Vol.2010-DBS-151 No.8
2010/11/12

クラウド環境でのストリーミングアプリケーション向け動的資源配置手法

秋岡明香[†1]　加藤慶一[†1]
村岡洋一[†1]　山名早人[†1]

時系列に沿って短時間間隔でデータが到着する入力データストリームに対して，リアルタイムに詳細な解析を行なうストリーミングアプリケーションの重要性が高まっている．本稿では，こうしたストリーミングアプリケーションを並列化し，動的にクラウドなどの広域分散環境に配置する手法を提案する．シミュレーションにより従来通りにストリーミングアプリケーションを処理した場合と提案手法によりストリーミングアプリケーションを並列処理した場合を比較したところ，現実的なアプリケーションを想定した場合には従来通りの実行方法では処理が間に合わず，データを破棄する必要があることを示した．また，提案手法を用いることで，全てのデータを破棄することが可能であることを示し，そのオーバーヘッドは 26%であった．

Dynamic Resource Allocation for Streaming Applications in Cloud Environment

Sayaka Akioka,[†1] Norikazu Kato,[†1]
Yoichi Muraoka[†1] and Hayato Yamana [†1]

Streaming application, which requires to process data frequently arrives in chronological order, is now a center of interest. This paper proposes a methodology to parallelize, and dynamically allocate streaming applications over distributed environment such as cloud computing environment. The simulation results approved that practical streaming applications need to be processed in parallel in order to avoid loss of data for lack of processing time. However, the methodology proposed in this paper enables all the input data processed with 26% overhead of average execution time of each block of input data.

1. 研究の背景

近年では，アプリケーションが処理すべきデータ量は爆発的に増加し，こうしたデータを詳細かつリアルタイムに処理することが求められる状況が増えてきた．こうした大量のデータを処理するアプリケーションの代表例としてストリーミングアプリケーションがあり，ストリーミングアプリケーションを効率よく処理するアルゴリズムとして，ストリーミングアルゴリズムの提案が増えている．

ストリーミングアプリケーションとは，監視カメラの動画データなど，時系列に沿って次々と到着するデータストリームを滞りなく処理することが必要なアプリケーションである．従来より，科学技術計算のように大量のデータ処理を必要とするデータインテンシブアプリケーションをグリッドやクラウド環境で効率的に実行するための研究は，多く行なわれてきた[1)–6)]．しかしストリーミングアプリケーションは，データインテンシブアプリケーションとデータアクセスパターンが大きく異なるため，これらの成果を用いることは難しい．

データインテンシブアプリケーションにおけるデータアクセスパターンは，write-once-read-many 型である[2)]．つまり，計算に必要なデータは計算環境に保存されており，各計算ノードはその一部を繰り返し参照しながら計算を進める．したがって，各計算ノードが必要とするデータを，該当計算ノードからのアクセス効率の良い場所に移動またはコピーすることがアプリケーションの高速実行の鍵となる．

一方ストリーミングアプリケーションでは，入力データは時系列に沿って連続的に到着し，到着したデータをパイプライン的に処理するモデルが主流である．したがって，データアクセスパターンとしては，到着したデータに対する処理を到着した時点で行ない，次データが到着すれば，処理済みの過去のデータに後からアクセスすることはない．このことから，データのキャッシュ等がアプリケーションの実行性能を大きく左右することはないため，従来のデータインテンシブアプリケーション向け資源配置は有効ではない．ストリーミングアプリケーションにおいてアプリケーション性能を向上させるためには，データストリームを滞りなく流すための計算ノードを選択することと，新規データが到着するまでに前データの処理を完了するために充分な計算資源を，パイプラインの各ステージに割り当てることが重要である[7)]．

