クラウド環境を利用した Elastic なデータストリーム処理

石井　惇志†　　鈴村　豊太郎†‡

データストリーム処理を用いたリアルタイム性の高いアプリケーションを運用する場合、データレートの増大に対して、レイテンシの発散を抑えることが重要である。しかし現実的には、計算資源は有限であるため、データレートが一定以上になった場合に、レイテンシが発散することは避けられない。本研究ではそのような状況下で、クラウド環境上に処理を委譲することで、レイテンシの発散を抑える手法及びアーキテクチャを提案する。一方で、クラウド環境を利用するには経済的なコストを要するため、アプリケーションのレイテンシと経済的コストのトレードオフが発生する。本研究ではこれを最適化問題として扱い、コスト最適なスケジューリングを実行することで、クラウド環境の利用コストを最小化する手法も同時に提案する。また、そのシステムを実クラウド環境である Amazon EC2 を用いて構築し評価を行い、常にクラウド環境を利用する構成と比べて、経済的コストを 80%削減することに成功した。

Elastic Data Stream Processing with Cloud

ATSUSHI ISHII † and TOYOTARO SUZUMURA † ‡

In the case of operating applications running on the Data Stream Management System(DSMS), it is desired to avoid divergence of application's latency when data rate is high. However actually, it is almost inevitable because computational resources are finite. In this paper, we present a method and an architecture that transfers processes with the Cloud environment in order to avoid divergence of latency. Since a tradeoff exists between application's latency and economic costs when using the Cloud environment, we treat it as an optimization problem and describe optimal scheduling algorithm to minimize the economic cost for using Cloud environment. We also implemented this system using Amazon EC2, and through the experimental result, we succeeded reduce 80% economic cost while keep application's latency.

1. はじめに

データストリーム処理とは、時々刻々と送られるデータに対して、リアルタイムに処理を行う計算手法であり、この手法を用いたアプリケーションの運用ではリアルタイムの応答性が重要になる。データストリーム処理において、入力データを与えてから結果が出るまでの時間である、レイテンシは短ければ短いほど良い。アプリケーションによって、要求されるレイテンシ（秒単位、マイクロ秒単位など）は異なるが、時間とともに変化するデータレートが局所的に増大した場合でも、可能な限りレイテンシの増大を避け、リアルタイム性を維持することが要求される。

しかし、データレートの急激な増加に対して、計算資源を物理的に、かつ即時に追加するのは、配置スペース、手続き上の問題で現実的でない。逆に、急激な増加を見越して十分な計算資源を用意した場合、通常時は大部分の計算資源が無駄になり、余分な初期費用、維持費が発生する。

そこで、本研究では、データレートの急激な増加に対して、クラウド上の仮想マシン（Virtual Machine：以下、VM）を追加することで、レイテンシの発散を抑えることを提案する。クラウド環境を用いれば、数分程度で計算資源を追加することができ、物理的な計算資源の追加と比べれば遥かに高速にデータレートの増加に対応でき、更に配置スペースを考慮する必要がない。局所的な負荷の増大に対しても、クラウド上の計算資源をその間だけ追加することで、計算資源の構成を弾力的に、すなわち

† 東京工業大学
Tokyo Institute of Technology
‡ IBM 東京基礎研究所
IBM Research - Tokyo

Elastic に変化させ、レイテンシの増大を抑えることが出来る。

ただし、クラウド環境を利用するには経済的コストが伴う。具体例としては、Amazon EC2[2]などのパブリッククラウドは従量課金制である。レイテンシの発散を抑えられるとはいえ、必要以上に VM を起動してしまうと、その分コストが大きくなってしまう。そのため、SLA(Service Level Agreement)となるレイテンシを維持した上で、経済的コストを最小限に抑える必要が生じる。この問題に対処するため、本研究では、レイテンシと経済的コストのトレードオフを最適化問題として定式化し、必要十分な VM を確保する手法を提案する。

本研究の貢献として、以下の四点が挙げられる。第一に、どのようなアルゴリズムで、クラウド上に処理を委譲するか。第二に、どのようにレイテンシと経済的コストのトレードオフを定式化するか。第三に、どのタイミングでクラウド VM の起動、停止を制御するか。最後に現行のデータストリーム処理系には実行時に動的に計算ノードを追加する機構がないので、どのようにそれを解決するか、である。

続く第 2 章では、本研究で扱うデータストリーム処理と、クラウド環境について説明する。第 3 章では、本稿で提案するデータストリーム処理系の概要について述べ、第 4 章ではコスト最適なクラウド上の VM 数を導出する最適化問題の定式化について解説する。第 5 章では、実装したシステムのアーキテクチャについて述べ、第 6 章ではその性能評価を行う。第 7 章では実験結果に基づく議論を行い、第 8 章では関連研究について述べ、最後に第 9 章で、本稿の総括と今後の課題について述べる。

