データベースシステム 116－12
（1998. 7. 8）

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

安井 隆宏† 田村 孝之†‡ 小口 正人† 喜連川 優†
†東京大学 生産技術研究所
‡三菱電機 (株) 情報通信システム開発センター

概要

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 クラスタシステムへの実装を行ってきた．本稿では，動的負荷分散機構における現在の負荷の管理方式にについて見直し，管理ノードの負荷を分散させる方式を提案する．そして，実験結果からその有効性を示す．

キーワード 負荷分散，並列問合せ処理，ハッシュラインマイグレーション

Implementation Issues about Method of Gathering Load Information for Dynamic Load Balancing in Parallel DBMS

Takahiro YASUI†, Takayuki TAMURA‡, Masato OGUCHI†
Masaru KITSUREGAWA†
†Institute of Industrial Science, University of Tokyo
‡Information & Communication Systems Development Center,
Mitsubishi Electric Corporation

Abstract

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. We implemented the proposed scheme on the PC cluster system, which consists of 100 node Pentium Pro PCs connected through ATM switch. In this paper, we describe a problem in centralized algorithm which we selected first, and propose strategy which resolves them and can improve the performance. At last, this strategy is an effective from experimentation results.

Keywords load balancing, parallel query processing, hash line migration

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 [4][3][1] 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 [6] 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, 偏りが大きい場合には大幅な性能改善が可能であるという結果が得られた [5].

現在, 本アルゴリズムのチューニングを行っている. 本アルゴリズムの実装面における問題の1つとして, フォアマンと呼ぶ制御ノードによる負荷情報の集中管理が上げられる. 問合せのデータサイズの大規模化と共に, フォアマンの負荷が高くなり, ハッシュラインマイグレーションのスケジューリング時間が増加し, 負荷の偏りに対する反応が遅れる傾向にある. 本稿では, 負荷情報収集方式について見直し, フォアマン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検討すると共に, 新しい方式を提案し, PC クラスタ実験機実装することにより, 従来方式に比べて高い性能向上が得られることを明らかにする.

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 〜 k $(0 \leq k \leq l)$, ステージ2をノード $k+1$ 〜 $\cdots$ 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.

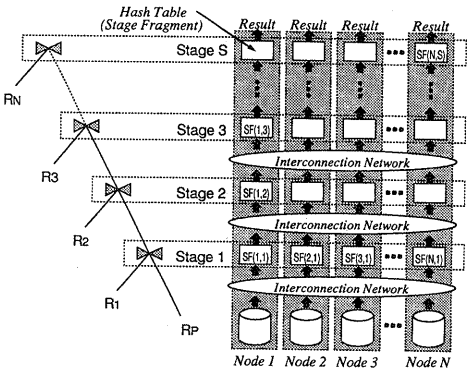


図 1: Right-deep 結合演算実行モデル

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．

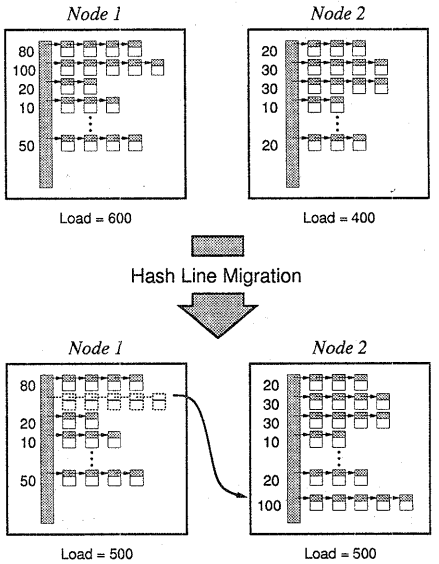


図 2: ハッシュラインマイグレーション

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，前に測定しておく．

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．

$$skew_j = \frac{max_i L_{SF}(i,j) - min_i L_{SF}(i,j)}{\sum_{i=1}^{N} L_{SF}(i,j)/N}$$

$skew_j > \alpha$：ステージjの負荷は不均一 (α：閾値) (3)

