平成24年度　　卒業研究論文

ネットワーク帯域の変動に動的に対応するHadoopReduceタスクスケジューリング手法

指導教官
松尾 啓志 教授
津邑 公暁 准教授
梶岡 慎輔 助教

名古屋工業大学 工学部情報工学科
平成23年度編入学　21115171番

山崎 一樹

平成25年 2月 12日

目 次

1 はじめに

インターネットの普及とサービスの多様化により，世界中で生成されるデータは増大の一途を辿り，それに伴い処理する必要のあるデータも大規模になってきている．単一の計算機では処理能力の向上に限界があり，大規模なデータを処理するには多大な時間を要する．大規模なデータを高速に処理可能である高性能な計算機は非常に高価である上，障害が発生した場合システム全体を停止しなければならないという問題がある．そこで，安価な複数の計算機をネットワークで接続したクラスタを構成し，処理を分散することにより，全体での処理能力を向上させる分散並列処理が一般的となってきている．しかし，分散並列プログラミングは，計算機間のデータ通信処理や障害に対する処理など，実際に行いたいデータ処理以外にも様々な記述を行う必要があり繁雑である．このような分散並列プログラミングをより容易に行うための環境として，Hadoop[1] が注目を集めている．

Hadoop は大規模データを処理するための分散並列処理フレームワークである．これは，Google の利用している分散並列処理フレームワーク MapReduce[2] と分散ファイルシステム Google File System(GFS)[3] のオープンソース実装であり，それぞれ Hadoop MapReduce と Hadoop Distributed File System(HDFS) として実装されている．Hadoop を利用することにより，ユーザは入力を並列に処理する Map と，その結果を集約する Reduce という 2 つの処理を記述するだけで分散並列処理を行うことができる．分散処理に必要なノード間の通信処理や，タスクのスケジューリング，障害に対する処理などに関しては Hadoop のフレームワークとして実装されているものを利用する．このように Hadoop を用いることで容易に分散処理が実現できるため，Hadoop の需要は増加している．しかし，Hadoop には問題点も存在している．その問題点の 1 つとして Reduce タスクのスケジューリングが効率的でないということが挙げられる．Reduce タスクのスケジューリングでは，Map タスクのスケジューリングのようなデータローカリティを考慮した実装が行われておらず，ネットワークを介したデータの転送が多く必要となるようなスケジューリングが行われる可能性がある．この結果，処理時間の増加や，ネットワークリソースの浪費などを引き起こす．また，Hadoop はネットワークトポロジ情報を設定ファイルから読み込み，パフォーマンスの向上を目

的として，Map タスクのスケジューリングや HDFS のレプリケーションにこの情報を用いている．しかし，クラウドコンピューティング環境上で Hadoop を実行する場合は，実際のネットワークトポロジを把握し設定することは困難であり，Hadoop のパフォーマンスを低下させてしまう．

本研究は，Hadoop の Reduce タスクのスケジューリング手法の改善を行い，Hadoop のパフォーマンスを向上させることを目的とする．そのために，ネットワークの遅延に基づき Reduce タスクをスケジューリングする手法を実装し，ネットワークの状況に応じたスケジューリングを行うことでネットワークの負荷変動に動的に対応する．そして，バックグラウンドでネットワークに負荷をかけた状態で Hadoop を実行し，実装した手法がどの程度有効に働くか評価を行った．

本論文では，第 2 章で分散並列処理フレームワーク Hadoop について詳しく述べ，Reduce タスクのスケジューリング手法の問題点について説明する．第 3 章で Reduce タスクのスケジューリングの既存研究について触れ，第 4 章で提案手法とその実装の説明を行う．第 5 章では実装したスケジューラの評価と考察を行い，第 6 章で今後の課題を述べ，第 7 章で論文全体をまとめる．

2 研究背景

2.1 Hadoop の概要

Hadoop とは，Apache Software Foundation によって開発されている，大規模データを分散並列処理するためのフレームワークである．Hadoop は大別して，高スループットでのアクセスを可能とする分散ファイルシステム Hadoop Distributed File System(HDFS) と，分散並列処理フレームワーク Hadoop MapReduce の 2 つミドルウェアから構成される．これらは，それぞれ Google が提案し利用している分散ファイルシステム Google File System(GFS)，および分散並列処理フレームワーク MapReduce をオープンソースで実装したものである．以下でそれぞれのミドルウェアについて詳しく述べる．

