データベースシステム 119−66
（1999. 7. 23）

並列 DBMS に於ける動的負荷分散機構の実装

安井 隆宏† 田村 孝之†‡ 小口 正人† 喜連川 優†

分散メモリ型並列データベースシステムは，スケーラビリティに優れており，近年の大規模化しつつあるデータベース処理に非常に適しているといえる．しかしながら，ノード数が増えると，ノード間の負荷に偏りが生じ易くなる．この問題を解決するために，我々は，ライトディープハッシュ多重結合演算の結合演算フェーズにプロセッサ間でハッシュラインの移動を行い負荷の均等化を行う動的負荷分散アルゴリズムを提案し，PC 100台を ATM スイッチで結合した PC クラスタシステムを用い，30ノード規模での実験と検討を行って来た．今回，我々は処理ノード数の増大に対し問題となる主記憶の制限を緩和すべく，ハッシュテーブル及びマイグレーションテーブルを動的に再構築する機構を導入した．本稿では，実験結果から本動的負荷分散手法の100ノード規模の環境における有効性を示す．

Implementation Issues of Dynamic Load Balancing in Parallel DBMS

Takahiro YASUI†, Takayuki TAMURA†‡, Masato OGUCHI†
Masaru KITSUREGAWA†

The scalability of shared nothing architecture makes parallel database systems ideal for handling today's ever growing databases. However, this scalability comes at the cost of increased susceptibility to skew. In order to resolve this problem, we propose a dynamic load balancing algorithm which operates during the join phase of a right-deep hash multi-join executing on a shared nothing system, resolving skew among the processors using hash-lines migration technique. Furthermore, for large-scale data, it is necessary to improve the memory utilization for query execution. Therefor, we introduce dynamic structures namely hash and migration tables which have the capability to improve the query memory usage. The proposed algorithm is implemented on Pentium Pro PC cluster of 100 nodes that connected through ATM switch. The experimental results show the effectiveness of our load balancing algorithm for a cluster of nearly 100 nodes.

1 はじめに

近年，データベースサイズが大規模化する一方で，意思決定支援システム等に見られるような，複雑な問合せ，特に大規模リレーションへの問合せに対する高速な応答への要求が高まっている．しかし，分散メモリ並列計算機環境においては，ノード間の負荷の偏りのために充分な並列効果を得られないことがあり，タスクスケジューリング等の研究が盛んに行われて来た．並列データベースシステムにおいても，データの偏り（スキュー）のためにノード間の負荷にばらつきが生じる事が多い．スキューに関しては，Walton らによって分類がなされている [4] が，この中には並列結合演算処理を始める前に予測することが困難なものもある．これまでにスキューを考慮した並列結合演算アルゴリズムが数多く提案されている [1][3][4] が，並列結合演算を始める前に負荷を予測するというものであり，予測が外れた場合に対する対処などは行われていない．

これに対し，我々は，関係データベースにおける Right-deep 方式の多重結合演算処理についてハッシュラインマイグレーションという動的負荷分散アルゴリズムを提案すると共に，シミュレーションによる評価を行い，その有効性を示してきた [2]．更に，PC クラスタ [7] 上で動作している DBKernel に対し，この動的負荷分散アルゴリズムのプロトタイプを実装し，簡単な問合せを用いた性能評価を行ったところ，30ノード規模の環境では，大幅な性能改善

†東京大学 生産技術研究所
Institute of Industrial Science, University of Tokyo
‡三菱電機（株）情報通信システム開発センター
Information & Communication Systems Development Center, Mitsubishi Electric Corporation

が可能であるという結果が得られた[6]. また, 問合せデータサイズの大規模化に対応すべく, 制御ノードの負荷の一部を問合せ処理ノードへ分散させる方式についても提案し, その有効性を示して来た[5].