2. データストリーム処理とクラウド環境

2.1 データストリーム処理

データストリーム処理[3][4][5][6][7]とは、止め処なく生成される情報の流れをストリームと呼び、このストリームを蓄積することなく逐次処理していくという新しい計算パラダイムである。バッチ処理と呼ばれる計算対象を全てストレージに蓄積してから計算する従来の手法と違い、リアルタイムの応答が要求される場合や、時系列で前後する僅かなデータのみを参照すればよい計算や、全データの蓄積が物理的に困難な処理に適している。このような手法は音声や動画のストリーミングなど一部の処理では利用されていたが、データストリーム処理はこれを抽象・汎用化し、幅広い処理に対して適用できるよう洗練された処理系としてまとめられている点で従来とは異なっている。

このような処理系を DSMS / DSPS (Data Stream Management / Processing System)と呼ぶが、その一例として M.I.T.の Borealis[3]や IBM Research の System S[6][7]などが存在し、ここ数年活発な研究がなされている。多くの DSMS がシングルノードでの実行を前提としているが、Borealis と System S は分散環境上で実行可能である。System S はデータフロー図から直感的に処理を記述できる SPADE[3]という宣言的言語を持ち、自動性能最適化を行う SPADE コンパイラと SPC[4]という実行基盤を用いてデータストリーム処理を実現する。SPADE では組み込みオペレータで処理を記述するが、C++や Java といった汎用言語を用いた独自のオペレータや関数により高度に柔軟な処理も実装できる。このように、データストリーム処理やミドルウェアの研究に適するため、本研究では System S を利用する。System S の詳細については[4],[16],[17]を参照して頂きたい。

2.2 クラウド環境

クラウド環境とは、ネットワークを介して計算資源を仮想マシンとして提供する環境（IaaS: Infrastructure as a Service）であり、従量課金制などの課金方式によって外部に計算資源を提供するパブリッククラウドと、社内などの一定の範囲内で利用されるプライベートクラウドなどに分類される。既存のクラウド環境の例としては、パブリッククラウドである Amazon EC2[2]や、オープンソースの Eucalyptus[1]などが存在する。

本研究では、実際のクラウド環境での有効性を示すため、クラウド環境として Amazon EC2 を使用することとした。そこで以降では Amazon EC2 の概要について触れることにする。

Amazon EC2 は、Amazon Web Service のサービスの一つで、従量課金制に基づき VM をオンデマンドに提供するパブリッククラウドである。

VM の起動、停止などの各処理は、API として外部に公開されており、これを利用することで動的な VM の操作が可能になる。

VM インスタンスを立ち上げる実際の物理マシンを集約するデータセンターは世界各所に点在しており（例：アイスランド、アメリカ合衆国、シンガポール、東京）、自身の環境に近いデータセンターを利用すれば、レイテンシを小さく抑えることができ、逆に各所のデータセンターをそれぞれ利用すれば、世界各所からアクセスされる Web サービスを運営する上で都合が良い。

また、起動することのできる VM イメージの種類は「インスタンスタイプ」として起動時に選択可能である。VM イメージの数は多岐に渡り、Amazon が提供するイメージ、サードパーティが提供するイメージ、更に自分で独自のイメージを作成して使用することもできる。あらかじめ必要なソフトウェア、をインストールしておいたイメージを作成しておけば、個々の VM に対して、起動後にインストール作業をする必要がなく、時間の短縮につながる。

起動した VM インスタンスとの通信は、X.509 セキュリティ証明書を介して行う。また、ポートの解放の設定は、セキュリティグループと呼ばれる設定を利用することで、一律あるいは個別に設定可能である。

課金体系は、起動時間による課金、ネットワークトラフィックによる課金、Amazon S3(Amazon Simple Storage Service)や Amazon EBS(Amazon Elastic Block Store)などのストレージ使用料、更にオプションとしての種々の機能の使用料があり、基本的に従量課金制である。課金単位となる時間は 1 時間で、1 時間未満の起動時間は一律に 1 時間の起動とみなされる。

3. ElasticStream システムの概要

本稿で提案する、クラウド環境を用いた弾力性のある、Elastic なデータストリーム処理系を、"ElasticStream"と呼ぶことにする。本章では、本研究における、ElasticStream システムを実装する上での仮定と前提について述べ、その上で ElasticStream システムの設計について述べる。

3.1 システム要件と設計

最初に、本研究において、サービスの質の基準となる SLA(Service Level Agreement)はレイテンシとする。また、本研究におけるクラウドの定義は、Amazon EC2 、 Eucalyptus 等 の IaaS (Infrastructure as a Service) のみを指す。また、自身の保有する計算資源の数は固定的で、変化させないものとする。また、それらの計算資源では処理が追いつかない場合のみを対象とする。つまり、保有する計算資源だけでは SLA を守れず、必ずクラウド環境を使用することになることを前提とする。

