ネットワーク帯域幅の不確実性を許容するエッジ・クラウドスケジューラの実装

深見 健太郎[1,a)]　有次 正義[2,b)]

概要：近年, モバイル端末や IoT センサの増加に伴い大量のデータが生成され, リアルタイム処理を必要とするアプリケーションの需要が高まっている．パブリッククラウドは従量課金制で豊富なリソースを提供するが, 実際の性能や帯域幅はデータセンタの物理ハードウェアや同じマシンに割り当てられた他のユーザのリソースの使用状況によって変動するため，リアルタイム処理に利用することは容易ではない. 一方でエッジコンピューティングは, IoT デバイスやユーザの近くにあるエッジで分散処理を行うことでアプリケーションの低遅延処理を容易に実現可能であるが, エッジの性能が均一でない場合や, エッジで期待できるリソースが乏しい場合を考慮する必要がある．本研究では, Apache Storm を用いてエッジとパブリッククラウドで構成する Storm クラスタの実装について議論する．具体的には, Storm クラスタの計算ノードの性能の不均一とネットワーク帯域幅の不確実性を考慮したスケジューラの実装を試みる．既存のスケジューラとリアルタイム処理を必要とするアプリケーションのエンドツーエンドレイテンシを比較した結果, 提案手法はレイテンシの減少を示した.

キーワード：負荷分散, スケジューリング，クラウドコンピューティング，リアルタイム処理技術

Implementation of an Edge Cloud Scheduler under Network Bandwidth Uncertainty

Abstract: In recent years, with the increase in the number of mobile devices and IoT sensors, a large amount of data has been generated, and the demand for applications that requires real-time processing has been increasing. Public clouds provide abundant resources on a pay-as-you-go basis, but the actual performance and bandwidth will vary depending on the physical hardware of the datacenter and the resource usage by other users assigned to the same machine. Therefore, it is not easy to use them for real-time processing. On the other hand, edge computing can easily achieve low-latency processing of applications by distributed processing on edges located near IoT devices and users, but it is necessary to consider cases where the performance of edges is not homogeneous and the resources expected at the edges are poor. In this research, we discuss an implementation of a Storm cluster consisting of edges and public cloud instances using Apache Storm. Specifically, we attempt to implement a scheduler that takes into account the heterogeneous performances of the computational nodes in the Storm cluster and the uncertainty of the network bandwidth. We measured the latency of an application in order to compare the proposal method with existing Apache Storm scheduler. Evaluation results show that our proposal method reduced the end-to-end latency compared with other schedulers.

Keywords: Load balancing, scheduling, cloud computing, real-time processing technology

1. はじめに

近年, AWS[1] などのパブリッククラウドは, 従量課金制で豊富なストレージや計算リソースを提供しており, アプリケーションなどの主要なタスクをクラウド上で処理することは効果的であると考えられている [2]. また, 近年モバイル端末や IoT センサの増加に伴い大量のデータが生成されるようになり, リアルタイムに処理を行う必要のあるアプリケーションの需要が高まっている [3].

[1] 熊本大学大学院自然科学教育部
[2] 熊本大学大学院先端科学研究部
[a)] fukami@st.cs.kumamoto-u.ac.jp
[b)] aritsugi@cs.kumamoto-u.ac.jp

© 2022 Information Processing Society of Japan

パブリッククラウドでは，豊富なストレージや計算リソースを利用できるが, 実際の性能や帯域幅は, データセンタの物理ハードウェアや同一のマシン上に割り当てられた他のユーザのリソースの使用状況によって変動するため, アプリケーションのリアルタイム処理に利用することは容易ではない．一方で, IoT センサやユーザの近くに物理的に配置されたエッジで分散処理を行いリアルタイムな処理をサポートするエッジコンピューティングが提案されている [4]．このエッジコンピューティングでは, アプリケーションの低遅延処理が期待できるが, エッジコンピューティングに用いるエッジは必ずしも計算ノードの性能が均一であるとは限らない．そして, 計算ノードの性能が不均一な環境では, アプリケーションのレイテンシが増加する場合がある [5]．

本研究では, Apache Storm(以降, Storm) を用いてエッジとパブリッククラウドで構成された Storm クラスタを構築し, リアルタイムな処理を必要とする Real-Time Edge Vision Application[5] を実装する．そして, エッジにおいてアプリケーションのレイテンシの削減が見込める環境とパブリッククラウドにおいてアプリケーションのレイテンシの削減が見込める環境の両方に対応するスケジューラ [6] の実装の詳細を議論する．既存の Storm のスケジューラは, 計算ノードの性能が均一であることを前提として開発されていたり, クラウドの利用を想定して開発されていないなどの問題点がある．そのため, ここではベイズ推定を用いてネットワーク帯域幅の不確実性を考慮したスケジューラの実装を試みる．そして, 実装したスケジューラの性能を評価するために, アプリケーションのエンドツーエンドレイテンシを測定し, 既存の Storm のスケジューラにおけるアプリケーションのエンドツーエンドレイテンシとの比較を行う．