今回, 本動的負荷分散手法を 100 ノード規模の環境に適応するためにネックとなっていた主記憶領域の制限を緩和するために, ハッシュテーブル及びマイグレーションテーブルを動的に再構築する機構を検討し, PC クラスタ上に実装を行った. 本稿では, 本動的負荷分散手法が 100 ノード規模の環境においても有効であること実験結果を用いて説明する.

2 Right-deep 結合演算処理の並列実行方式

Right-deep 結合演算 $R_N \bowtie \cdots (R_2 \bowtie (R_1 \bowtie R_P))$ において, R_1 と R_P の結合演算 (Join 1) を行う仮想的な処理単位をステージ 1, R_2 とステージ 1 の結果との結合演算 (Join 2) を行う仮想的な処理単位をステージ 2, Join k $(k \leq N)$を行う仮想的な処理単位を ステージ k で表す. Right-deep 結合演算を並列実行する場合, ステージ 1 をノード $1 \sim k$ $(0 \leq k \leq l)$, ステージ 2 をノード $k+1 \sim \cdots$ の如く, ステージごとに各ノードに処理を割り当てる方式も考えられるが, 本研究では, 並列度が得やすいという点から, 各ステージを全ノード (N ノード) で処理する方式を採用した. N ノード, S ステージで構成された Right-deep 結合演算の並列実行モデルを図 1 に示す.

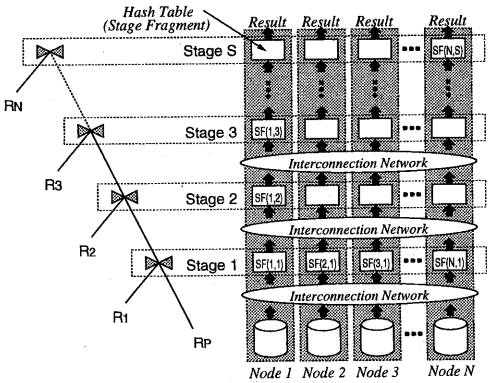


図 1: Right-deep 結合演算実行モデル

次に, このモデルを用いて処理の流れを説明する. 並列実行モデルにおいてもビルドフェーズとプローブフェーズからなり, 各フェーズでは, 以下のように処理を行う.

ビルドフェーズでは, リレーション $R_1, R_2, \cdots, R_N$ をハッシュ値により処理すべきノードに送出し, 各ノードでハッシュテーブルを構築する. 各ステージのハッシュテーブルは N ノードで分割され, 各ノードのメモリ上には, 全ステージのハッシュテーブルが展開される. 各ステージの 1 ノードあたりの処理単位をステージフラグメント (stage fragment) と呼び, ノード i, ステージ j のステージフラグメントを $SF(i,j)$ と表す. このステージフラグメントには, プローブ処理や, タプルの送受信処理など, この結合演算を行う処理全てが含まれる.

各ステージフラグメントにおけるハッシュテーブルの構築が完了すると, プローブフェーズが始まる. リレーション R_P をディスクから読み出し, ハッシュ値により処理すべきノードへ送出する. 各ステージフラグメントでは, 入力タプルを用いてプローブを行い, 結合属性が条件に一致した場合には結果タプルを生成し, その結果タプルを次ステージへ送出する. このように, 全体としてパイプライン方式で処理を行う.

3 動的負荷分散機構

ノード間における負荷の偏りを解消するため, ハッシュラインマイグレーションという技法を用いて動的に負荷分散を行う. これは負荷の高いノードから負荷の低いノードへハッシュラインをマイグレートすることにより, ハッシュラインにかかる負荷を他ノードへ移動させるというものである. この技法を用いると, 結合演算実行中に動的に負荷分散を行うことが可能になる. 本節では, 動的負荷分散機構について説明する.