対象とするアプリケーションに入力されるトラフィックのデータレートは変動が激しいものとする。

基本的に、アプリケーションの機能を、データの受信部分と計算処理部分に分割し、計算処理部分を外部に抽出、分散させることで並列化を図る。この並列化された計算処理を担当するアプリケーションの一部を、クラウド環境上で実行することで、動的な並列処理を実現する。

クラウド環境上に起動する、インスタンスタイプごとの VM の数の決定には、次章で述べる、レイテンシと経済的コストのトレードオフをモデル化した最適化問題を利用する。この結果を受けて、外部スクリプトで Amazon EC2 の API を操作することで、VM の動的な追加、削除を実現する。

3.2 実装

まず、第 2 章で述べたように、基盤となるデータストリーム処理系には IBM System S を用い、クラウド環境には、Amazon EC2 を用いる。現行の System S に無い機能については、Ruby を用いて実装する。また、計算処理のアプリケーション部分は、System S で実装することも可能だが、System S をアカデミックライセンスで利用するため、クラウド環境に導入することはできない。そのため、Ruby スクリプトを用いて実装する。また、クラウド環境の操作には Amazon EC2 提供のコマンドラインツールを利用する。

4. コスト最適な VM 数の計算手法

本章では、前章で述べた ElasticStream システムにおける、レイテンシの発散を抑えつつ、かつ経済的コストを最小にできる VM の個数を求めるための、最適化問題の定式化について述べる。

4.1 対象とするアプリケーションの分類

本研究では、ElasticStream システムで構成されるアプリケーションを、以下の二種類に分類する。入力されるデータストリームを分散して負荷を抑えるデータ並列型アプリケーションと、独立した処理内容が複数ある場合に、それらを分散させて処理時間の軽減を図る、タスク並列型アプリケーションである。

データ並列型アプリケーションは、データストリームの中の個々のデータに対して処理を行うアプリケーションで、全てのストリーム内のデータを観る必要がないアプリケーションが該当し、データストリーム処理の性質上、多くのアプリケーションがこちらの型に分類される。入力されるデータストリームを任意の割合で計算ノードに分散させ、各マシンの性能を最大限に引き出し、最小限のマシン数でレイテンシを維持することを図る。具体的なアプリケーションとしては、Twitter の投稿からの文字列マッチングや株式取引における VWAP（Volume Weighted Average Price：出来高加重平均価格）の計算などがある。

タスク並列型アプリケーションは、独立した計算処理が複数あり、処理負荷が大きい CPU インテンシブなアプリケーションが該当する。複数ある処理を各マシンに分散させ、入力データストリームは複製して全てのマシンに送る。これにより、データストリーム全体の統計情報等を必要とするアプリケーションにも対応できるが、代わりに複製したデータストリームの合計データ量が、ネットワークのバンド幅を超えてはいけないという条件が付く。分散されたマシンでは、各々が割り当てられた処理のみの結果を返し、最終的に出力を統合して最終的な出力とする。具体的なアプリケーションとしては、多次元要素に対する SST（Singular Spectrum Transformation）アルゴリズムによる異常検知[17]などが挙げられる。

4.2 クラウド制御の方針

第 2 章で述べたように、Amazon EC2 は従量課金制をとっている。本研究ではその中で、稼働時間課金とトラフィック課金に注目し、これらを最小化することで、全体の経済的コストを最小化することを考える。ただし、Amazon EC2 では 1 時間未満の起動時間は 1 時間分の課金額として計上されるなどの、プロバイダに応じた課金ルールが存在するが、本稿では起動時間について最小化を目指す。

$$Min:$$
$$Cost = \sum_{type}^{VMtype}(P_{type} + P_{Nin} \times D_{type}) \times x_{type} \quad ...(1)$$
$$Where:$$
$$\forall x_{type} \geq 0, \qquad \forall x_{type} \in N,$$
$$\sum_{type}^{VMtype}(D_{type} \times x_{type}) \geq (D_{next} - D_{local}) \quad ...(2)$$

図 1　最適化問題の定式化

本研究では、VM の起動、停止などの操作、及び最適化問題の計算を一定の時間間隔で行い、その時間間隔を“TimeSlot”と定める。このため、TimeSlot の幅以内の時間内に発生、収束するデータレートのバーストには対応できなくなる。しかし、TimeSlot の時間は十数分～数時間の時間幅を想定しているので、この時間内に発生、収束するようなバーストは、長時間稼働するアプリケーションの運用という観点から見て極めて局所的である。それゆえ、この瞬間的なバーストに対応できるまでの性能を求めるのは現実的ではない。以上より、このような局所的なバーストに対応することは本研究の範囲外とし、考慮しない。

コスト最適な VM の起動数を求めるために、TimeSlot で決められる単位時間ごとに、次の TimeSlot の時間内のデータレートを予測し、それをもとに最適化問題を解くことにする。この次の TimeSlot 時のデータレートを予想するための手法としては、[16] で用いられている、AR(Auto Regressive Model)モデルを変形したモデルである、SDAR (Sequentially Discounting AR Model)モデルを用いた時系列予知アルゴリズムなどが挙げられる。また、このデータレートの予測機構はシステム内の 1 コンポーネントとして実装されるため、アプリケーションの性質、状況に応じて適切なアルゴリズムを搭載することで性能最適化を図ることが出来る。