2. 関連研究

Storm には, Round Robin と RAS[7] の 2 つのスケジューラがデフォルトで実装されている．しかし, どちらのスケジューラも Storm クラスタの計算ノードの性能が全て均一であることを前提として設計されている．そのため, 計算ノードの性能が均一であるとは限らないエッジではレイテンシが高くなり, エッジにおけるスケジューラとして適切ではない場合がある [5]．

Zhang ら [5] は, Storm を用いて計算ノードの性能が不均一であるエッジサーバで分散処理を行う Storm クラスタを構築した．そして, 計算ノードの性能が不均一である Storm クラスタにおいて, リアルタイム処理を要求するアプリケーションのレイテンシを削減することを目的としたスケジューラである Latency aware Task Scheduler(以降, LaTS) を提案した．しかし, Zhang らは, Storm クラスタを全てエッジで構築しており, クラウドの利用を考慮していない．そのため, LaTS ではスケジューリングに用いるネットワーク帯域幅は変動しないことを前提としている．本研究では, LaTS のスケジューリングに用いるネットワーク帯域幅の不確実性に対処するためにベイズ推定によるネットワーク帯域幅の推定を行うベイズ推定を用いた LaTS を実装する．

Muhammad-Bello ら [8] は, Iaas クラウドにおけるリソースや性能の変動による不確実性に対処したロバストな期限制約付きワークフロースケジューリングアルゴリズムを提案した．Muhammad-Bello らは, IaaS クラウドの不確実性は, IaaS クラウドにおいてスケジューリングの際に考慮しなければならない重要な特異性であると述べている．そのため, タスクの実行時間の予測とクラウドリソースのプロビジョニングの遅延に関連する不確実性に対処している．タスクの実行時間の不確実性を考慮する際には, 実行時間の推定に正確な推定値を使用するのではなく, 不確実性係数を用いて実行時間の推定値に上限と下限を決定し求めた実行時間の区間を使用する．また, ワークフローのメイクスパンが期限制約を守るようにスケジューリングする際に, クラウドの VM(Virtual Machine) の起動時間を考慮している．本研究では, Storm クラスタを構築するインスタンスのネットワーク帯域幅の不確実性を考慮する．そして, ネットワーク帯域幅の推定には, ベイズ推定を用いて作成した事後分布を用いる．

3. Apache Storm

Storm[9] はオープンソースのリアルタイム分散処理システムである．Storm は, Spout, Bolt そして Topology の 3 つで構成される．Topology とは, Spout と Bolt から構成される有向非巡回グラフである．Spout は Topology 内のストリームデータのソースであり, 外部ソースからタプルを読み取り, 後続の Bolt へ出力する．ここでのタプルとは処理されるデータの基本単位である．Bolt は, Spout または他の Bolt から受け取ったタプルに対して決められた処理を行い, 後続の Bolt へ新たなタプルを出力する．

Storm クラスタは, Master Node, Slave Node, Zookeeper から構成される [10]．図 1 に Storm クラスタの構成の例を示す．Master Node は Slave Node へのタスクのスケジューリングを行う Nimbus というデーモンを実行する．Slave Node は Supervisor というデーモンを実行するノードであり, Supervisor が立ち上がった際に, Worker プロセスを Java Virtual Machine(以降, JVM) に立ち上げる．そして, Topology が実行される際に, Nimbus が各 Slave Node 上の Worker プロセスの Executor に Spout と Bolt のタスクの割り当てを行い Topology の処理が実行される．Zookeeper は, Master Node と Slave Node の間で Nimbus と Supervisor の状態を記録し, 保持することで, Storm クラスタのノード間の状態管理と同期を行う．

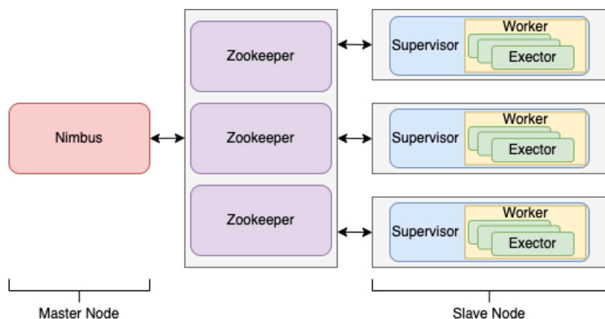

図 1　Storm クラスタの構成

4. 実装手法

本章では, ネットワーク帯域幅の不確実性を考慮するスケジューラである, ベイズ推定を用いた LaTS[6] の実装の詳細について述べる.