3.1 ハッシュラインマイグレーション機構

負荷の偏りを解消するために, ハッシュラインマイグレーション (Hash Line Migration) というハッシュラインを単位とした負荷分散機構を導入する. ハッシュラインマイグレーションとは, あるステージフラグメントのハッシュテーブルから, 同一ステージの他のステージフラグメントのハッシュテーブルへハッシュラインを移動 (マイグレート) するという技法である. 負荷が偏る原因として, 入力タプル数がノード間で偏る PS (Probe Skew) や, 結合率が偏る JPS (Join Product Skew) が挙げられるが, ハッシュラインを他ノードへマイグレートすると, そのハッシュラインにかかる負荷も他ノードへ移動するため, 負荷の高いノードから負荷の低いノード

へハッシュラインをマイグレートすることによりステージ内の負荷を均等化することができる．このように，ステージ単位で負荷分散を行う．ハッシュラインのマイグレーションが完了すると，ハッシュラインの移動先をマイグレーションテーブルに登録する．

ハッシュラインマイグレーションの概念図を 図2に示す．ノード 1, 2 の負荷 (Load) がそれぞれ 600, 400 であり，ノード 1 に負荷が偏っているため，負荷が 100 であるハッシュラインをノード 2 にマイグレートすることにより，負荷を均等にすることができる．

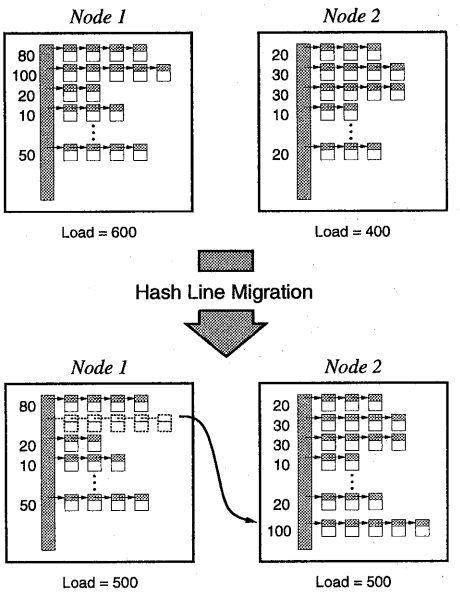


図 2: ハッシュラインマイグレーション

3.2 各ノードの統計情報

負荷の算出に必要な情報を，結合演算の実行時に統計情報として保持する．統計情報には，ステージフラグメント $SF(i,j)$ に対する統計情報と，ハッシュライン $HL(i,j,k)$ に対する統計情報があり，それぞれ対象は違うものの内容的には同じ統計を取る．必要とする統計情報には以下のものがある．

ステージフラグメント $SF(i,j)$ の統計情報

- 入力タプル数：$N_{recv}(i,j)$
- 比較回数：$N_{comp}(i,j)$
- 出力タプル数：$N_{send}(i,j)$

ハッシュライン $HL(i,j,k)$ の統計情報

- 入力タプル数：$N_{recv}(i,j,k)$
- 比較回数：$N_{comp}(i,j,k)$
- 出力タプル数：$N_{send}(i,j,k)$

負荷の偏りの時間的な変化への対応を考慮し，負荷の計算には，直前の一定時間に収集した統計情報を使用する．収集する負荷には以下のものがある．

3.3 ステージフラグメントの負荷

ステージフラグメント $SF_{i,j}$ の負荷 $L_{SF}(i,j)$ を

$$L_{SF}(i,j) = N_{recv}(i,j) \times T_{recv} + N_{comp}(i,j) \times T_{comp} + N_{send}(i,j) \times T_{send} \quad (1)$$

と定義する．$T_{recv}, T_{send}, T_{comp}$ は，それぞれ1タプルあたりの受信時間，結果生成時間，送信時間，及び，1回の比較時間であり，測定値を元に決定される．

3.4 ハッシュラインの負荷

ステージフラグメント $SF(i,j)$ における k 番目ハッシュラインの負荷 $L_{HL}(i,j,k)$ をステージフラグメントの負荷と同様に以下の式で算出する．

$$L_{HL}(i,j,k) = N_{recv}(i,j,k) \times T_{recv} + N_{comp}(i,j,k) \times T_{comp} + N_{send}(i,j,k) \times T_{send} \quad (2)$$