†1 早稲田大学
Waseda University

© 2010 Information Processing Society of Japan

こうした背景から，ストリーミングアプリケーションに特化した，広域分散環境での計算資源割り当てに関する研究は増えてきた[7),9),10)]．これらの手法は，静的手法であったり，パイプラインにおける各ステージの計算コストを事前に見積もっておく必要があったりする．しかし，実際のストリーミングアプリケーションでは，システムログ監視のように，データストリームのデータ量が一定でなく，パイプラインにおける各ステージの計算コストを充分に正確に見積もることが難しい場合が多い．さらには，クラウド環境のような広域分散環境では，計算資源の負荷は一定ではないため，資源配置を決定した時点での最適解が保証されるとは限らない．そこで本稿では，入力データストリームの変化や，計算環境の負荷変動へ柔軟に対応可能な，ストリーミングアプリケーション向けの動的資源配置手法を提案する．

本稿の構成は以下の通りである．第 2 章では，広域分散環境におけるストリーミングアプリケーションの資源配置手法に関する関連研究を紹介する．第 3 章では，本稿で提案するストリーミングアプリケーション向け動的資源配置手法の詳細について述べる．第 4 章では，提案する動的資源配置手法と，第 2 章で紹介した代表的なストリーミングアプリケーションの資源配置手法をシミュレーションにより比較評価する．第 5 章でまとめと今後の課題について述べる．

2. 関 連 研 究

Agarwalla らは，グリッド環境でのストリーミングアプリケーションを対象としたスケジューラである Streamline を提案した[7)]．Streamline は，事前に対象アプリケーションのパイプライン全体，および各ステージの計算コスト，データ入力コスト，データ出力コストを把握しておき，これらのコストで重み付けをした優先度により作成したリストに基づき，リストスケジューリングを行なう．その後，同一アプリケーションのステージは同一計算ノードに割り当てる，アプリケーションが要求するシステム要件（GPU の有無など）を満たす計算資源を選択する，などの条件を満たすように，計算資源を割り当てて行く．またこの時，Network Weather Service[8)] からの情報を元に，ネットワーク帯域も考慮した計算資源割り当てを行なう．Streamline では，パイプラインの計算コスト見積もり，およびデータ入出力コストを正確に把握することがスケジューリング性能を大きく左右する．しかし，実際のアプリケーションにおいては，これらを充分正確に予測することは非常に難しい．特に，入力データは実アプリケーションでは変動する場合が多く，これを正しくモデル化し，コストを見積もることは困難である．

Zhang らは，グリッド環境でのストリーミングアプリケーション向けに，利用するプロセッサ数，ネットワーク帯域幅，ハードディスク容量を指定してジョブ投入が可能なシステムを提案した[9)]．待ち行列に投入されたアプリケーションは，それぞれの要求プロセッサ数，ネットワーク帯域幅，ハードディスク容量と待ち時間の長さに応じて，FCFS 方式で計算資源を割り当てられる．Zhang らのジョブシステムは，Streamline 同様，アプリケーションの要求条件を充分に正確に見積もることが要求されるが，実運用ではこれは非常に難しい．

Chen らは，グリッド環境でストリーミングアプリケーションに静的資源割り当てを行なう GATES を提案した[10)]．GATES は，計算資源をネットワーク帯域幅を絶対値とする負数により重み付けしたグラフにより表現し，各データ入力源に対して最小全域木を求め，この最小全域木にタスクグラフをマッピングする．これにより，ネットワークコストを最小化する資源配置が実現可能となる．また，データ入力源の数が n，パイプラインが m 段，利用可能な計算ノードの数が k（$k > m$）である場合，全探索により資源配置を決定する場合の計算時間の下限は $\Omega(n^n)$ であるが，GATES では $O(nk^2)$ である．しかし，GATES のアルゴリズムは静的であることの他に，パイプラインの最終段では全てのデータストリームが集約されること，計算コストはアプリケーション性能のボトルネックとはならないこと，などの前提条件が課されている．こうした前提条件は，実アプリケーションの要求には沿っておらず，GATES の効果を充分に得る事ができる場面は極めて限られる．

また，これらの関連研究は，すべてアプリケーション単位でのタスクグラフに対する計算資源の割り当て手法の提案であり，本稿で提案するような単一アプリケーションを並列化する提案はない．