4.1 ベイズ推定を用いた LaTS

本研究では, Storm クラスタにクラウドを利用することで発生するネットワーク帯域幅の不確実性に対処するためにベイズ推定を用いて帯域幅の推定を行う. 帯域幅の推定には, 下記の (1) 式と (2) 式 [11] を用いる. また, 表 1 に (1) 式と (2) 式に用いた記号とその定義を示す.

$$\bar{\mu} = \frac{\frac{n}{\sigma^2}\bar{x} + \frac{1}{\tau^2}\eta}{\frac{n}{\sigma^2} + \frac{1}{\tau^2}} \tag{1}$$

$$\bar{\sigma^2} = \frac{\frac{\sigma^2\tau^2}{n}}{\frac{\sigma^2}{n} + \tau^2} \tag{2}$$

表 1　主な記号と定義

記号	定義
μ	正規母集団の母平均
σ^2	正規母集団の母分散
n	正規母集団から抽出したデータ数
$\bar{x}$	標本平均
η	事前分布に用いる正規分布の平均
τ^2	事前分布に用いる正規分布の分散
$\bar{\mu}$	事後分布の平均
$\bar{\sigma^2}$	事後分布の分散

(1) 式, (2) 式は, 母平均 μ, 母分散 σ^2 に従う正規母集団から n 個の標本を抽出し, 標本平均を $\bar{x}$, 事前分布を平均 η, 分散 τ^2 の正規分布とする場合の事後分布の平均 $\bar{\mu}$, 分散 $\bar{\sigma^2}$ を導出する式である. 本研究では, 事前実験を行い Storm クラスタの帯域幅を計測する. そして, 計測した帯域幅を用いてベイズ推定を行い, 導出した事後分布の平均を帯域幅の推定値とする.

LaTS は, Storm クラスタの計算ノードの性能が不均一である環境においてレイテンシを削減することを目的としたスケジューラである [5]. LaTS は以下の 3 つの要素で構成されている.

- タスクの要求リソースと実行時間の推定.
- Storm クラスタ内の利用可能なリソースの監視.
- Latency Aware Task Scheduling.

4.2 タスクの要求リソースと実行時間の推定

タスクのスケジューリングのために, タスクの実行時間, タスクが要求するメモリ, ネットワーク使用量, そして CPU 使用率の 4 つを利用する. メモリ使用量とネットワーク使用量はタスクがどの計算ノードで実行されても一定なので JVM の ThreadMXBean[12] を用いて取得する. タスクの実行時間の推定には推定モデルを作成する. まず, CPU 使用率を 5% から 100% まで 5% 毎に変動させながら各タスクの実行時間を測定する. 実行時間の測定には JAVA Timer API[13] を使用する. そして, 測定値を用いて予測平均二乗誤差が最小となるよう 3 次の多項式曲線を導出しタスクの実行時間を推定する推定モデルを作成する. CPU 使用率は ThreadMXBean で取得する.

4.3 Storm クラスタ内の利用可能なリソースの監視

Storm クラスタの各計算ノードで利用可能なリソースを取得する. 取得するのは以下の 4 つである.

(1) lscpu ユーティリティを使用して計算ノードの CPU クロック周波数と使用率を取得する.
(2) Storm のデーモンから Worker プロセスのメモリ使用量を取得する.
(3) ThreadMXBean を使用して計算ノードの CPU 使用率を取得する.
(4) ベイズ推定を用いて計算ノードのネットワーク帯域幅を推定し取得する.

4.4 Latency Aware Task Scheduling

LaTS は, 割り当てられるタスクのプールが与えられた場合に, CPU 時間の降順にソートを行い 1 つずつ割り当てを行う. その際, 計算ノードの CPU 使用率を測定し, 4.2 節の推定モデルを使用してタスクの実行時間を推定する. 次に, ベイズ推定を行い取得したネットワーク帯域幅と Bolt の出力するタプルのデータサイズを用いてデータ転送時間を推定する. そして, 推定した 2 つの推定値の合計時間が最小となる計算ノードへ Bolt の割り当てを行う. その後, 割り当てられた Bolt によって消費される CPU 使用率, メモリ, そして帯域幅を利用可能なリソースから削除する. 全ての Bolt の割り当てが完了するまで上記の処理を繰り返す.