3.5 負荷の偏りの判定

本アルゴリズムでは，各ステージ毎に負荷の判定を行う．ステージ j の負荷の偏りの判定を以下の式によって行う．

$$skew_j = \frac{max_i L_{SF}(i,j) - min_i L_{SF}(i,j)}{\sum_{i=1}^{N} L_{SF}(i,j)/N}$$
$$skew_j > \alpha : \text{ステージ}j\text{の負荷は不均一} \quad (\alpha : \text{閾値}) \quad (3)$$

3.6 動的負荷分散の流れ

各ノードの負荷の管理のために，問合せ処理ノードとは別の制御ノード (フォアマン) を用意し，前述したハッシュラインマイグレーション技法を用いる

ことにより, 動的に負荷分散を行う. 負荷分散は一定のインターバルごとに繰り返される. 動的負荷分散の処理の大まかな流れは以下のようになる.

(a) ステージフラグメントの負荷情報の収集

フォアマンが負荷分散開始のメッセージを全ノードに発行すると, 各問合せ処理ノードは, ステージフラグメントの統計情報から負荷の算出を行い, その負荷情報をフォアマンに送信する.

(b) 負荷偏りの判定

ステージ毎に負荷の偏りの判定を行い, 偏りが検出された場合には, (c) 以降の処理を行う. 偏りが検出されない場合には, (g) の処理に飛ぶ.

(c) ハッシュラインの統計情報の収集

フォアマンは, 偏りの検出されたステージについて, 負荷の高いステージフラグメントのハッシュラインに関する負荷情報を収集する.

(d) マイグレーションのプランニング

フォアマンは, 収集した負荷情報を元にステージフラグメントの負荷が均等になるように, ハッシュラインのマイグレーションプランを立てる.

(e) マイグレーションの実行

各ノードはマイグレーションプランに従ってハッシュラインのマイグレーションを行う.

(f) マイグレーションテーブルの更新

全てのノードでマイグレーションが完了すると, マイグレーションテーブルにハッシュラインの移動先を登録する.

(g) 統計情報のリセット

各ステージフラグメントが保持している統計情報を 0 にリセットする.

4 ハッシュテーブル及びマイグレーションテーブルの動的再構築

我々はこれまで, 30 ノード規模という小規模な環境でハッシュラインマイグレーションを用いた動的負荷分散機構の有効性を模索してきた. 問合せのサイズについても, より大きなものに対応すべく, フォアマンの負荷を問合せ処理ノードへ分散させる手法についても検討を加えて来た.

ところが, 更に大規模な問合せを高速に処理するためには処理ノード数を増加させる必要があるが, 従来の実装方式では, 主記憶領域という制約が大きかったため実現が難しかった. そこで, 主記憶の利用効率を高めるため, ハッシュテーブル及びマイグレーションテーブルを動的に再構築する機構を導入した.

ハッシュラインマイグレーション実行時には, 移動先のノードの主記憶上にマイグレートされたハッシュラインを保持するためのハッシュテーブルの構築が必要であるがこれを動的に行えるようにし, 保持するハッシュライン数が 0 になった場合には, ハッシュテーブルの開放を行うようにした. 更に, マイグレーションテーブルについても同様に構築, 開放を行えるようにした.

5 性能評価

前述した負荷情報収集方式を並列 DBMS DBKernel に実装し, 簡単な問合せを用いて性能評価を行った. 本節では, 実行トレースを用いて, 100 ノード規模の環境においても本動的負荷分散機構が有効であることを示す.

5.1 実験環境

本負荷分散手法の実装には, 我々の研究室で構築した 大規模 PC クラスタ を使用した. この大規模 PC クラスタは, コモディティハードウエアを用いた超並列データベースサーバの実現可能性及び有効性を示すことを目的として構築されたシステムである. 各ノードには, コストパフォーマンスに優れた PC (CPU: Pentium Pro 200MHz, Memory: 64 Mbyte) を採用しており, その PC 100 台を 155 Mbps の ATM および 10 Mbps イーサネットの 2 系統のネットワークで相互結合したシステムである (図 3). オペレーティングシステムには Solaris 2.5.1(for x86) を採用し, 並列 DBMS として我々の研究室で構築した DBKernel が実装されている. この DBKernel に対し動的負荷分散機構の実装を行った.