4.3 最適化問題の定式化

以下では、レイテンシと経済的コストのトレードオフの最適化問題の定式化について述べる。本研究では、この問題を、VM インスタンスタイプごとの起動数の線形計画問題として解く方針をとる。

定式化の主旨は、次の TimeSlot 時におけるデータレートを予測し、ローカルの計算資源で処理可能な量を求め、余剰分をクラウド上の VM に委譲するというものである。アプリケーションがデータ並列の場合は、ローカルで処理できるデータストリーム量を計算すればよく、タスク並列の場合は、ローカルで処理できるタスク数を求めればよい。

ローカル環境、クラウド上の VM で処理可能な量は、事前処理による性能測定により基準値を求めておけば一定の性能が期待できる。また、事前処理による固定値だけでなく、運用中にレイテンシのフィードバックを受けることで、動的にパラメータの更新をすることでも性能向上を図ることができる。

図 1 が、データ並列アプリケーションについて定式化した、目的関数と束縛条件である。目的関数は、VM が稼動している時間である起動時間と、クラウド VM へのデータ送信であるアップロードのトラフィック量を最小化するものである。

ただし、アップロード課金については、VM の起動数が多く、データ転送量が小さい場合には、全体の課金額に占めるトラフィック課金額の割合が小さくなる場合が考えられる。このような場合、VM に送信するデータ量の多寡よりも、そもそも VM を起動するか否かの方が、課金総額として大きな意味を持つ。そのため、データ転送量が最適化問題の結果に影響を及ぼさず、トラフィック課金のパラメータは除外できる。

クラウド VM からローカル環境へのデータ転送であるダウンロードのトラフィック量は、アプリケーションに依存することから最適化問題としては考慮しない。P_{type} はインスタンスタイプごとの課金額で、P_{Nin} はアップロードのデータ転送量課金額である。D_{type} が各インスタンスタイプに割り当てるデータ量で、x_{type} が求めるインスタンスタイプごとの起動台数である。また、(2)式において、D_{next} が予測された次の TimeSlot の間に入力される平均のデータレートで、D_{local} がローカル環境で処理可能なデータ量である。つまり、次のデータレートから求められる次の TimeSlot の間に到着するデータ量が、ローカル環境で処理できるデータ量を超えた場合、その超過したデータを処理できる最小の VM 数をインスタンスタイプごとに求めている。ここで必要なのは、各インスタンスタイプの起動台数なので、目的関数の値それ自体は重要でない。ゆえに TimeSlot の値に関係なく、課金パラメータは 1 時間ごとの課金額などの固定値を用いる。

束縛条件は、基本的にローカルの計算資源が処理できないデータストリームの量、もしくはタスクを、クラウド VM 側が全て処理できていることである。

5. ElasticStream システムの実装

本章では、第 3 章、第 4 章で述べた、ElasticStream システムの実装構成について述べる。図 2 は、今回実装した ElasticStream システムのコンポーネント図である。基盤となるデータストリーム処理系には System S を、System S に無い機能については Ruby スクリプトを利用して外部的に実装している。第 6 章でも触れるが、本稿ではデータ並列アプリケーションとして ElasticStream システムを実装する。また、定式化された線形計画問題を解く処理には、C++で実装された、オープンソースである lp_solve[13]を利用する。

動作の基本的な流れは、入力データストリームを StreamManager がローカル計算資源、クラウド

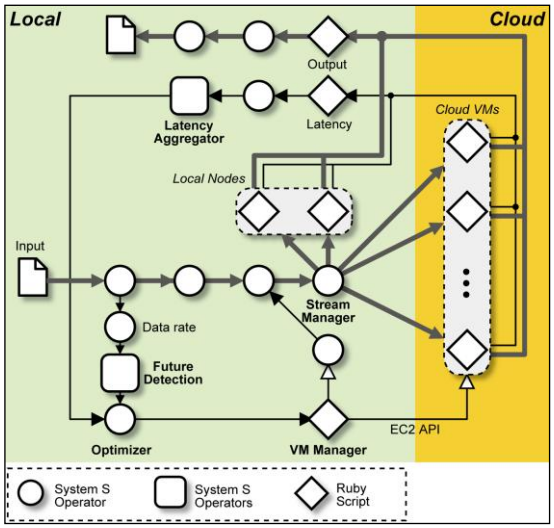

図２　ElasticStream システムの実装

VM にそれぞれ分散させ、計算結果を再びローカル側で集約する。一方で、現在のデータレートから次の TimeSlot 時のデータレートを予測し、その結果を受けて Optimizer が最適化問題を解き、次の TimeSlot の間に使用する VM の台数、及び各計算ノードに分配して割り振るデータストリームの比率を VM Manager に通知する。VM Manager は次の VM 数の通知を受けて、必要に応じて VM の起動・停止を行い、StreamManager にコネクションの追加、除去の通知とストリームの分配比率を送る。StreamManager は VM Manager からの通知を受けてコネクションの追加・除去を行い、ストリームの振り分け比率を更新する。この一連の処理を TimeSlot の間隔で繰り返す。