3. 提 案 手 法

提案する資源配置手法の特徴は以下の通りであり，3.1，3.2 および 3.3 節ではそれぞれの詳細を説明する．

(1) データストリームを滞り無く流すための時間制約条件を考え，この制約条件を満たすように資源割り当てを行なう．
(2) データの入力コストと比較して特定ステージの計算コストが高い場合，入力データストリームを複製し，計算コストが高い処理を並列化することで，入力データストリームの取りこぼしを防ぐ．
(3) 初期配置決定後もデータストリームや計算資源の監視を続け，ストリーミング処理の遅延や停止が発生した場合には動的に計算資源の再配置を行なう．

© 2010 Information Processing Society of Japan

Vol.2010-DBS-151 No.8
2010/11/12

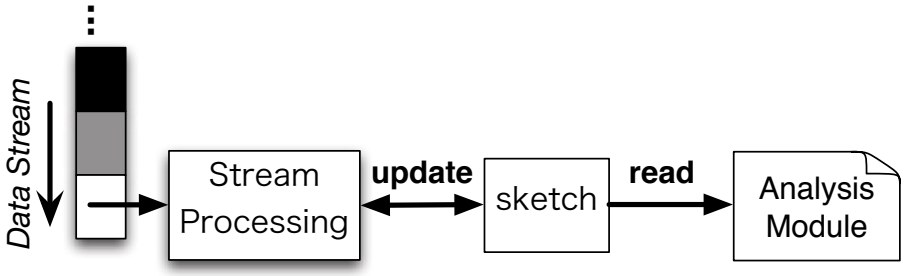

図 1 ストリーミング処理モデル

3.1 時間的制約条件

図 1 に，一般的なストリーミング処理のモデルを示す．データストリームを処理することに特化したストリーミング処理では，計算の途中経過を保存するスケッチを持つことで，計算処理によりデータ入力処理が滞ることを防ぐのが一般的である．本稿では，データストリームからデータフレームを受け取り，スケッチを更新するまでの部分をストリーム処理部（以下，SP），スケッチの内容を参照し，さらなる計算処理を行なう部分を解析モジュール（以下，AM）と呼ぶ．

また図 2 に，連続するデータストリームをパイプライン処理する場合のモデルを示す．図 2 においては，ひとつのデータフレームを処理するパイプラインは横方向で，データストリームの流れは縦方向で示してある．ここで，一般にストリーミングアルゴリズムでは，スケッチの内容をインクリメンタルにアップデートすることで計算を進める．したがって，d_{n+1} を処理中の計算ノードは，d_n を処理中の計算ノードのスケッチを参照する必要がある．したがって，全ての処理を単一計算ノードで行なうことを前提とすると，データ入力を滞らせず，かつ計算結果を正しく維持するためには，SP によるスケッチのアップデートは，必ず AM による前フレームのデータを解析するためのスケッチ読み出し以降に行なわれる必要があることが分かる．

そこで，n 番目のデータフレーム d_n に対する SP における計算時間を P^n_{SP}，AM における計算時間を P^n_{AM}，SP がデータストリームから d_n を読み出すために必要な時間を R^n，SP がスケッチをアップデートするために必要な時間を U^n_{SP}，AM がスケッチからデータを読み出すために必要な時間を R^n_{AM}，d_n が到着してから d_{n+1} が到着するまでの時間間隔を Δt^{n+1}_n として，データ入力およびパイプライン処理を滞らせないための時間制約を考える．

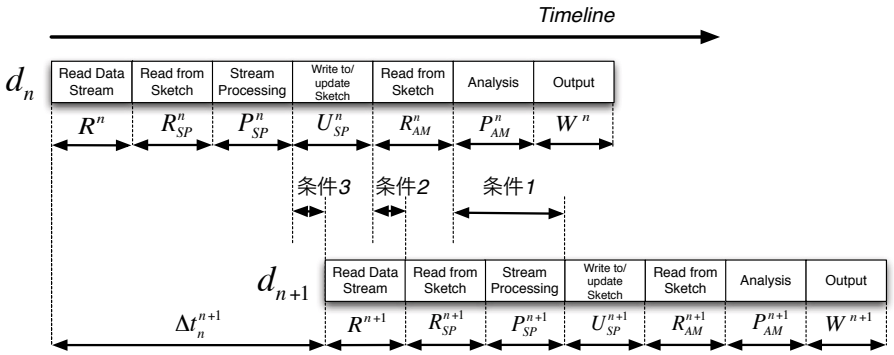

図 2 ストリーミング処理のパイプライン化

d_{n+1} の処理について，SP によるスケッチアップデートは必ず d_n の処理における AM によるスケッチからの読み出しの後でなくてはならないことから，以下の制約条件が成り立たねばならない（図 2 中の条件 1）．