2.2 Hadoop Distributed File System

Hadoop Distributed File System（以下 HDFS）は，Google File System のオープンソース実装である．HDFS は，データのメタデータを管理する 1 台のネームノードと，実際にデータを格納する複数台のデータノードから構成される．HDFS の主な特徴として，大容量，高スケーラビリティ，高スループット，耐障害性などが挙げられる．データノードとなる計算機を増やすことによりファイルシステムの容量を容易に拡張することができる．それと同時に HDFS では 1 つのデータを固定長のブロックに分割し，そのメタデータをマスタノードで管理し，ブロックは複数のデータノードに分散配置される．ブロックサイズはデフォルトで 64MB と比較的大きく設定されており，これにより HDFS は大規模データに対して高スループットでのアクセスが可能となっている．また，ブロックはその複製であるレプリカが作成され，それぞれが別のデータノードに格納されるため，たとえそのうちのあるデータノードが故障したとしてもデータが失われることはない．レプリカはデフォルトで 3 つ作成される．HDFS は Hadoop に設定されたクラスタのネットワークトポロジ情報からラック構成を考慮したレプリケーションを行う．図 1 のように，複数のネットワークスイッチによってクラスタがラックで分けられている場合は，2 つが同じラック内のノードに，残り 1 つを別のラック内のノードに格納するといったクラスタのラック構成を考慮した配置を行い，ラックスイッチの故障で 3 つ全てのレプリカにアクセスできなくなる状況の発生を防いでいる．

2.3 Hadoop MapReduce

2.3.1 Hadoop MapReduce の概要

MapReduce は，大規模データを複数の計算機を用いて分散並列処理するための Google が提案したプログラミングモデルであり，これをオープンソースで実装したものが Hadoop MapReduce である．Hadoop MapReduce（以下 MapReduce）では，クライアントが要求した仕事の単位をジョブと呼び，ジョブを複数のノードで並行実行可能な独立した処理に分割したものをタスクと呼ぶ．MapReduce では，ジョブやタスクのス

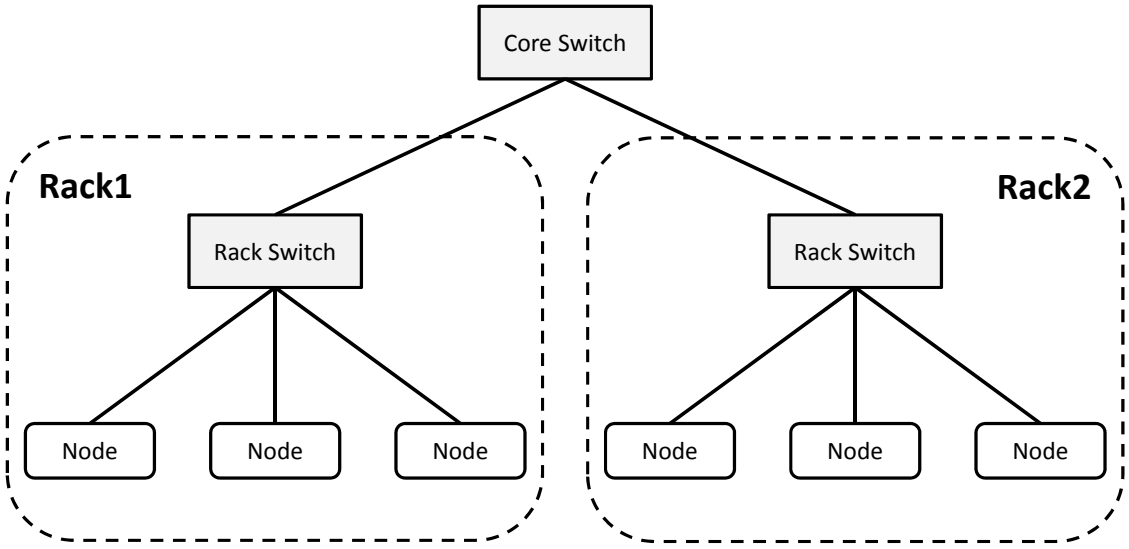

図 1: クラスタのラック構成

ケジューリングを行う 1 台の JobTracker と呼ばれるプログラムと，JobTracker によってスケジューリングされたタスクを実際に実行する複数の TaskTracker と呼ばれるプログラムから構成される．クライアントの要求したジョブを JobTracker がタスクに分割し，TaskTracker にスケジューリングし実行させることで MapReduce の処理が行われる．MapReduce における処理は大別して，入力データを複数ノードで並行に処理し中間データを出力する Map 処理と，その中間データを集約し最終的な結果を出力する Reduce 処理の 2 つがあり，それぞれ Map フェーズ，Reduce フェーズと呼ばれる．Map フェーズでは Map タスクが，Reduce フェーズでは Reduce タスクが JobTracker により TaskTracker にスケジューリングされ実行される．

MapReduce を利用することで，分散並列処理を容易に記述することができる．Map タスクと Reduce タスクをそれぞれ記述するだけで，分散並列処理を実現することができる．分散処理に必要なデータ通信や障害に対する処理などは，MapReduce のフレームワークとして既に実装されているため，ユーザは自身が実際に行いたいデータ処理の記述に専念することができる．