StreamManager は、メッセージを受け取るたびに各 VM への TCP コネクションの追加・除去を行い、また並列して、自身の保有するストリームの振り分け比率に基づいて、到着したストリームをそれぞれの VM に振り分けて送信する。

VM Manager へは、インスタンスタイプごとの VM 増減数と現在起動中の VM 数、及び各インスタンスタイプとローカル環境へのストリームの分配比率を送る。また、StreamManager へは、コネクションの追加を命令する ADD メッセージと、コネクションの削除を命令する REM メッセージの２種類を必要に応じて送信する。ADD メッセージには、インスタンスタイプ、Amazon EC2 側で与えられるインスタンス ID（例: i-1234abcd）、およびインスタンスアドレスを送り、REM メッセージではコネクションを一意に特定するためのインスタンス ID のみを送信する。また、インスタンスへのアドレスは VM 再起動時にも変化するため、VM を追加する度に、新しく割り当てられたアドレスを StreamManager に通知する必要がある。

VM Manager の動作アルゴリズムは次のようになる。最初に、Optimizer から受け取ったメッセージを元に、最初に追加する VM の起動を行い、その後停止する VM のコネクション除去メッセージ、追加メッセージをこの順で送り、最後に VM の停止を行う。これにより、まだ起動していない VM にストリームを送ろうとしたり、ストリームの送信が行われている VM を急に停止したりすることが保証される。

実際に計算処理を行う、ローカル、クラウド両方のノードでは、データの受信と結果の送信に使う TCP コネクションを追加し、以降は受信したデータを基に計算処理を行い、結果をローカル環境に送る。各計算ノードから送られる結果を、System S のオペレータが集約し、全体の結果とレイテンシを出力する。レイテンシは、入力されたタプルが出力オペレータ（図 2 における”Output”部分）に返却されるまでの時間差である。また、レイテンシは、インスタンスタイプごとにも集約し、出力している。本稿では利用しなかったが、このインスタンスタイプごとのレイテンシを利用し、Optimizer の最適化問題のパラメータを動的に更新することも可能である。

6.　性能評価

本章では、実装した ElasticStream システムの性能評価について述べる。本章では、第 4 章で分類したアプリケーションのタイプのうち、データストリーム処理において主流であるデータ並列アプリケーションを用いて性能評価を行う。

実装構成は第 5 章で述べた通りであるが、データレートの予測機構に関しては、ElasticStream システムの最高性能を測定するため、入力として与えるデータレートを、そのまま予測値として与えている。

実験の流れは、用意したアプリケーションに手動で生成したデータレートに基づいた入力を与え、1 秒毎のレイテンシと、使用した VM の台数と時間から求められる課金スコアとして評価する。課金スコアとは、Amazon EC2 で用いられる VM の時間課金額（ドル／時間）を用いて算出した、実験時間内に発生した金額の総計である。ただし、実際の Amazon EC2 では、一時間未満の起動は一律一時間として集計するが、本章では実験時間の短縮、効率化のため、その仕様を撤廃しているので、求めた課金スコアと実際の請求額は異なっている。なお、TimeSlot の幅を 1 時間位にすれば、このスコアは Amazon EC2 の請求金額と一致する。また、トラフィックによる課金額は、後述する本実験で用いるアプリケーションのデータ転送量では、時間課金額に比べて非常に少額で、かつインスタンスタイプに関わらず課金率は同額であるため、最適化問題のパラメータから除外している。

実験環境として、ローカルの計算資源には、1Gbps の Ethernet で接続された 2 台の計算機クラスタを使用する。1 台は分散された計算処理のみを行い、残りの 1 台で、ElasticStream システムを稼