4.4.1 関数 scheduleComponent

LaTS を実装したコードの一部を, 図 2 に示す. 図 2 は, 割り当てられるタスクのタスク名, タスク名とタスクを実行する Executor のマップ, そして利用可能な supervisor のリ

図 2 関数 scheduleComponent

```
1  private void scheduleComponent(String task, Map<String, List<ExecutorDetails>>
       componentToExecutors, List<String> supervisorsList) {
2      List<ExecutorDetails> executors = componentToExecutors.get(task);
3      if (executors == null) {
4          return;
5      }
6      for (ExecutorDetails ed : executors) {
7          String optimalNode = findOptimalNodeForComponent(task, supervisorsList
               ); // figure3
8          if (!executorToNodeMap.containsKey(optimalNode)) {
9             List<ExecutorDetails> executorDetailsList = new ArrayList<>();
10            executorToNodeMap.put(optimalNode, executorDetailsList);
11         }
12         executorToNodeMap.get(optimalNode).add(ed);
13         updateNodeResource(optimalNode, task, componentToExecutors); //
               figure5
14         if (nodeContainsTaskMap.get(optimalNode) == null) {
15             List<String> taskList = new ArrayList<>();
16             taskList.add(task);
17             nodeContainsTaskMap.put(optimalNode, taskList);
18         } else {
19             nodeContainsTaskMap.get(optimalNode).add(task);
20         }
21     }
22 }
```

図 3 関数 findOptimalNodeForComponent

```
1  private String findOptimalNodeForComponent(String task, List<String>
       supervisorHostList) {
2      double lowestLatency = Integer.MAX_VALUE;
3      String optimalNode = null;
4      for (String node : supervisorHostList) {
5          double latency = computeOverallLatencyForTask(task, node); // figure4
6          if (latency < lowestLatency) {
7              lowestLatency = latency;
8              optimalNode = node;
9          }
10     }
11     return optimalNode;
12 }
```

ストを入力とする．2 行目から 5 行目では, タスクを実行する Executor のリストを取得し, Executor が存在しない場合は, タスクのスケジューリングを行わず関数を終了する．6 行目から 21 行目の for 文では, Executor を割り当てる計算ノードを決定し, 計算ノードから割り当てられた Executor が消費する CPU 使用率, メモリ, そして帯域幅を利用可能なリソースから削除する．タスクを割り当てる計算ノードの決定は, 7 行目の関数 findOptimalNodeForComponent で実行する．findOptimalNodeForComponent については 4.4.2 節に示す．8 行目から 11 行目では, Executor を割り当てる計算ノードと割り当てられる Executor のマップに 7 行目で決定した計算ノードが含まれていない場合, マップに追加する．12 行目では, Executor を割り当てる計算ノードと割り当てられる Executor のマップに新たに割り当てられる Executor を追加する．13 行目の関数 updateNodeResource で計算ノードから Executor が消費するリソースを利用可能なリソースから削除する処理を行う．updateNodeResource については 4.4.4 節に示す．14 行目から 20 行目では, タスクを割り当てる計算ノードと割り当てられるタスクのマップに, 7 行目で決定した計算ノードが含まれていない場合に, 計算ノードと割り当てられるタスクを追加する．すでに, マップに計算ノードが含まれている場合は, 計算ノードに割り当てられるタスクをマップに追加する．関数 scheduleComponent は上記の処理を, 計算ノードへの全てのタスクの割り当てをマッピングするまで繰り返す．

4.4.2 関数 findOptimalNodeForComponent

関数 findOptimalNodeForComponent のコードを図 3 に示す．図 3 のコードは, 割り当てを行うタスクのタスク名と利用可能な Supervisor を持つ計算ノードのリストを入力とする．そして, 計算ノード毎のタスクの実行時間とタプルの転送時間の合計を比較し, 合計値が最小となる計算ノードを返す．2 行目と 3 行目では, タスクの実行時間とタプルの転送時間の合計の最小値を代入する変数と, 比較を行う計算ノードのタスクの実行時間とタプルの転送時間の合計値を格納する変数を宣言する．4 行目から 10 行目までの for 文では, 関数 computeOverallLatencyForTask でタスクの実行時間とタプルの転送時間の合計を取得する．computeOverallLatencyForTask については 4.4.3 節に示す．そして, 暫定で合計が最小の計算ノードと新たな計算ノードの合計値の比較を繰り返し, 合計が最小となる計

© 2022 Information Processing Society of Japan

図 4 関数 computeOverallLatencyForTask

```
1  private double computeOverallLatencyForTask(String task, String node) {
2      double memoryFree = resourceMonitor.nodeTable.get(node).getMemoryFree();
3      double memoryDemand = nodeTopologyManagerMap.get(node).getTaskMemory(task)
           ;
4      if (memoryFree < memoryDemand) {
5          return Integer.MAX_VALUE;
6      }
7      double cpuUtilization = resourceMonitor.nodeTable.get(node).
           getCpuUtilizationFree();
8      double computingLatency = nodeTopologyManagerMap.get(node).getTaskLatency(
           task, cpuUtilization);
9      double uploadBandwidth = resourceMonitor.nodeTable.get(node).
           getUploadBandwidth();
10     double transmissionLatency = 0.0;
11     if(uploadBandwidth > 0){
12         transmissionLatency = nodeTopologyManagerMap.get(node).getTaskOutput(
               task) / uploadBandwidth * 1000;
13     }else{
14         transmissionLatency = Integer.MAX_VALUE;
15     }
16     double sum = computingLatency + transmissionLatency;
17     return sum;
18 }
```