$$R^n + R^n_{SP} + P^n_{SP} + U^n_{SP} + R^n_{AM} \leq \Delta t^{n+1}_n + R^{n+1} + R^{n+1}_{SP} + P^{n+1}_{SP} \quad (1)$$

同様に，d_{n+1} を処理するために d_n のスケッチの内容を参照する必要があるため，d_{n+1} を処理中の SP によるスケッチ参照は，d_n を処理中の SP によるスケッチのアップデートが終了してから行なわれる必要がある（図 2 中の条件 2）．

$$R^n + R^n_{SP} + P^n_{SP} + U^n_{SP} \leq \Delta t^{n+1}_n + R^{n+1} \quad (2)$$

さらに，単一の計算ノードで全てのデータフレームを処理する場合，データストリームを取りこぼしなく処理するためには，d_{n+1} が到着するまでに d_n の読み込みが終了していなければならない．したがって，以下の制約条件が成り立たねばならない（図 2 中の条件 3）．

$$R^n + R^n_{SP} + P^n_{SP} \leq \Delta t^{n+1}_n \quad (3)$$

3.2 計算処理の並列化

式 1，式 2 および式 3 は，データストリームの取りこぼしを防ぎ，計算結果を正しく維持するための条件であり，式 1，式 2 および式 3 が成り立つ条件下では，資源の再配置やアプリケーションの並列化は不要である．しかし，データフレームの大きさや SP での処理時間に大きなバラツキがある場合，すなわち $R^n >> R^{n+1}$ や $P^n_{SP} >> P^{n+1}_{SP}$ となる場合や，データ到着間隔 Δt^{n+1}_n が極端に小さい場合には，式 1，式 2 および式 3 は成り立たなくなる．そこで本稿では，式 1，式 2 および式 3 が成立しない場合には，各データフレーム毎に

© 2010 Information Processing Society of Japan

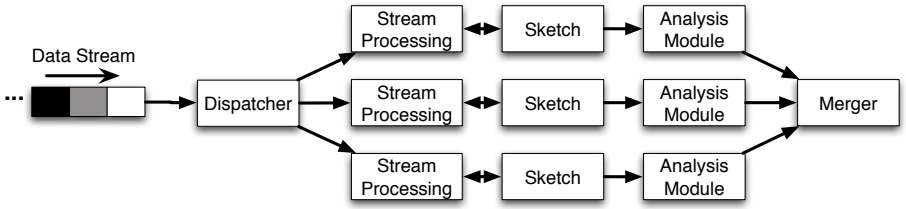

図 3 ストリーミング処理の並列化

異なる計算ノードを割り振り，ストリーム処理を並列化することを提案する．図 3 に 3 並列の場合の例を示す．ここで，Dispatcher はストリームとして到着するデータをフレーム毎に分割し，異なる計算ノードに割り振る．Merger は，それぞれの計算ノードから到着した結果を，元のデータストリームと同じ順番に並べ直し，データストリームとして出力する．