3.6 動的負荷分散の流れ

各ノードの負荷をフォアマンと呼ぶ別の制御ノードにより集中管理し，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

フォアマンは，負荷分散開始のメッセージを全ノードに送付し，各ノードからステージフラグメントの負荷情報を収集する．

(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，各ノードから偏りを持つステージのハッシュラインの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，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 フォアマンへの負荷集中に対する問題

前節で述べた負荷分散アルゴリズムに関し, 本節では, まず, 従来のフォアマンによる負荷情報収集フェーズについて説明し, フォアマンへの負荷の集中による問題点を取り上げ, 続いてその解決手法について説明する.

4.1 従来の負荷情報の収集方式

フォアマンは, ステージフラグメントの負荷情報を収集する際に, 各ノードが保持している統計情報をそのまま受取り, 式 (1) により負荷の算出を行う. そして, 式 (3) により負荷の偏りの判定を行う. この時, 偏りが検出されると, 偏りのあるステージからハッシュラインの統計情報を収集し, 式 (2) を用いてハッシュラインの負荷の算出を行う. マイグレーションのプランニングの際には, 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. このフェーズでは, 負荷の大小でハッシュラインをソートし, 負荷の大きいハッシュラインを優先的にマイグレーションプランに加えるという方針のもとで行い, 比較的, 特定のハッシュラインに負荷が集中するような場合にも対応できるようにした.

4.2 従来の負荷情報収集方式における問題点

前述したように, フォアマンという単一の制御ノードにより負荷の集中管理を行うと, 対象とする問合せで処理するデータ量が大きくなるにつれて, フォアマンの負荷が高くなり, ハッシュラインのマイグレーションプランニング時間に影響を与える. 負荷の偏り検出時には, 可能な限り迅速な対処が望ましいが, スケジューリング時間が長くなると対応が遅れることになる. 本アルゴリズムは比較的小さな問合せには効果的であるが, ビルド側のリレーションサイズが大きくなるにつれて, フォアマンにかかる負荷が高くなり, 十分な性能が得られなくなる.

4.3 負荷情報収集方式の改良

今回, 我々は, フォアマンと各処理ノード間での負荷情報の転送量の削減とフォアマンにおける処理のスケジューリング時間も削減を行うために, 負荷情報の収集方式に関し詳細な検討を行った. フォアマンの負荷を分散させるために, 各ノードは, ステージフラグメントとハッシュラインの両方において統計情報から負荷を算出し, フォアマンは負荷の形で情報の収集を行うようにした. フォアマンはステージフラグメントの負荷情報を収集し, 偏りの判定を行い, 偏りを検出した場合には, そのステージの負荷の平均値を上回るステージフラグメントに対してからのみハッシュラインの負荷情報を収集する. 負荷の高いステージフラグメントでは, 各ハッシュラインに対し負荷の算出が行われるが, 更に, 負荷の高いハッシュラインについてのみファアマンに送信する (図 3).

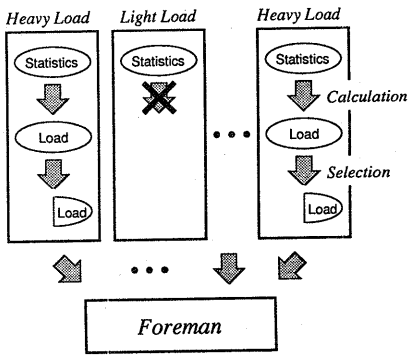


図 3: 負荷情報収集モデル

フォアマンは, ハッシュラインの負荷情報収集後, ハッシュラインのマイグレーションプランニングを行う. この時, ハッシュラインの負荷の大きさによるソートは行わない. ソートは負荷の高い処理であり, ハッシュラインのソート時間がマイグレーションプランニング時間に占める割合は高かったが, 各ノードで負荷の高いハッシュ

ラインを選択し, フォアマンに送信することにより, フォアマンの負荷を軽減することが可能となる.