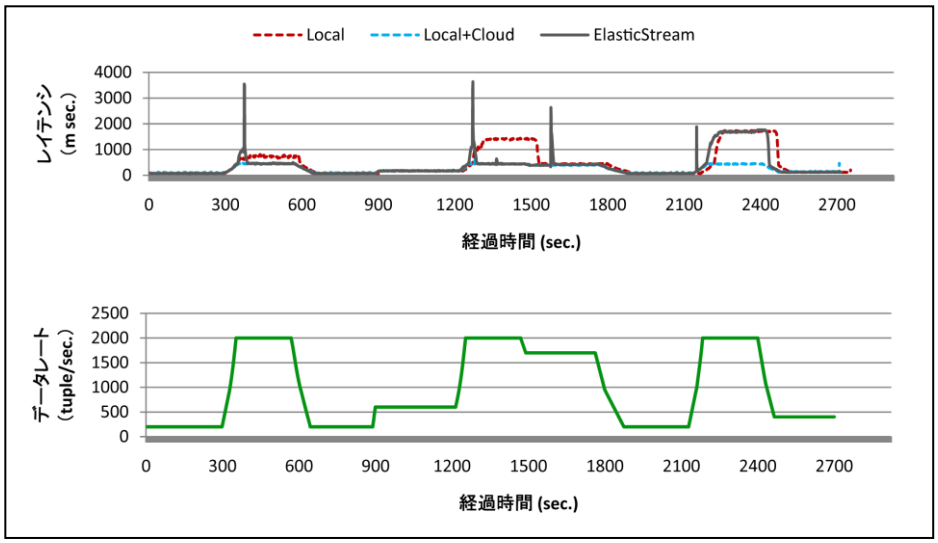

図４　時間経過によるレイテンシ及びデータレートの変化

表３ 各インスタンスタイプのハードウェア性能

Type	CPU	Cores	Mem (GB)	Price ($ / h)
Small	1 ECU	1 Core	1.7	0.095
Medium	5 ECU	2 Core	1.7	0.19

動させる。ハードウェア構成は、計算用ノード 1 台 : CPU AMD Phenom 9850 Quad-Core Processor 2.5GHz, Mem 8GB を、ElasticStream 用ノード : CPU AMD Phenom 9350e Quad-Core Processor 2GHz, Mem 4GB、ソフトウェア構成は CentOS 5.4 kernel 2.6.18-164 AMD64（計算用クラスタ 2 台）、CentOS 5.4 kernel 2.6.18-128 AMD64（ElasticStream 用ノード）、InfoSphere Streams 1.2.0(System S)、gcc version 4.1.2、Ruby 1.9.1 を用いる。

クラウド環境としては、Amazon EC2 を使用し、VM のインスタンスタイプとしては、Basic 32-bit Amazon Linux AMI Beta 2010.11.1 Small（以下、Small と略記）、同 High-CPU Medium（以下、Medium）の 2 種類を使用する。各インスタンスタイプのハードウェア性能、1 時間あたりの課金額は表 3 の通りである。なお、表中の”ECU”は、Amazon EC2 の VM における CPU 処理能力の単位である。また、VM を起動するデータセンターの場所は、事前測定により、レイテンシの最も小さかった US-West（北カリフォルニア）Region を使用する（一連の実験は、東京データセンターが設置される前に行われた）。

実験に使用するアプリケーションは、Twitter の投稿データの解析を想定した、正規表現マッチングを行うアプリケーションで、受け取った擬似データに対して正規表現マッチングを行い、マッチしたデータをローカル環境に結果として送信するものである。データストリーム内の個々のデータであるタプルのデータサイズは 1 つにつき 1KB としている。データサイズに関しては、0.1KB から 10KB までデータサイズを変えながらベンチマークを行ったところ、レイテンシに影響を与えなかったので結果として 1KB を設定している。また、実際の解析に要する負荷を考慮して、正規表現マッチング処理の前に、出力とは無関係な単純なループ処理を加えている。ローカル環境、VM Small、VM Medium に割り当てる最大データレートは、事前に行ったベンチマークの結果を基に、それぞれ 1600、40、90（タプル/秒）として、Optimizer にパラメータとして与えてある。また、TimeSlot の時間幅は 5 分間で実験を行った。

図 4 は、時間経過によるレイテンシの変化のグラフと、入力データレートの変化のグラフである。比較対象として、クラウド環境を利用しないローカル環境のみの構成（Local）と、ローカル環境及び VM Small を 1 台、VM Medium を 2 台常に使用する構成（Local+Cloud）でも同じ実験を行い、図 4 に示してある。レイテンシのグラフから、ローカル環境は高データレートの間にレイテンシが増大しているのに対し、ElasticStream システムは、VM を起動することでスループットを向上させ、レイテンシの増大を発散時の最大 3 分の 1 にまで抑えていること分かる。また、ローカル環境と VM の混成構成では、入力データレートに対して十分対応できる構成にしているが、ElasticStream システムは、VM の起動台数をそれ以下に抑えた上で、レイテンシの値は混成構成と同等にしていることが分かる。

最後のバーストに対しては、ElasticStream システムはレイテンシの発散を抑えられていないが、これは Optimizer が次の TimeSlot 間の平均データレ