図 5 関数 updateNodeResource

```
1  private void updateNodeResource(String nodename, String task, Map<String, List
       <ExecutorDetails>> componentToExecutors) {
2      ResourceMonitor.Node node = resourceMonitor.nodeTable.get(nodename);
3      double memoryFree = node.getMemoryFree();
4      double memoryDemand = nodeTopologyManagerMap.get(nodename).getTaskMemory(
           task);
5      node.setMemoryFree(memoryFree - memoryDemand);
6      double cpuUtilizationFree = node.getCpuUtilizationFree();
7      double cpuUtilizationDemand = nodeTopologyManagerMap.get(nodename).
           getTaskCostUtil(task, cpuUtilizationFree);
8      node.setCpuUtilizationFree(cpuUtilizationFree + cpuUtilizationDemand);
9      double uploadBandwidthFree = node.getUploadBandwidth();
10     double uploadBandwidthDemand = nodeTopologyManagerMap.get(nodename).
           getTaskUpBandwidth(task, cpuUtilizationFree);
11     double remainUploadBandwidth = Math.max(uploadBandwidthFree -
           uploadBandwidthDemand, 0);
12     node.setUploadBandwidth(remainUploadBandwidth);
13 }
```

算ノードを決定する．最後に, 12 行目でタスクを割り当てる最適な計算ノードを返す．

4.4.3 関数 computeOverallLatencyForTask

関数 computeOverallLatencyForTask のコードを図 4 に示す．図 4 のコードは, 割り当てられるタスクのタスク名と計算ノードの名前を入力とする．そして, タスクの実行時間とタプルの転送時間の合計時間を返す．2 行目から 6 行目のコードでは, 計算ノードの利用可能なメモリとタスクが要求するメモリを比較し, 計算ノードのメモリが不足している場合は int 型の最大値を返り値とする．これによって, メモリが不足しているノードにはタスクが割り当てられないようにしている．7 行目と 8 行目では, 計算ノードの CPU 使用率を取得し, 4.2 節で作成した実行時間の推定モデルを用いて, タスクの実行時間を推定する．9 行目と 10 行目では, ベイズ推定を用いて推定したネットワーク帯域幅を取得し, タプルの転送時間の値を代入する変数を宣言する．11 行目から 16 行目では, 計算ノードのネットワーク帯域幅が 0 より大きい場合に, タプルのデータ量とネットワーク帯域幅からタプルの転送時間を計算する．計算ノードのネットワーク帯域幅が 0 以下の場合は, 計算ノードにタスクが割り当てられないように int 型の最大値をタプルの転送時間とする．そして, 17 行目でタスクの実行時間とタプルのデータ転送時間の加算を行い, 18 行目で合計値を返す．

4.4.4 関数 updateNodeResource

関数 updateNodeResource のコードを図 5 に示す．図 5 のコードは, 計算ノードの名前, 割り当てられるタスクのタスク名, Executor のリストを入力とする．そして, 計算ノードにタスクが割り当てられることで Executor が消費する計算ノードの CPU 使用率, メモリ, そしてネットワーク帯域幅を計算ノードで利用可能なリソースから削除する．2 行目のコードでは, 計算ノードのクロック周波数, CPU 使用率, CPU のコア数, メモリ, ネットワーク帯域幅を取得する．3 行目から 5 行目では, 計算ノードの現在のメモリと Executor が要求するメモリを取得し, 減算を行い計算ノードの利用可能なメモリを更新する．6 行目から 8 行目

© 2022 Information Processing Society of Japan

では, 計算ノードの現在の CPU 使用率と Executor によって増加する CPU 使用率を取得し, 加算を行い CPU 使用率を更新する. 9 行目から 12 行目では, 計算ノードの現在の帯域幅とタスクが出力するタプルの転送によって消費される帯域幅を取得し減算を行う. そして, 計算後の帯域幅が 0 以上ならば帯域幅を減算後の値に更新し, 0 未満ならば帯域幅の値を 0 に更新する.

5. 実験

5.1 Real-Time Edge Vision Application

Real-Time Edge Vision Application とは, モバイル端末や IoT 機器で撮影された画像, 動画データを入力とし, フレームごとに処理を行うリアルタイムアプリケーションである. 本研究で Topology として実装するのは, ステレオカメラから取得した 2 枚の画像データを入力とし視差画像を出力するアプリケーション [5] である.