5 性能評価

前述した負荷情報収集方式の実装を行い簡単な問合せを用いて性能評価を行った. 本節では, 新旧の負荷情報収集方式の結果を比較し, 実際に性能が向上することを示す.

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 4). 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. 本研究ではこの DBKernel に対し負荷分散機構の実装を行い, 性能評価を行った.

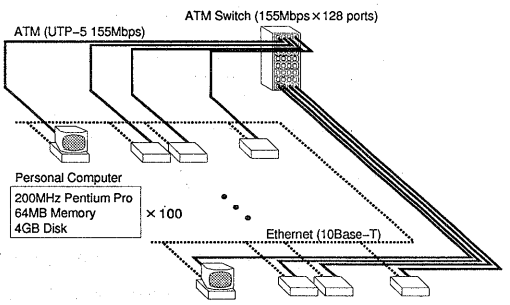


図 4: 大規模 PC クラスタの構成図

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, 1 ステージフラグメントあたりのハッシュラインが 5000 本, プローブタプル数が 500,000 個である. また, このデータには人為的なスキューを導入し, ステージ 1 全体の 25% の負荷を $SF(1,1)$ に割り当て, その他のノードの負荷は均等になるようにした. この条件のもと 30 ノードを用いて実験を行った (表 1).

ノード数	30 [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	500,000 [tuples/node]
ハッシュライン数	5000 [lines/node]
負荷の偏り	$SF(1,1)$ に高負荷 (25%)

表 1: 実験環境

5.2 負荷情報収集時の処理時間

ステージフラグメント情報の収集時間, ハッシュライン情報の収集時間, マイグレーションプラニング時間という 3 種類の処理時間時間について負荷情報収集方式の変更前後の 2 通りで測定を行い, 表 2 の結果を得た. 変更前では, ハッシュライン情報の収集時間とマイグレーションプランニング時間が非常に長いことがわかる. マイグレーションプランニング時間にはハッシュラインのソートの時間が大部分を占めている. この結果より, 負荷分散の処理フェーズが始まってから約 14$[sec]$ は, 負荷が偏った状態で処理が行われ, ハッシュラインのマイグレーションはその後に行われることになる. その結果, 本負荷分散アルゴリズムの十分な効果を得られないことになる.

一方, 前述した負荷情報収集方式の改良を行った時の結果と比較すると, ハッシュライン情報の収集時間およびマイグレーションプランニング時間が大幅に短縮でき, 全体として 1 $[sec]$ 弱でマイグレーションを始めることが可能になっていることが分かる. この大幅な処理時間の短縮の要因の一つに, 集計情報量の削減が上げられる. 特に, この問合せにおいては, ハッシュラインの負荷情報の収集に関しては, データサイズが 30 分の 1 になっている (図 3). ステージフラグメント情報の収集時間では, 変更前後の値が等しいが,

ステージフラグメントのデータ量が小さく, 圧縮の効果も低いためである. 実際には, 480 [*byte*] が 120 [*byte*] となっている.

5.3 実行トレースを用いた解析

負荷情報収集方式の変更前後の両方について, 問合せ処理の実行トレース (図5) を用いて, 振舞いの違いについて解析を加える. 図5では, 負荷の高いノードとして Node 1, 負荷の低いノードから Node 2 をピックアップし, 各トレースでは, CPU の利用率, Disk のスループット, ネットワークの送受信のスループットについての時間変化を示した. このトレースからハッシュラインマイグレーション後には, 負荷情報収集方式の変更前後のいずれも負荷が均等になるっており, 負荷分散による効果は得られているが, ハッシュラインマイグレーションの実行開始時刻が大きく異なる. 前述したように, 負荷情報収集方式の変更前では, フォアマンへの過負荷から実行開始から約22 [*sec*] の時点での実行となる. そのため, 負荷情報収集方式の変更後よりも長い間負荷が偏った状態となり, 処理効率が低下してしまう. このように, 負荷情報収集方式の変更により, ハッシュテーブルサイズが大きい場合にも, 処理効率を落とすことなく負荷分散手法を適用できることが分かる.