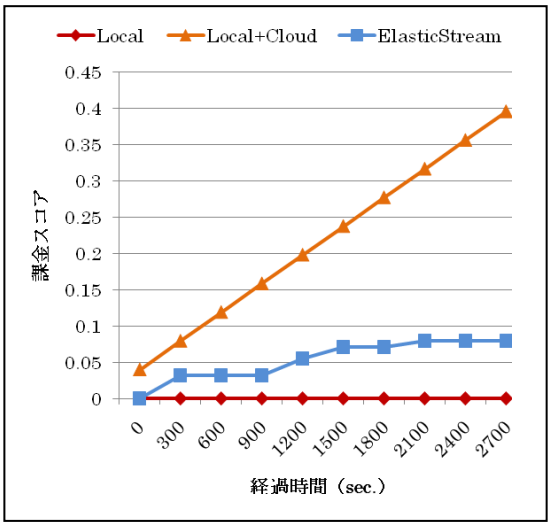

図５　各構成における課金スコア

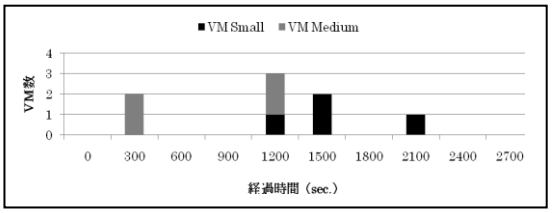

図６　システムが使用した VM 数

ートを元に最適化問題を解いているため、実際に届く最大のデータレートより低い値を平均値として受け取ることに起因する。これらの全てのバーストに対応するには、TimeSlot 時間幅を更に短くする、あるいはデータレートの予測アルゴリズムを改善し、次の TimeSlot における最大のデータレートを取得することが考えられる。また、ElasticStream システムのレイテンシは数箇所で一秒程大きく発散しているが、これは追加した VM にコネクション繋ぐ際の一瞬のデータ振り分けの停止、及び VM が起動する前にデータレートの変動が起きた事に起因するもので、実装上の問題として今後改善する予定である。

図 5、図 6 はこの実験による、ElasticStream が使用した VM の起動台数の変化と、すべての構成における課金スコアの変化を示したグラフである。課金スコアのグラフより、VM を常に使用するローカル環境とクラウド VM の混成環境では課金スコアが線形に増加していくのに対し、ElasticStream システムは必要な時だけ VM を起動しているので、スコアの増加を大きく抑えられていることが分かる。この実験による ElasticStream システムの最終的な課金スコアは、常に VM を利用した場合から約 8 割スコアを削減することに成功している。

全体の結果として、ローカル環境では経済的コストはかからないがレイテンシの発散を防げず、ローカル環境とクラウド VM の混成環境では、レイテンシの発散を常に防ぐことができるが経済的コストが大きくなってしまうのに対し、本稿で提案した ElasticStream システムでは、課金額を常に VM を立ち上げた場合の 20%に抑えた上で、レイテンシの発散を可能な限り抑えることに成功した。

7. 議論

本章では第 6 章で得た実験結果を元に、本稿で提案した ElasticStream システムの問題点と改善案について述べる。

第 6 章で行った実験では、TimeSlot の時間幅は 5 分間で、その時間が経過する度に VM の起動・停止を行っていたが、Amazon EC2 では 1 時間未満の稼働時間は 1 時間分として課金されるので、このシステムをそのまま利用しても余計なコストが生じる。これに対応するには、VM Manager に各 VM の起動時間を管理する機構を追加し、かつ VM の管理ポリシーに「一度起動した VM は 1 時間は必ず使用する」事を追加することで、TimeSlot を短くしたまま、Amazon EC2 の課金ルールに対応できる。

また、第 6 章では TimeSlot を短くすることで、短時間のデータレートのバーストに対応できることを述べたが、TimeSlot を短くし過ぎると、約 100 秒程度の VM の起動時間が全体の起動時間を大きく占めることになり、頻繁に VM の起動・停止が必要になるデータレートの変動が起きた場合に無駄が生じる。これについては、TimeSlot の幅を変更しながら、様々なデータレートを入力として実験を行うことで、最適な TimeSlot の幅を求められると考えられる。

最後に、本研究では起動するインスタンスタイプごとの VM 数の決定に、線形計画法を用いた最適化問題を利用しているが、この手法以外にも、データレートの変動を検知して、一定数の VM を立ち上げるなどの手法も考えられる。他の手法に対する、最適化問題を利用するメリットとしては、必要十分な VM の量を直接求められる点と、線形計画問題を拡張することで、各インスタンスタイプの処理性能だけでなく、起動するリージョンの位置や、ユーザーの要求に合わせた重み付けを盛り込むことが出来る点が挙げられる。

8. 関連研究

データストリーム処理における負荷分散手法として、データの一部を間引いて処理を行う Load Shedding という技法があるが、[8]では、その Load Shedding を用いながらオペレータの分割を行っている。また、[20]でも、クラウド環境とローカル計算資源の混成環境での負荷分散を、Load Shedding を用いて行っている。これらの研究に対して、本研究では、データは間引かず全て処理する方針をとっ

ている。

[15]では、System S のオペレータとして負荷分散機能を組み込んでいる。この研究では単一オペレータ内で負荷に応じて Elastic にスレッド数を増減させているのに対し、本研究では処理の委譲先をクラウド環境に求めているという点で異なる。