しかし，ストリーム処理において正しい計算結果を維持するためには，d_{n+1} を処理中の計算ノードは，d_n を処理中の計算ノードのスケッチを参照する必要がある．そこで，従来のようにスケッチを計算ノード内に持つのではなく，他のノードからも参照できる場所に置いた場合を考える．SP がスケッチからのデータ読み出しに要する時間を S^n_{SP} とすると，式 1，式 2 および式 3 と同様に，以下の時間制約を満たす必要がある．

$$R^n + S^n_{SP} + P^n_{SP} + U^n_{SP} + R^n_{AM} \leq \Delta t^{n+1}_n + R^{n+1} + S^{n+1}_{SP} + P^{n+1}_{SP} \quad (4)$$

$$R^n + S^n_{SP} + P^n_{SP} + U^n_{SP} \leq \Delta t^{n+1}_n + R^{n+1} \quad (5)$$

$$R^n + S^n_{SP} + P^n_{SP} \leq \Delta t^{n+1}_n \quad (6)$$

そこで提案アルゴリズムでは，以上の時間制約条件を考慮した上で，データの取りこぼしが発生する可能性がある場合に計算処理の並列化を行なう．

3.3 計算資源の動的再配置

3.2 節で述べた計算処理の並列化が必要な局面は，計算環境の負荷変動や入力データの変動により生じる．たとえば，計算環境の負荷増加やデータ入力コストの急増により，計算ノード数を増やして処理の並列化を行なう必要がある場合もあれば，計算環境の正常化やデータ入力コストの減少により，計算ノード数を減らして不要な計算資源を解放することが必要な場合もある．

ストリーミング処理の並列化に用いることができる計算ノードの数が m であるとき，以下の時間制約を満たすことで，データストリームを止めることなく計算処理を行なうことができる．

$$f(n) = R^n + S^n_{SP} + P^n_{SP} + U^n_{SP} + R^n_{AM} + P^n_{AM} + W^n \quad (7)$$

$$f(n) \leq \sum_{k=0}^{m-1} \Delta t^{n+k+1}_{n+k} + f(n+m-1) \quad (8)$$

式 7 より，計算ノード数が m では計算が完了せず，$m+1$ で完了する場合を考えると，制約条件は式 9 となる．そこで提案手法では，アプリケーションの動作および計算資源を監視し，式 9 を満たす範囲で最小の m 個の計算ノードを用いて処理を行なうように動的に m の値を変更する．

$$\sum_{k=0}^{m-1} \Delta t^{n+k+1}_{n+k} + f(n+m-1) \leq f(n) \leq \sum_{k=0}^{m} \Delta t^{n+k+1}_{n+k} + f(n+m) \quad (9)$$

4. 評　　価

提案する動的タスク配置アルゴリズム，および第 2 章で述べた資源配置アルゴリズムを，シミュレーションにより比較評価する．アプリケーションモデルとして，Twitter のツイートに対する処理を行なうような場合を想定し，ひとつのデータフレームに対して処理を行なうために必要な時間コストとデータフレームの到着間隔を 1 : 2，1 : 1，1 : 0.5，1 : 0.1 として，従来のようにストリームデータをストリームマイニングした場合と，提案手法によりストリームデータに対する処理を負荷分散および並列化した場合での比較を行なった．なお，今回のシミュレーションではデータ到着間隔およびデータ処理時間は一定であるが，想定するアプリケーションモデルを考慮すると，今回のジョブモデルは性能を評価するのに充分であると同時に，タスクの処理状況を明確に解析できるため，適していると言える．

なお，昨今では，Twitter では 1 分間に 1 億ツイート程度が投稿されているとされている．つまり，0.6×10^{-6} 秒ごとに新たなデータが到着する状況であり，今回のシミュレーションにおけるデータフレームごとの処理時間は，最大でも $6.0 \times 10^{-0.6}$ 秒しかない（ストリーム処理時間:データ到着間隔＝ 1 : 0.1 の場合）という状況である．したがって，今後の大規模ストリーム処理を想定した場合，今回のシミュレーションよりも厳しい状況でストリーム処理をしなくてはならない状況が発生する可能性は高い．

4.1 使用ノード数の比較

表 1 に，従来通りにストリーム処理を行なった場合と，提案手法を用いて並列化および負荷分散を行なった場合の最大利用ノード数の比較を示す．ここで従来手法では，並列化等

