見てもわかるように、パイプライン化を適用することで通信時間が隠蔽され、ノード数が少ない時に大きい性能の向上がみられる。また図 4.2 のグラフは、図 4.1 のノード数が多い時の部分を拡大したグラフである。このグラフを見ると、並列パイプライン化適応前と適応後の性能の差は、ノード数が少ない時ほど大きくないものの、性能が向上している。

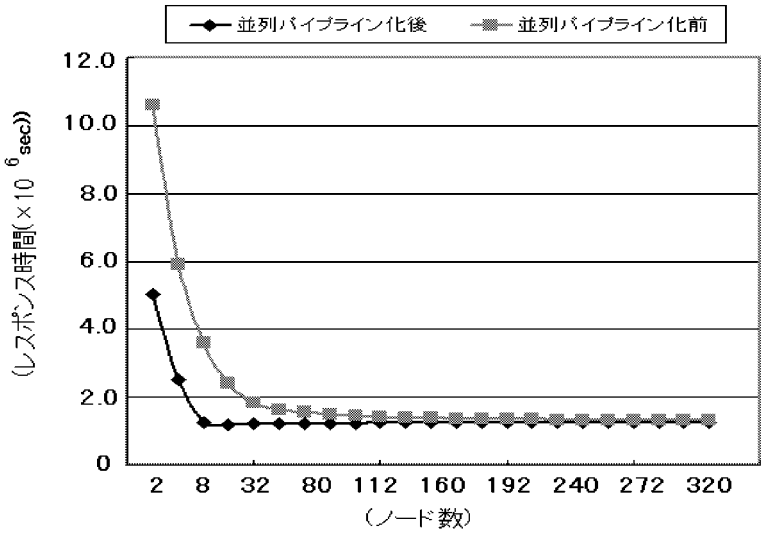

図 4.1　パイプライン化による性能向上率 (tm=10ms)

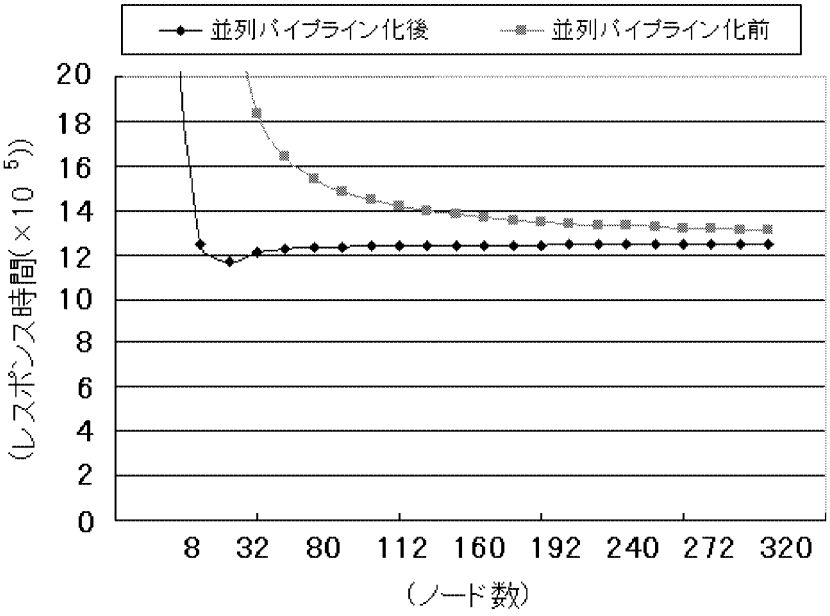

図 4.2　ノード数が多い場合のパイプライン化による性能向上率 (tm=10ms)

次に tm=0.01ms と通信速度が比較的早い場合にどのようになるのかを調べる。図 4.3 に Q=100,G=50000,T=100,tm=0.01ms の時の並列パイプライン化の適用前と適用後の性能向上率のグラフを示す。図 4.1 のグラフと比較すると、通信速度が速いので全体的に性能が向上しているのがわかる。また、通信速度が速くなっても通信時間の隠蔽による性能向上率は、比較的遅い時と同じようにノード数が少ない時に性能が向上していることがわかる。また図 4.4 は図 4.3 のグラフのノード数が多い時の部分を拡大したグラフであるが、これも通信速度が遅い時と同じように、並列パイプライン化適応前と適応後の性能の差は、ノード数が少ない時ほど大きくないものの、性能が向上している。