5.2 エッジにおいてレイテンシの削減が見込める環境でのスケジューリング

本実験では, Storm に 5.1 節で述べた Topology を実装する. Storm クラスタは, 4 台の実マシンで構成したエッジと AWS の EC2 提供の 3 台のインスタンスで構築する. そして, エッジサーバにおいてアプリケーションのエンドツーエンドレイテンシの削減が見込める環境で Topology を実行する. また, 実験には, 提案手法であるベイズ推定を用いた LaTS, LaTS, RAS, そして Round Robin の 4 つのスケジューラを使用する. Topology は, ステレオカメラで撮影した解像度 640×480 のグレースケール画像を入力として, スケジューラ毎に 1fps で 3 分間実行する. そして, 1 フレーム毎に発生するエンドツーエンドレイテンシを測定し比較を行う. 実験に使用したマシンとインスタンスのスペックを表 2, 表 3 にそれぞれ示す. また, 表 4 と表 5 にベイズ推定を用いて導出した帯域幅の推定値と iperf を用いて測定した LaTS のスケジューリングに使用する帯域幅をそれぞれ示す.

5.2.1 実験結果

5.2 節の実験結果を図 6 と表 6 に示す. 図 6 と表 6 より最小値から最大値までの値全てにおいて, ベイズ推定を用いた LaTS のレイテンシが最も小さい値を示した. RAS は, ベイズ推定を用いた LaTS の次に低いレイテンシを示した. Round Robin は最小値はベイズ推定を用いた LaTS, RAS と近い値を示しているが最大値が他のスケジューラと比較して最も大きい値を示しており, 図 6 を見るとレイテンシの値のばらつきが大きいことがわかる. また, LaTS は, 中央値から最大値は Round Robin より小さい値を示したが最小値から第 1 四分位数のレイテンシでは他のスケジューラと比較して最も大きい値を示した.

また, Storm UI[14] を用いて各スケジューラの割り当てを確認した結果, ベイズ推定を用いた LaTS では, エッジである i9-9900K のマシンに全てのタスクの割り当てが行われていた. そのため, マシン間のデータ通信による遅延が発生せずレイテンシが削減されたと考えられる. RAS は, i9-9900K と i7-3930K の 2 台にタスクの割り当てが行われており, ベイズ推定を用いた LaTS と比較するとマシン間のデータ通信による遅延が発生したため, ベイズ推定を用いた LaTS の次に小さいレイテンシを示したと考えられる. Round Robin は全てのマシンにタスクの割り当てが行われておりエッジとクラウド間でデータ通信が非常に多く発生したためレイテンシの値が大きくなったと考えられる. LaTS はベイズ推定を用いた LaTS と違い i9-9900K と Xeon E5-2686 の 2 台にタスクの割り当てが行われており, エッジとクラウド間のデータ通信による遅延が発生したためレイテンシの値が大きくなったと考えられる. ベイズ推定を用いた LaTS と LaTS のスケジューリング結果が異なるのは, データ転送時間の計算に用いたネットワーク帯域幅によるものだと考えられる. 表 4 と表 5 より, ベイズ推定によるネットワーク帯域幅の推定値は, LaTS のスケジューリングに使用するネットワーク帯域幅と比較して, エッジとクラウドのネットワーク帯域幅の差が大きい. そのため, ベイズ推定を用いた LaTS では, インスタンスのデータ転送時間とエッジのデータ転送時間の差が大きくなり, エッジに全てのタスクの割り当てが行われたと考えられる.

5.3 クラウドにおいてレイテンシの削減が見込める環境でのレイテンシの測定

本実験では, 3 台の実マシンで構成したエッジと AWS の EC2 提供の 3 台のインスタンスで Storm クラスタを構成する. Topology には, 5.1 節で述べたアプリケーションに標準正規分布に従う乱数を用いて作成した 2 つの 400×400 の正方行列の積を計算する処理を追加したものを実装する. そして, インスタンス側で処理を行うことでエンドツーエンドレイテンシの削減が見込める環境で Topology を実行する. 実験には, 5.2 節と同様に提案手法であるベイズ推定を用いた LaTS, LaTS, RAS, そして Round Robin の 4 つのスケジューラを使用する. そして, ステレオカメラで撮影した解像度 640×480 のグレースケール画像を入力として, Topology をスケジューラ毎に 1fps で 3 分間実行し, 1 フレーム毎に発生するエンドツーエンドレイテンシを測定し比較を行う. 実験に使用したマシンとインスタンスのスペックを表 7, 表 8 にそれぞれ示す. また, 表 9 と表 10 にベイズ推定を用いて導出した帯域幅の推定値と iperf を用いて測定した LaTS のスケジューリングに使用する帯域幅をそれぞれ示す.