© 2010 Information Processing Society of Japan

表 1 データ処理時間とデータ到着間隔を変化させた場合の従来手法と提案手法での最大利用ノード数の比較

	1:2	1:1	1.0.5	1.0.1
従来手法	1	1	1	1
提案手法	1	2	15	15

の処理を用いていないため，データ処理時間とデータ到着間隔の比に関わらず，最大利用ノード数は 1 である．

提案手法を用いた並列化および負荷分散を行なった場合，データフレーム処理時間とデータ到着間隔の比が 1:2 の場合には，並列化を行なわずに全てのデータフレームを処理することが可能であった．しかし，データ到着間隔が短くなるにつれ，並列度を上げて処理することが必要になっているが，最も短時間間隔でデータが到着する場合（データ処理時間とデータ到着間隔の比が 1 : 0.1 の場合）でも，最大で 15 ノードを利用することで，全ての到着データを処理することが可能であった．

使用ノード数の比較により，Twitter の全ツイートを処理するような大規模なストリーム処理を想定した場合，単一のノードを用いたストリーム処理では全てのデータを取りこぼさずに処理することは難しく，ストリーム処理を並列化して 15 並列度以上で処理を行なうことが必要であることが確認できた．

4.2 データドロップ率の比較

次に，ストリーム処理時間とデータ到着間隔の比をさまざまに変化させた場合の，従来手法と提案手法でのデータドロップ率を図 4 に示す．ここで，データドロップ率とは，新たなデータフレームが到着した場合に利用可能な計算資源を確保できず，処理を行なわずに該当データを廃棄するに至ったデータ数の全到着データに対する割合である．ここで提案手法では，データを破棄しないために並列化および並列度を上げて処理を行なうため，データドロップ率は常に 0%となる．

一方で，従来手法を用いた場合のデータドロップ率は，データ到着間隔が短くなるにつれて単調増加する．データ処理時間とデータ到着間隔の比が 1 : 2 の場合には，従来手法を用いた場合でも，全ての到着データを処理する事が可能で，データドロップ率は 0%である．しかし，データ処理時間とデータ到着間隔の比が 1 : 1 になると 50%のデータを損失し，1 : 0.5 になるとデータドロップ率は 66%，1 : 0.1 ではデータドロップ率が 90%となり，ほとんどのデータは処理せずに廃棄することになる．

本比較結果および表 1 の結果をあわせて考察すると，従来のような単一ノードでのストリーム処理で到着データを全て処理することが可能な状況は，データ到着間隔がデータ処理時間の倍程度ある状況のみであり，これよりも複雑な処理を行ないたい場合や，データがより短時間間隔で到着する場合には，単一ノードのみで処理する場合には，データを破棄する必要性が生じることが分かる．つまり，本章の冒頭で述べたような現実的なアプリケーションを想定する場合には，提案手法のような手段を用いての並列化が不可欠であることが分かる．