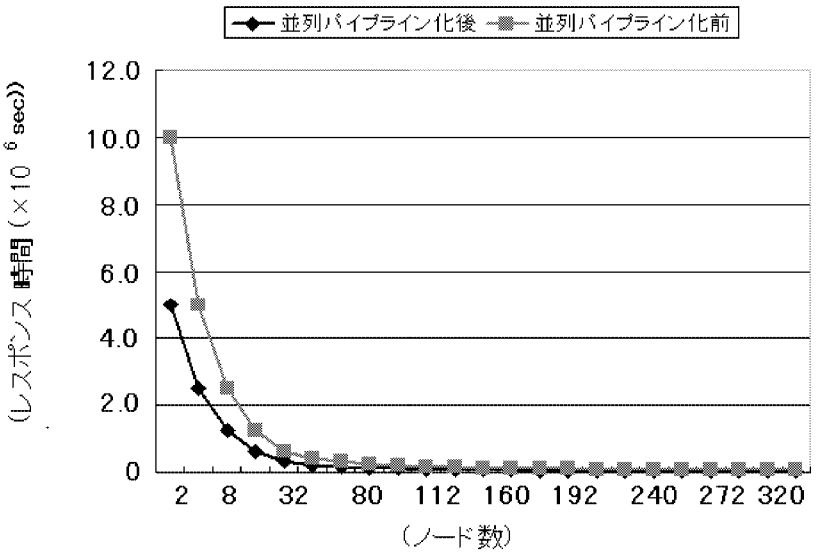

図 4.3 パイプライン化による性能向上率 (tm=0.01ms)

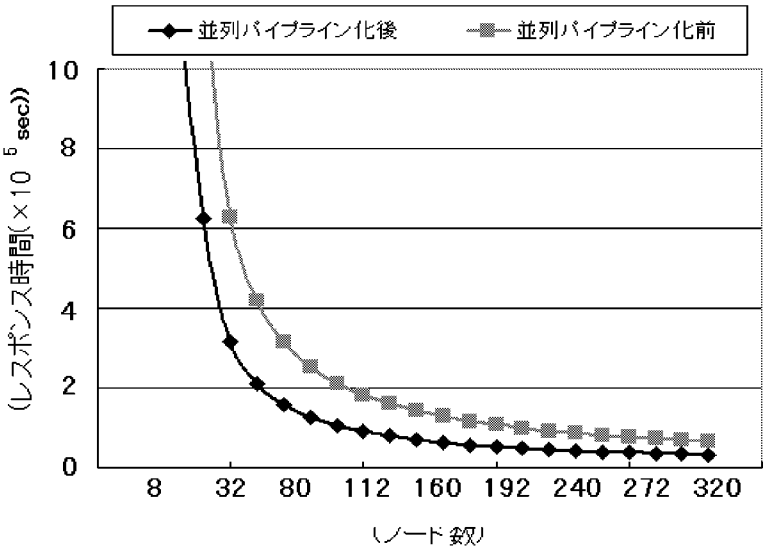

図 4.4 ノード数が多い時のパイプライン化による性能向上率 (tm=0.01ms)

最後に表 4.2 に tm が 10ms と 0.01ms の時に並列パイプライン化により隠蔽された時間を記す。この表を見てもわかるようにノード間の通信速度が速い場合に比べ、通信速度が遅い場合の方がより隠蔽される時間が多くなる。本節での評価結果から、本アルゴリズムは単一方向にしかノード間通信を行わないため、通信量があまり無くノード数の増加に対しての拡張性に優れている。また、並列パイプライン化を適用することによって更なる性能の向上が可能となるということが言える。また、通信速度が速い場合より、比較的遅い場合に隠蔽率が高くなることから、本アルゴリズムは通信速度が比較的遅い場合に適していることもわかった。

ノード数	tm=10ms の時の隠蔽時間 (sec)	tm=0.01ms の時の隠蔽時間 (sec)
2	5625567	5001201
4	3438250	2501688
8	2344591	1251935
16	1251669	627059
32	626718	314622
64	418400	210475
80	314242	158402
96	251747	127159
112	210084	106329
128	180324	91451
144	158004	80293
160	140645	71614