5.3.1 実験結果

実験結果を図 7 と表 11 に示す. 図 7 と表 11 より, ベイズ推定を用いた LaTS と LaTS は最小値から第 3 四分位数

© 2022 Information Processing Society of Japan

表 2 5.2 節の実験に使用した実マシンのスペック

CPU	クロック周波数 (GHz)	コア数	メモリ (GB)	台数	ノード
i9-9900K	3.60	16	32	1	Slave Node
i7-3930K	3.20	12	16	1	Slave Node
i7-3770K	3.50	8	8	1	Master Node
i7-860	2.80	8	8	1	Slave Node

表 3 5.2 節の実験に使用したインスタンスのスペック

インスタンスタイプ	CPU	クロック周波数 (GHz)	コア数	メモリ (GB)	台数	ノード
m4.xlarge	Xeon E5-2686	2.30	4	16	3	Slave Node

表 4 ネットワーク帯域幅の推定値

マシン	帯域幅 (Mbps)
i9-9900K	725
i7-3930K	718
i7-860	720
Xeon E5-2686	502
Xeon E5-2686	503
Xeon E5-2686	476

表 5 LaTS の帯域幅の測定値

マシン	帯域幅 (Mbps)
i9-9900K	943
i7-3930K	942
i7-860	943
Xeon E5-2686	881
Xeon E5-2686	879
Xeon E5-2686	880

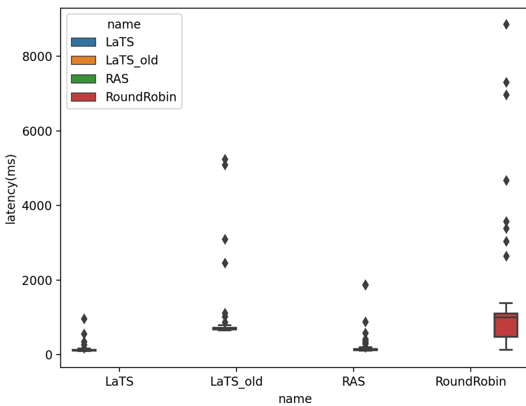

図 6 エッジにおいてレイテンシの削減が見込める環境でのスケジューラ毎の実験結果

表 6 スケジューラ毎の 1 フレームに発生するエンドツーエンドレイテンシ

スケジューラ	LaTS Bayes	LaTS	RAS	Round Robin
最小値 (ms)	91	648	106	135
第 1 四分位数 (ms)	106	675	124	481
中央値 (ms)	118	698	137	988
第 3 四分位数 (ms)	131	720	154	1104
最大値 (ms)	560	6617	1884	8857

までほとんど同じ値を示した．RAS は, ベイズ推定を用いた LaTS, LaTS と比較して最小値から第 3 四分位数まで高いレイテンシを示した．Round Robin は, 最小値においてベイズ推定を用いた LaTS, LaTS とほとんど同じ値を示した．また, 最大値においては最も低い値を示したが第 1 四分位数から第 3 四分位数において他のスケジューラと比較して最も高いレイテンシを示した．

Storm UI を用いて各スケジューラの割り当てを確認した結果, ベイズ推定を用いた LaTS と LaTS はどちらもインスタンスである 3 台の AMD EPYC 7R32 にすべての Bolt が割り当てられておりスケジューリング結果は同様のものとなっていた．これは, 表 9 と表 10 より, ベイズ推定によるネットワーク帯域幅の推定値と LaTS のスケジューリングに使用するネットワーク帯域幅の両方でクラウドのネットワーク帯域幅が大きいため両方のスケジューラでデータ転送時間がエッジのデータ転送時間より小さくなり, 全てのタスクがインスタンスに割り当てられたと考えられる．RAS は, 全ての Bolt がエッジである i7-3930K のマシンに割り当てられており, ベイズ推定を用いた LaTS と LaTS が割り当てを行なった AMD EPYC 7R32 との計算能力の差によってレイテンシが高くなったと考えられる．Round Robin は, 全てのマシンに割り当てを行なっておりエッジとインスタンス間でデータ通信が非常に多く発生したためレイテンシの値が大きくなったと考えられる．また, 図 7 と表 11 においてレイテンシの最大値が最も低い値を示していたが, 3 分間の Topology の実行時間中に処理を終えていないタプルが存在しており, 実際には最小値から最大値のレイテンシ全てにおいて他のスケジューラと比較して高いレイテンシとなっていると考えられる．

6. 終わりに

本研究では, パブリッククラウドを利用する際に発生するネットワーク帯域幅の不確実性に対処するためにベイズ推定によるネットワーク帯域幅の推定を行うスケジューラの実装を試みた．実験の結果, 既存の Storm のスケジューラと比較してベイズ推定を用いた LaTS は, エッジサーバにおいてアプリケーションのエンドツーエンドレイテンシの削減が見込める環境とクラウドにおいてエンドツーエンドレイテンシの削減が見込める環境の両方においてレイテンシの削減を確認できた．