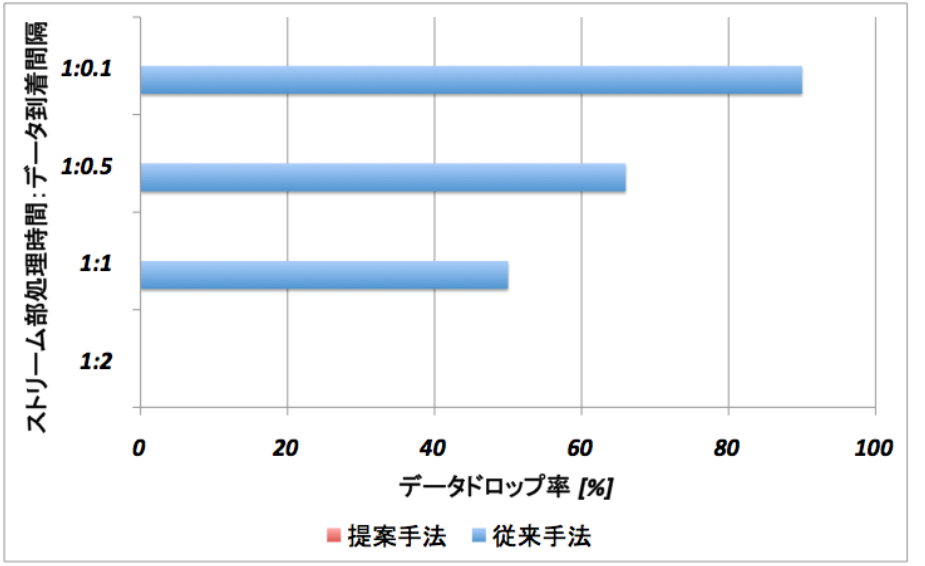

図 4 ストリーム処理時間とデータ到着間隔を変化させた場合の提案手法と従来手法でのデータドロップ率の比較

4.3 平均実行時間の比較

最後に，従来手法と提案手法でのデータフレームひとつに対する平均実行時間の比較を図 5 に示す．ここで平均実行時間とは，データが到着してから処理を行ない，処理結果が出力されるまでの時間を表す．図 5 ではストリーム処理時間とデータ到着間隔の比が 1 : 2 の場合（単一ノードによるストリーム処理が全てのデータを処理できるデータ到着間隔）のデータ処理時間を 1 とした相対処理時間を示している．さらに図 5 では，破棄したデータは集計対象としていない．したがって，常に単一ノードで処理する従来手法の平均実行時間は，常に 1 となる．

© 2010 Information Processing Society of Japan

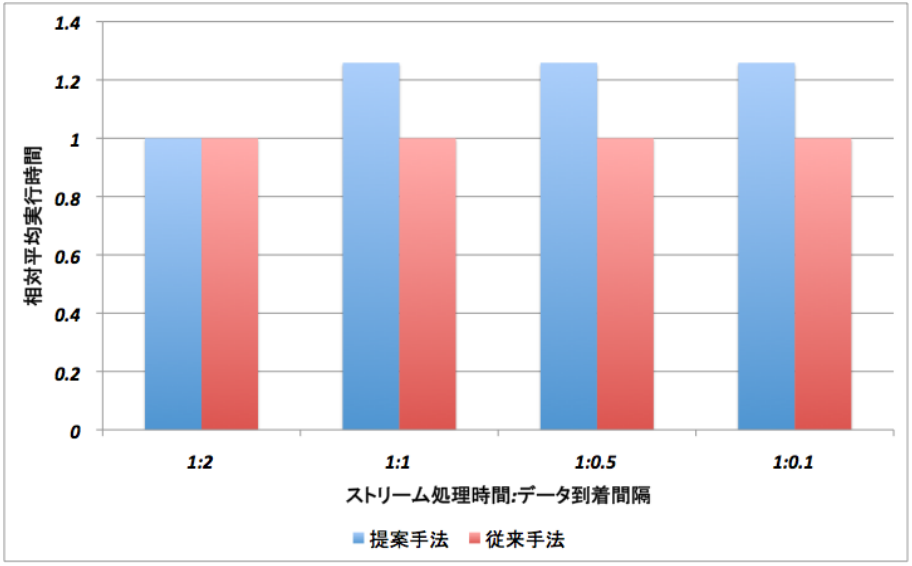

図 5 ストリーム処理時間とデータ到着間隔を変化させた場合の提案手法と従来手法での平均実行時間の比較

一方で，提案手法では，データが到着した後に，計算資源の選択を行ない，該当データを計算資源に移送してからデータ処理を開始するため，従来手法と比較してさまざまなオーバーヘッドが生じる．しかし，今回のシミュレーションでは，こうしたオーバーヘッドは約 26%であった．この結果について，別な味方をすれば，ストリーム処理時間とデータ到着間隔の比が 1 : 0.1 の場合（今回のシミュレーションでデータ到着間隔が最も短い場合）でも，提案手法では 26%程度のオーバーヘッドで全ての到着データを処理可能であるのに対して，従来手法では 90%のデータは破棄され，全体の 10%程度の到着データしか解析することができない．

4.4 評価実験のまとめ

ここでのシミュレーション結果は，ネットワーク負荷や利用可能な計算資源数，実際のデータ解析コスト等によって大幅に変化するものであるが，現実的なストリーム処理を想定した場合や，より複雑な解析処理を到着データに行ないたい場合には，ストリーム処理の並列化は不可欠であることが明確となった．