表 4.2 通信速度による隠蔽時間

4.3 mBC,mRep との比較

前節にて並列パイプライン化の有効性がわかった。そこで次に、並列パイプライン化を適用した本アルゴリズム、mBC アルゴリズム、m Rep アルゴリズムとの性能向上率の比較を

行う。図 4.5 に本方式と mBC 及び mRep との Q=100,G=50000,T=100 の時の性能向上率のグラフを示す。また、tm が 10ms と比較的遅い場合を判定する。このグラフから、mBC、mRep に比べて本アルゴリズムがノード数の増加に対しての拡張性がどのくらい優れているかが見てとれる。ノード数が 2～8 の時、mBC アルゴリズムが最も性能が優れており、順に mRep、本アルゴリズムになっている。しかしノード数を増加させると、まずノード数が 14 の時に mRep と mBC の性能が入れ替わり、さらにノード数を増やすことで、本アルゴリズムと mBC とではノード数が 48 以上の時に、mRep とではノード数が 96 以上の時にそれぞれ本アルゴリズムのほうが性能が優れていることがわかる。

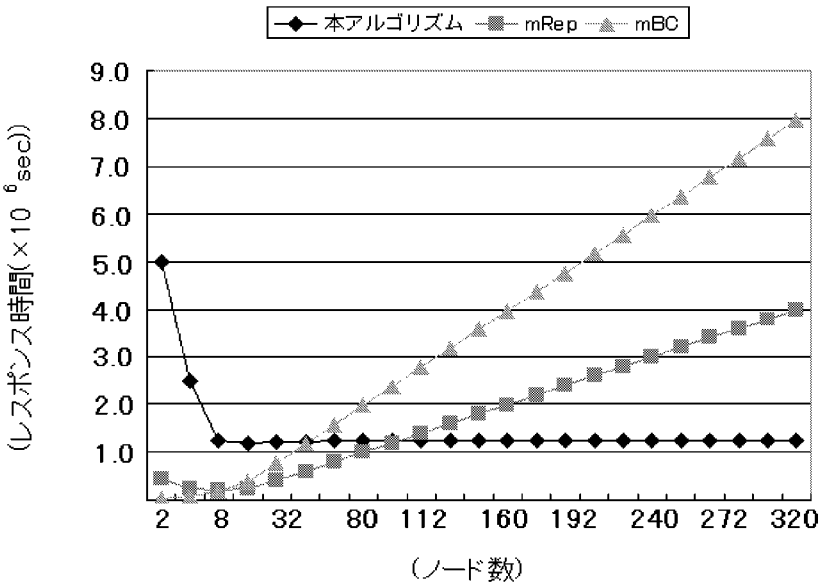

図 4.5　mBC,mRep, 本アルゴリズムの性能向上率 (tm=10ms)

次に tm=0.1ms、0.01ms と通信速度を速くした場合の性能を評価する。図 4.6 に tm=0.1ms の時、図 4.7 に tm=0.01ms の時のグラフをそれぞれ示す。

まず図 4.6 の通信速度が 0.1ms の時のグラフから、ノード数が 2～80 の時、mBC アルゴリズムが最も性能が優れており、順に mRep、本アルゴリズムになっている。しかしノード数が増加させると、まずノード数が 80 の時に mRep と mBC の性能が入れ替わり、さらにノード数を増やすことで、本アルゴリズムと mBC との比較ではノード数が 176 以上の時