今後の展望として, 高負荷な環境におけるスケジューラの評価を行う．本研究では, 単体のアプリケーションを 1fps という低負荷な環境で実行し提案手法を評価した．また, クラウドにおいてレイテンシの削減が見込める環境での実験でアプリケーションを実装する際に, 高負荷な環境を再現するために正方行列の積を計算する処理を追加した．しかし, 今後は, アプリケーションを複数同時に実行する場合や

© 2022 Information Processing Society of Japan

表 7 5.3 節の実験に使用した実マシンのスペック

CPU	クロック周波数 (GHz)	コア数	メモリ (GB)	台数	ノード
i7-3770K	3.50	8	8	1	Master Node
i7-3930K	3.20	12	16	1	Slave Node
i7-860	2.80	8	8	1	Slave Node

表 8 5.3 節の実験に使用したインスタンスのスペック

インスタンスタイプ	CPU	クロック周波数 (GHz)	コア数	メモリ (GB)	台数	ノード
c5a.2xlarge	AMD EPYC 7R32	3.30	8	16	3	Slave Node

表 9 ネットワーク帯域幅の推定値

マシン	帯域幅 (Mbps)
i7-3930K	643
i7-860	641
AMD EPYC 7R32	1951
AMD EPYC 7R32	1926
AMD EPYC 7R32	1954

表 10 LaTS の帯域幅の測定値

マシン	帯域幅 (Mbps)
i7-3930K	941
i7-860	943
AMD EPYC 7R32	4760
AMD EPYC 7R32	4750
AMD EPYC 7R32	4800

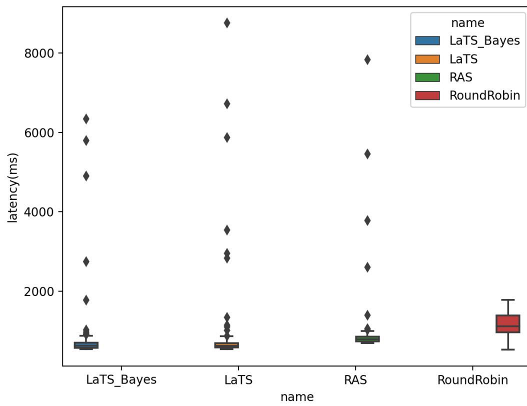

図 7 クラウドにおいてレイテンシの削減が見込める環境でのスケジューラ毎の実験結果

表 11 スケジューラ毎の 1 フレームに発生するエンドツーエンドレイテンシ

スケジューラ	LaTS Bayes	LaTS	RAS	Round Robin
最小値 (ms)	536	538	691	536
第 1 四分位数 (ms)	577	585	734	963
中央値 (ms)	629	624	786	1116
第 3 四分位数 (ms)	701	691	849	1329
最大値 (ms)	6347	8755	7835	1779

fps 数を増加させた場合などの高負荷な環境において提案手法がレイテンシを削減できるかを検討する．

参考文献

[1] Amazon web service. https://aws.amazon.com/jp/. (2022.1.21 参照).

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pp. 301–314, 2011.

[3] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pp. 68–81, 2014.

[4] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, Vol. 45, No. 5, pp. 37–42, 2015.

[5] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 1270–1278. IEEE, 2019.

[6] 深見健太郎, 有次正義. ネットワーク帯域幅の不確実性を考慮したエッジ・クラウドスケジューリング. 第 14 回データ工学と情報マネジメントに関するフォーラム, DEIM, 2022.

[7] Apache storm resource aware scheduler. https://storm.apache.org/releases/1.2.3/Resource_Aware_Scheduler_overview.html. (2022.1.21 参照).

[8] Bilkisu Larai Muhammad-Bello and Masayoshi Aritsugi. A robust algorithm for deadline constrained scheduling in iaas cloud environment. *Ieice Transactions on Information and Systems*, Vol. 101, No. 12, pp. 2942–2957, 2018.

[9] Apache storm. https://storm.apache.org/releases/1.2.3/. (2022.1.21 参照).

[10] Storm cluster. https://storm.apache.org/releases/1.2.3/Setting-up-a-Storm-cluster.html. (2022.1.21 参照).

[11] 姜興起. ベイズ統計データ解析. 東京:共立出版, 2010 年 7 月. 金明哲 (編).

[12] java.lang.management インタフェース threadmxbean. https://docs.oracle.com/javase/jp/6/api/java/lang/management/ThreadMXBean.html. (2022.1.21 参照).

[13] java.util.timer. https://docs.oracle.com/javase/jp/8/docs/api/java/util/Timer.html. (2022.1.21 参照).

[14] Storm ui. https://storm.apache.org/releases/1.2.3/STORM-UI-REST-API.html. (2022.1.21 参照).

© 2022 Information Processing Society of Japan