6 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, フォアマンへの負荷集中によるマイグレーションプランニングの処理時間の増大に対処すべく, 負荷情報収集方式に関し検討を進めた. 各統計情報を用いた負荷の計算を問合せ処理ノードに分散することにより, フォアマンの負荷を減らし, また, 必要度の高い負荷情報のみを収集することにより, 処理するデータ量を減少させる手法を提案し, 実装後, 性能評価を行った結果, 処理時間が大幅に短縮され, 大きな性能向上が得られた.

今後, 多数段の結合演算を用いた測定および評価, そして, 100 ノード規模での性能評価についても行う予定である.

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.

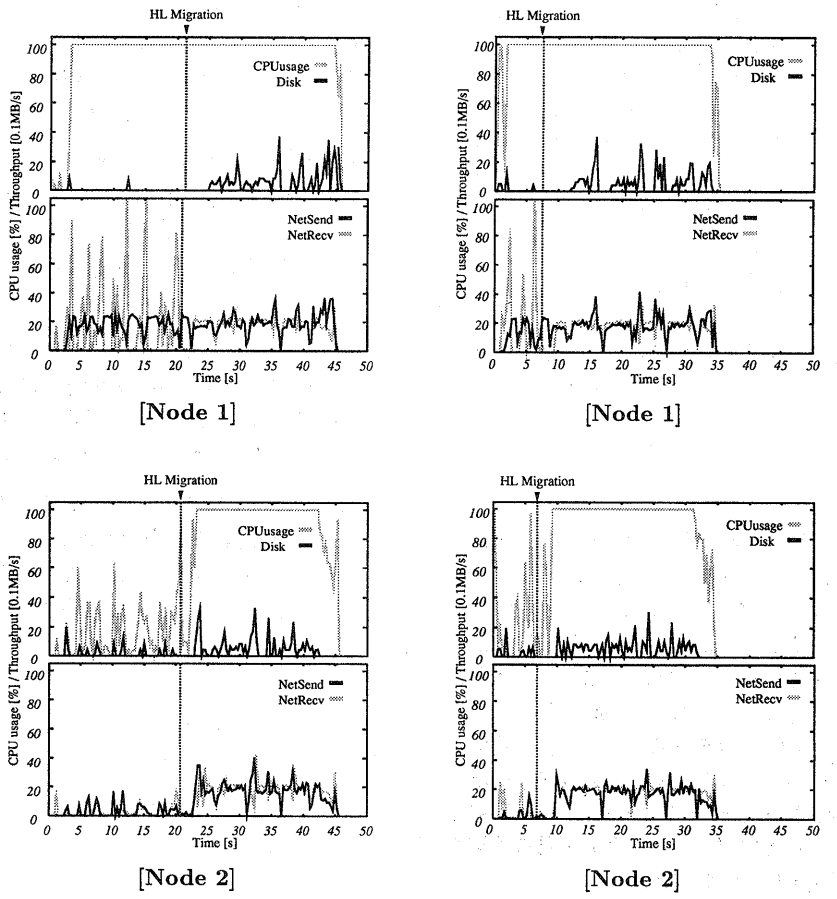
[6] 田村, 小口, 喜連川. 大規模 PC クラスタにおける並列関係問合せ実行方式とその 評価. Number 416 in 97, pp. 63–68. 電子情報通信学会人工知能と知識処理研究会, 1997.

内容	変更前 [sec]	変更後 [sec]
ステージフラグメント情報の収集時間	0.01	0.01
ハッシュライン情報の収集時間	6.55	0.53
マイグレーションプラニング時間	8.83	0.02
問合せ全体の処理時間	45.83	33.82

表 2: 負荷収集方式改良前後における処理時間

内容	変更前 [byte]	変更後 [byte]
ステージフラグメント情報の収集データ量	2,400,000	80,000
ハッシュライン情報の収集データ量	480	120

表 3: 負荷収集方式改良前後における転送データ量



(a) 集中型負荷情報収集方式　　(b) 分散型負荷情報収集方式

図 5: 問合せ処理の実行トレース