に、mRep との比較ではノード数が 256 以上の時にそれぞれ本アルゴリズムのほうが性能が優れていることがわかる。

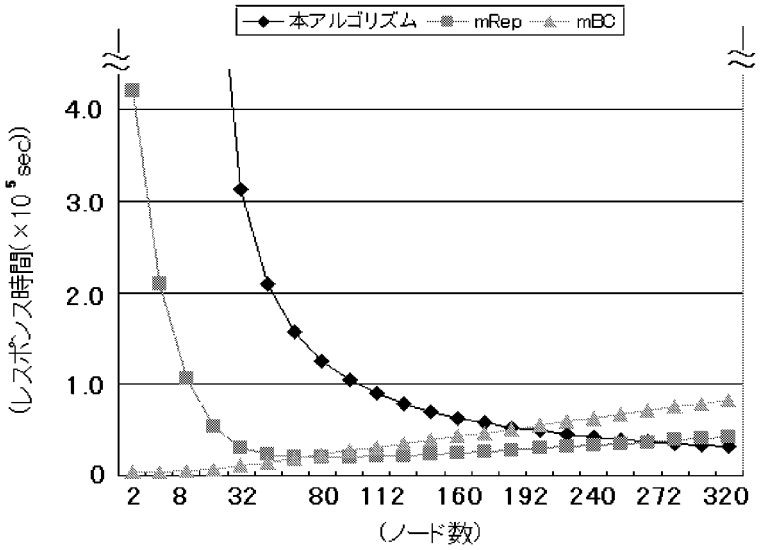

図 4.6　mBC,mRep, 本アルゴリズムの性能向上率 (tm=0.1ms)

さらに、図 4.7 の通信速度が 0.01ms の時のグラフを見てみると、ノード数が 2～160 の時、mBC アルゴリズムが最も性能が優れており、順に mRep、本アルゴリズムになっている。しかしノード数が増加させると、まずノード数が 160 の時に mRep と mBC の性能が入れ替わり、さらにノード数を増やすことで、本アルゴリズムと mBC とではノード数が 600 以上の時に、mRep とではノード数が 800 以上の時にそれぞれ本アルゴリズムのほうが性能が優れていることがわかる。

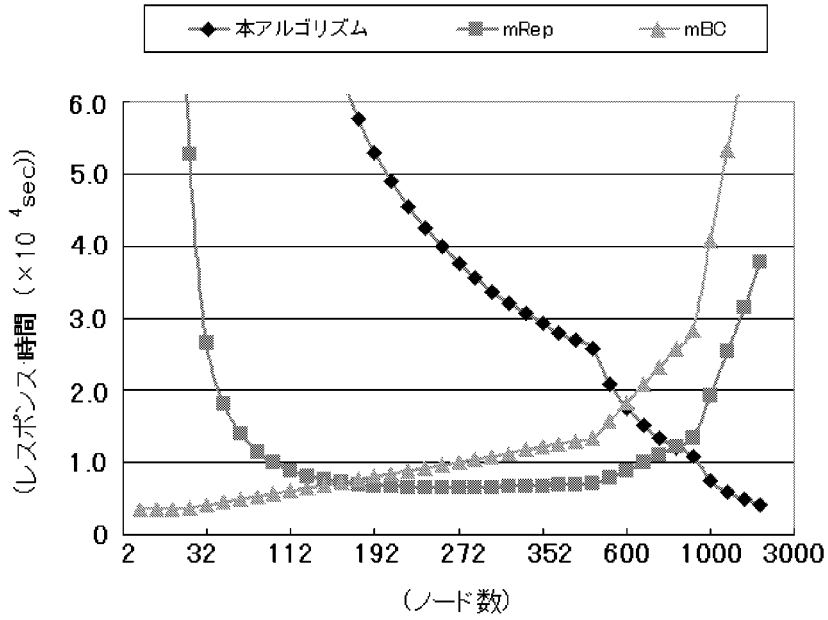

図 4.7　mBC,mRep, 本アルゴリズムの性能向上率 (tm=0.01ms)

これら、3 つのグラフから共通していえることは、ノード数の増加に対して mBC、mRep は性能が格段に落ちているのに対して、本アルゴリズムはノード数の増加に対して性能の減退が非常に少ないことが言える。なぜこのような結果になっているかを考察すると、mBC では 1 回の通信量が、ノード数 N に対して N^2 となるためノード数に対する、拡張性が少なく、大規模な並列計算機としては向いていないためである。mRep に関しては、1 回の通信量が mBC よりは少なく本アルゴリズムと同様であるが、通信経路は mBC と同様に N^2 になるため、ノード数の増加に対して、その分のコスト $O(n^2)$ でが増大し性能が減退する。それらに比べ本アルゴリズムは、通信経路が単一方向にしかなく、一回の通信量が各ノード毎に単一方向に 1 回と少ないためノード数の増加に対して、O(N) で性能が減退し最終的にほぼ一定の性能を得る。