性能評価に用いた問合せは, $R_2 \bowtie (R_1 \bowtie R_P)$ と表記される 2 重結合演算である. 問合せで用いるデータについては, 表 1 に示すようにステージフラグメントあたりのハッシュラインが 10,000 本, プローブタプル数が 3,000,000 個である. また, この

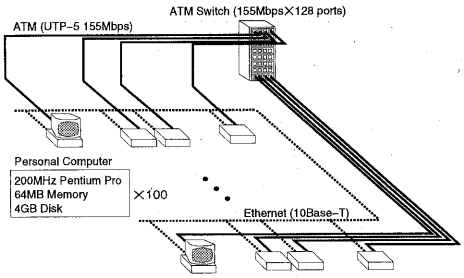


図 3: 大規模 PC クラスタの構成図

ノード数	96 [node]
ステージ数	2 [stage]
タプルサイズ	32 [byte/tuple]
プローブタプル数	3,000,000 [tuples/node]
ハッシュライン数	10,000 [lines/node]
負荷情報収集インターバル	28 [sec]
問合せの結合率	100 [%]

表 1: 実験環境

データには, 入力タプル数および結合率をノード間で偏らせることによる人為的なスキューを導入した. この条件のもと 96 ノードを用いて実験を行った.

6 性能評価

今回, 表1 に示すような 3 種類のスキューに対し測定を行った. *skew*1 では, ノード 0 のプローブ数(= 入力タプル数) が他のノードに比べ 2 倍あり, さらに結合率も 2 倍となっている. その結果, 出力タプル数は他のノードの 4 倍となっている. 表1 からは3 種類のスキューのいずれにおいても負荷分散を行う事により, 実行時間が大幅に短縮されていることがわかる.

*skew*2 の実行トレースとして, 負荷の高いノード 0 とその他のノードの代表としてノード 1 を示した (図 4, 5). 負荷分散を行わない場合には, 終始, ノード 0 の負荷が高く, ノード 0 のデータをディスクから読み出せない状態であり, 他のノードのデータの処理を終った後に (335 [*sec*]), やっと読み出す事ができるようになる. 負荷分散を行った場合, 1 回目の負荷分散 (#1) 後には, 負荷が均等となり, 処理がスムーズに行えるようになっていることが分かる. 今回の実験では, 負荷情報収集インターバルを 28 [*sec*] したため, 5 回の負荷情報収集が行われ, その各々においてハッシュラインマイグレーションが行われた. 1 回目の負荷情報収集時にはスキューは 4.51 であったが, 2 回目以降は, それぞれ 0.45, 0.5, 0.43, 0.28 となり偏りが少なくなったことも確認された.

7 おわりに

ハッシュラインマイグレーション技法を利用した動的負荷分散方式において, 主記憶の利用効率を高めるべく, ハッシュテーブル及びマイグレーションテーブルを動的に構築する機構を実装した. その結果, 100 ノード規模の環境を用いた問合せ処理に対して, 本動的負荷分散手法を適用することが可能となった. 実験結果からは, 本手法を適用することによる大幅な性能向上が確認された.

今後, 多数段の結合演算を用いた測定や複雑なスキュー分布に対する測定を行い, より詳細な考察を行う予定である.

参考文献

[1] M.S. Chen, M.L. Lo, P.S. Yu, and H.C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proc. of Int'l Conference on Very Large Data Bases*, 18th, 1992.

[2] S. Davis and M. Kitsuregawa. Simulation study of a runtime load balancing algorithm for pipelined hash multi-joins. 電子情報通信学会 データ工学研究会, volume 96, pp. 13–18, 1997.