また，今回のシミュレーション結果では，並列化および負荷分散によるオーバーヘッドは 26%程度となっており，計算環境の状況によってさらに悪化することも，より小さいオーバーヘッドで運用可能な場合もあり得る．こうした様々な要因を詳細に考察し，オーバーヘッドの縮小を今後の課題としたい．

5. ま と め

本稿では，Twitter のツイートを時系列に沿って全て解析したり，多数ある監視カメラ等のセンサ情報をリアルタイムに処理するアプリケーションモデルを想定し，こうしたアプリケーションの入力データを全て確実に処理するための負荷分散および並列化の手法を提案した．また，シミュレーションにより，こうしたアプリケーションの入力データを破棄する事なくリアルタイムに処理するためには，ストリーム処理の並列化が不可欠であり，提案手法によりストリーム処理を並列化・負荷分散することで，全ての入力データを処理可能となることを示した．さらに，負荷分散や並列化に要するオーバーヘッドは 26%程度であることも示した．

今後は，こうしたオーバーヘッドの詳細な解析や縮小に向けたアルゴリズムの改善と，実環境で用いることができるフレームワークの実装を目指す予定である．

謝辞 本研究の一部は，文部科学省 次世代 IT 基盤構築のための研究開発「Web 社会分析基盤ソフトウェアの研究開発」“多メディア Web 解析基盤の構築及び社会分析ソフトウェアの開発”によるものである．ここに記して謝意を示す．

参 考 文 献

1) Shyamala Doraimani, and Adriana Iamnitchi, “File Grouping for Scientific Data Management: Lessons from Experimenting with Real Traces”, Proc. The International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC’08), 2008.
2) Ioan Raicu, Ian T. Foster, Yong Zhao, Philip Little, Christopher M. Moretti, Amitabh Chaudhary, and Douglas Thain, “The Quest for Scalable Support of Data-Intensive Workloads in Distributed Systems”, Proc. The International ACM Synposium on High Performance Parallel and Distributed Computing (HPDC’09), 2009.
3) Ioan Raicu, Yong Zhao, Ian Foster, and Alex Szalay, “Accelerating Largescale Data Exploration through Data Diffusion”, Proc. ACM Workshop on Data-Aware Distributed Computing(DADC08), 2008.
4) Steven Y. Ko, Ramses Morales, and Indranil Gupta, “New Worker-Centric

© 2010 Information Processing Society of Japan

Vol.2010-DBS-151 No.8
2010/11/12

Scheduling Strategies for Data-Intensive Grid Applications", Proc. the 8th ACM/IFIP/USENIX International Conference on Middleware, 2007.

5) Srikumar Venugopal, and Rajkumar Buyya, "A Set Coverage-based Mapping Heuristic for Scheduling Distributed Data-Intensive Applications on Global Grids", Proc. The 6th IEEE/ACM International Conference on Grid Computing (Grid2006), 2006.

6) Arun Ramakrishnan, Gurmeet Singh, Henan Zhao, Ewa Deelman, Rizos Sakellarious, Karan Vahi, Kent Blackburn, David Meyers, and Michael Samidi, "Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources", Proc. The Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid2007), 2007.

7) Bikash Agarwalla, Nova Ahmed, David Hilley, and Umakishore Ramachandran, "Streamline: A Scheduling Heuristic for Streaming Applications on the Grid", Proc. The 13th Annual Multimedia Computing and Networking Conference (MMCN2006), 2006.

8) Rich Wolski, "Dynamically Forecasting Network Performance using the Network Weather Service", Journal of Cluster Computing, 1:119 – 132, January, 1998.

9) Wen Zhang, Junwei Cao, Yisheng Zhong, Lianchen Liu, and Cheng Wu, "An Integrated Resource Management and Scheduling System for Grid Data Streaming Applications", Proc. The 9th IEEE/ACM International Conference on Grid Computing (Grid2008), 2008.

10) Liang Chen, and Gagan Agrawal, "A static resource allocation framework for Grid-based streaming applications", Concurrency and Computation: Practive & Experience, Vol. 18, Issue 6, 2006.

© 2010 Information Processing Society of Japan