4.4　DDMP 上での試作

実際の環境でメモリ内でのハッシュ処理の並列パイプライン化を検証するため、現在、寺田、岩田研究室で開発している、同時並行処理性、パイプライン並列性を活かした細粒

度の並列処理を得意とする自己タイミング型パイプラインを持つ、データ駆動プロセッサ (DDMP)[2] を対象に、単一プロセッサ用のハッシュ処理プログラムを作成し、シミュレーションを行った。その結果、レスポンスタイムは 337 サイクル ($7.5[nsec] * 10^{-6}$)、スループットは $2666,667[packet/sec]$ と処理速度はあまり速くは無かったが、新型プログラミングツールである DGPR を用いることで、現在の約 2〜4 倍の性能が得られる見通しを得た。また、今回作成したハッシュ処理プログラムを実行したところ、出力される全てのデータの間隔が均一であったため、処理の負荷を考慮したプログラミングにより、バッファのオーバフローは起こりにくくできることが裏付けられた。

第 5 章

結論

本稿では、多重集約演算の並列処理に関して、拡張性に優れたリング構造向きのノード間通信アルゴリズムを提案し、さらに並列パイプライン化を適用することで通信時間の隠蔽を可能とする方式を提案した。

第 2 章では、既存のアルゴリズムの問題点を述べ、提案するアルゴリズムについての構成及び、動作手順について説明した後、本アルゴリズムを実装するにあたっての問題点となるバッファのオーバフロー、メモリのオーバフロー、ノード間通信のタイミングについての詳細説明とその解決策を提案した。

第 3 章では、提案するアルゴリズムの通信時間を隠蔽するため, 並列パイプライン化を提案した。まず、並列パイプライン化適用前と適用後の違いについて述べ、全体の処理時間の中で、隠蔽される部分をシーケンス図を用いて述べ、最後に本アルゴリズムの特徴である拡張されたバッファの構成及び、処理について詳しく述べた。

第 4 章では、リング構造を対象とするコストモデルを考案し、並列パイプライン化適用前と適用後の本アルゴリズム、mBC、ノード数が増加した時に mBC より有利となる mRep の 4 つのアルゴリズムに対する性能評価を行った。まず並列パイプライン化の有効性を考察するため並列パイプライン化適用前と適用後の本アルゴリズムについて評価したところ、全体的に処理速度が向上することがわかると同時に、比較的ノード数が少ない時に有効であるという結果も得られた。次に、通信速度を変化させた場合、ノード間通信時間が比較的遅い場合に改善の効果が大きかった。さらに、mBC、mRep、並列パイプライン化した本アルゴリズムとの性能比較を行った結果、mBC あるいは mRep が最も速くなる場合もあるが、目標としていたノード数が大きい場合はいずれのアルゴリズムよりも並列パイプライン化した

本アルゴリズムが高速であった。最後にハッシュプログラムを作成し、本アルゴリズムの動作を確認した。

今後の課題としては、今回行っていないシステム全体のシミュレーションを行い、個々のハードウェアに対して最適化を行うことで実用的なものにする。

謝辞

本研究において、懇切なる御指導、御鞭撻を賜った岩田 誠助教授、大森 洋一助手に心より感謝の意を表します。

日頃から御支援、適切な御指導を頂いた細美 俊彦氏に感謝の意を表します。

日頃から御支援を頂いた情報システム工学科岩田研究室の方々、森川 大智氏、橋本 正和氏、森安 亮氏、古家 俊之氏、別役 宣奉氏、假屋 文彦氏に感謝の意を表します。

参考文献

[1] 松澤, 福田, “複数の集約演算のための並列アルゴリズム”, 信学論 D-I,Vol.J82-D-I,No.1,pp.98-110(1999).

[2] H.Terada,SMiyata,and M.Iwata, “DDMP's:self-timed super-pipelined data-driven multimedia processors”,Proc.of IEEE,87(2),pp.282-296(1999).

[3] 福田, 森本, 森下, 徳山, “データマイニングの最新動向”, 情報処理学会誌 Vol.37,pp597-603(1996).

[4] A.Shatdal,J.Naughton, “Adaptive Parallel Aggregation Algorithms”,Proc.of the 1995 ACM-SIGMOD Conference,San Jose,CA,(1995).

[5] 前田, 大森, 岩田 “複数の集約演算に対する並列パイプライン化”,2000 SHIKOKU-SECTION JOINT CONVENTION RECORD THE INSTITUTES OF ELECTRICAL AND RELATED ENGINEERS,pp.143(2000).