[3] D.J. DeWitt, J. Naughton, D.A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of Int'l Conference on Very Large Data Bases*, 18th, 1992.

[4] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of Int'l Conference on Very Large Data Bases*, 17th, 1991.

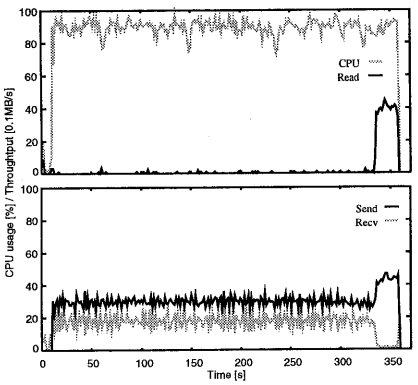
[5] 安井, 田村, 小口, 喜連川. 並列 dbms における動的負荷分散機構： 負荷情報収集に関する一考察. Number 57 in 98, pp. 87–94. 情報処理学会, 1998.

[6] 安井, 田村, 小口, 喜連川. 並列関係データベース処理システムに於ける動的負荷分散機構に関する一考察. 電子情報通信学会 第 9 回データ工学ワークショップ, 1998.

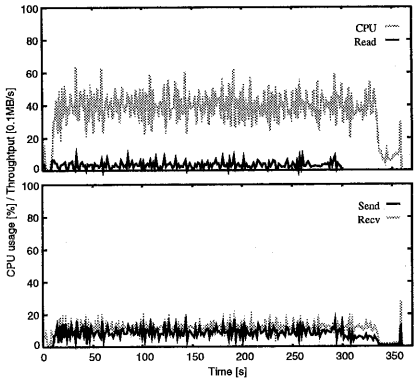
[7] 田村, 小口, 喜連川. 大規模 PC クラスタにおける並列関係問合せ実行方式とその 評価. Number 416 in 97, pp. 63–68. 電子情報通信学会人工知能と知識処理研究会, 1997.

		$skew1$	$skew2$	$skew3$
プローブ数の比	Stage 1	2 : 1 : ... 1	3 : 1 : ... 1	2 : 2 : 1 : ... 1
	Stage 2	1 : 1 : ... 1	1 : 1 : ... 1	1 : 1 : 1 : ... 1
結合率の比	Stage 1	2 : 1 : ... 1	3 : 1 : ... 1	2 : 2 : 1 : ... 1
	Stage 2	1 : 1 : ... 1	1 : 1 : ... 1	1 : 1 : 1 : ... 1
実行時間 [sec]	負荷分散無し	246.01	357.79	263.89
	負荷分散有り	169.88	178.15	185.76

表 2: 実験結果

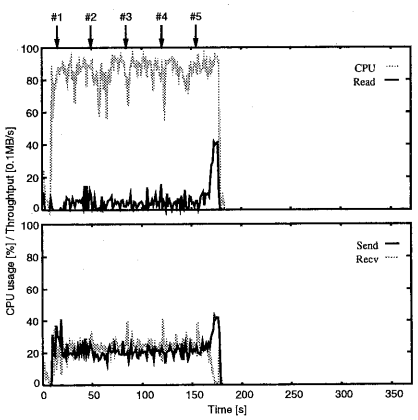


[Node 0]

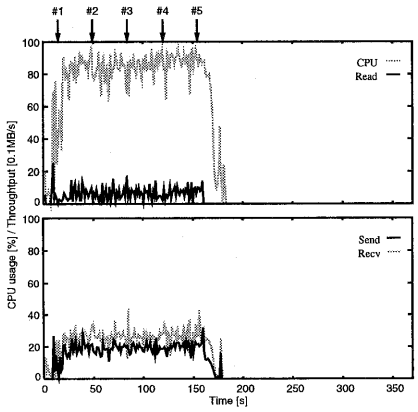


[Node 1]

図 4: 負荷分散無しの実行トレース ($skew2$)



[Node 0]



[Node 1]

図 5: 負荷分散有りの実行トレース ($skew2$)