ジョブスケジューリングについては、[9]で実行時間の最小化の線形計画問題を用いたスケジューリング手法について述べている。また、[18]では、読み込むデータの局所性と複数ユーザーの送るジョブの公平性を維持することを主眼にしたスケジューリング手法を提案している。また、クラウド環境への処理の委譲に関しては、[11]ではバッチ処理を対象として、コスト最適なジョブスケジューリングを行っている。[11]では、最適化問題によるスケジューリングは、事前処理として最初に行っているが、本研究ではデータストリーム処理という、処理の終わりがないジョブを対象とするため、TimeSlot という概念を導入し、その都度最適解をアップデートしているという点で異なる。更に、[19]では、クラウドを利用したジョブスケジューリングにおける、複数のスケジューリングポリシーのコストの比較を行なっている。

[14]では、データストリーム処理系におけるオペレータの分割について焦点を当てている。この研究では、オペレータの粒度を小さくして分散させることで並列化を狙っている。また、[10]では、複数クラウド環境を想定した VM の配置について検討している。本研究では単一のクラウド環境を想定しているが、Amazon EC2 という実環境上で評価を行ったという点で異なる。また、最適化の対象として、[12]のように消費電力の最小化を目的関数とするという研究もある。

9. まとめと今後の課題

本稿の貢献として、データストリーム処理を用いたアプリケーションの負荷分散手法としての ElasticStream システムのアーキテクチャの提案をしたこと、また処理の委譲先であるクラウド環境の利用コストとレイテンシのトレードオフを最適化問題として定式化し、コスト最適な負荷分散を実現する手法を提案したこと、更に、現行のデータストリーム処理系に無い、実行中の動的な計算ノードの追加を外部的に実装したこと、最後に、それらを実クラウド環境上で実装、性能評価を行い、結果としてレイテンシの発散を可能な限り抑えた上で、クラウドの利用コストを、常に利用する場合と比較して 80%削減することに成功したことが挙げられる。

今後の展望として、データレート予測アルゴリズムの改善が挙げられる。また、本稿において、ElasticStream システムはミドルウェアである System S 上で実装しているが、これは本来、ミドルウェアの機能として内包されるべき機能である。よって、実際のデータストリーム処理系の内部に ElasticStream システムの機能を適用することも今後の課題である。

謝辞 : 本研究の一部は科学研究費補助金・挑戦的萌芽研究（課題番号:22650017) の助成によって行われた。

参 考 文 献

[1] Daniel Nurmi, et al., The Eucalyptus Open-source Cloud-computing System, Proceedings of the 2009 9th IEEE/ASCM International Symposium on Cluster Computing and the Grid

[2] Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/

[3] Daniel J. Abadi, et al., The Design of the Borealis Stream Processing Engine, CIDR'05, Asilomar, CA, 2005

[4] Bugra Gedik, et al., SPADE: The System S Declar ative Stream Processing Engine" SIGMOD 2008

[5] Lisa Amini, et al., SPC: A Distributed, Scalable Platform for Data Mining DM-SSP 2006

[6] J. L. Wolf, N. Bansal, et al, SODA : An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems, Middleware 2008.

[7] Bugra Gedik, A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing, ASCM CIKM 2009

[8] Barzan Mozafari, et al., Optimal Load Shedding with Aggregates and Mining, ICDE 2010

[9] JOSE R. CORREA, et al., LP-Based Online Scheduling: From Single To Parallel Machines, Mathematical Programming: Series A and B Volume 119 Issue 1, 2009

[10] Sivadon Chaisiri, et al., Optimal Virtual Machine Placement across Multiple Cloud Providers, Services Computing Conference, 2009. APSCC 2009

[11] Ruben Van den Bossche, et al., Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads, 2010 IEEE 3rd International Conference on Cloud Computing

[12] Gaurav Dhiman, et al., vGreen: A System for Energy Efficient Computing in Virtualized Environments, ISLPED '09

[13] Lp_solve, http://sourceforge.net/projects/lpsolve/

[14] Vincenzo Gulisano, et al., StreamCloud: A Large Scale Data Streaming System, In Proceedings of ICDCS'2010.

[15] Scott Schneider, et al., Elastic Scaling of Data Parallel Operators in Stream Processing, Parallel & Distributed Processing, 2009. IPDPS 2009.

[16] 松浦紘也、鈴村豊太郎,データストリーム処理系 System S と Hadoop の統合実行環境, Comsys 2010

[17] 森田康介、鈴村豊太郎,データストリーム処理システム System S を用いた異常検知アルゴリズムの実装と GPU による性能最適化, データ工学研究会 2010

[18] Matei Zaharia et al., Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, EuroSys '10 Proceedings of the 5th European conference on Computer systems

[19] Marcos Dias de Assunçao et al. , Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters, HPDC '09 Proceedings of the 18th ACM international symposium on High performance distributed computing

[20] Wilhelm Kleiminger et al. , Balancing Load in Stream Processing with the Cloud, 6th International Workshop on Self Managing Database Systems (SMDB), April 2011