2.3.2 MapReduce のデータフロー

MapReduce におけるデーフローを図 2 に示す．MapReduce の処理は，複数の Map タスク，Reduce タスクによって行われ，これらのタスクのためにそれぞれ新しく JVM が生成され並列に実行される．まず，HDFS 上の入力データは複数のスプリットと呼ばれる小さなデータに分割される．スプリットごとに 1 つの Map タスクが対応する．スプリットは，複数の Key-Value ペアという単純なデータ構造に変換され map 関数に入力され Map 処理が行われる．map 関数は，それぞれの Key-Value ペアに対して処理したのち，中間データとなる Key-Value ペアを出力する．中間データは，Reduce タスクと同数のパーティションと呼ばれるグループに分割される．パーティションは，同一の Reduce タスクで処理が行われる中間データの集合である．

次に，Reduce 処理が行われるが，Reduce タスクは Map タスクによって出力されたパーティションを入力として処理を行う．パーティションはクラスタ全体の各ノードに分散されているため，まずこれらのパーティションは対応する Reduce タスクを実行するノードへコピーされる．集められたパーティションは，Key によってソートさ

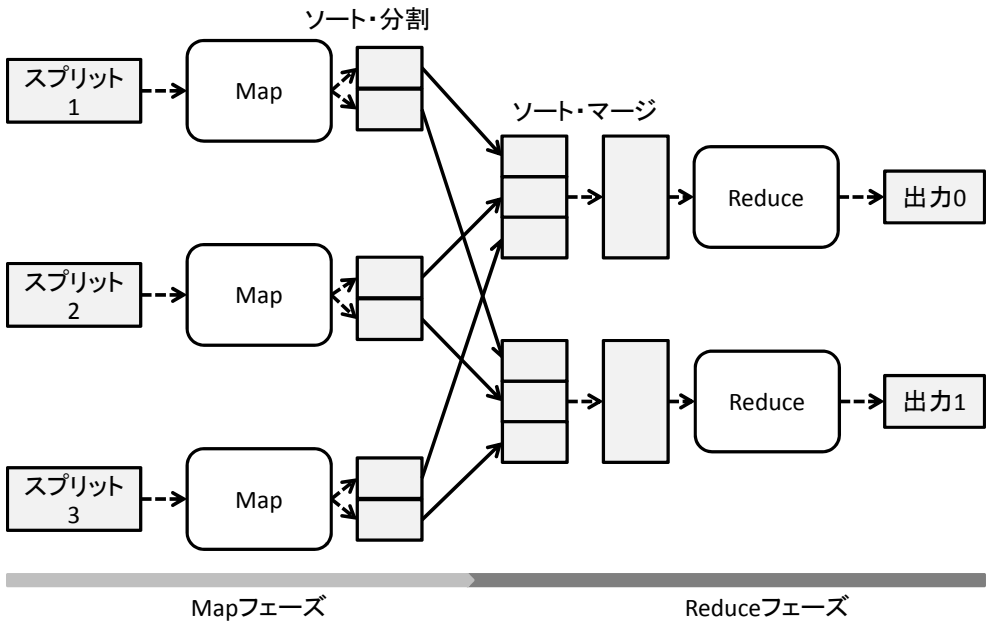

図 2: MapReduce のデータフロー

れ，さらに同一の Key を持つ複数の Key-Value ペアは，1 つの Key が複数の Value を持つ Key-Values ペアに変換される．この過程をマージと呼び，マージされた中間 Key-Values ペアは reduce 関数に入力され，集約処理が行われた最終的な結果の Key-Value ペアが HDFS 上にファイルとして出力される．

2.3.3 Map 処理の詳細

Map フェーズでは，並列に入力データの処理を行い中間データを生成する．Map フェーズの処理の流れを図 3 に示す．

入力データは HDFS に格納されており，これを複数のスプリットと呼ばれる小さなデータに分割する．それぞれのスプリットに 1 つの Map タスクが対応し，Map タスクは並列にスプリットを処理することが可能である．Map タスクでは，スプリットを複数の Key-Value ペアに変換する RecordReader が起動されている．MapRunner が RecordReader から順番に Key-Value ペアを１つずつ取り出し，ユーザによって定義された map 関数に入力する．map 関数では，入力された Key-Value に対してユーザによって記述された処理を行い，中間データとなる Key-Value を Map タスク内の MapOutputBuffer に出力する．MapOutputBuffer がある程度中間 Key-Value ペアで埋まると，SpillThread によって spill が行われる．spill は，MapOutputBuffer を空けるためにバッファの内容を Spill ファイルと呼ばれるファイルに書き出す処理である．このとき，MapOutputBuffer 内の Key-Value ペアは，Key によってソートされた後，パーティションに分割されて Spill ファイルに書き出される．パーティションへ分割は Partioner によって行われ，デフォルトでは Key をハッシュ関数にかけた値がパーティション番号に用いられるが，ユーザが独自の Partitioner を実装することもできる．MapRunner が全ての Key-Value ペアを map 関数に入力し処理し終えると，複数の Spill ファイルは 1 つのファイルにマージされ，Map タスクの中間データファイルとなる．

2.3.4 Reduce 処理の詳細

Reduce フェーズでは，Map フェーズで生成された中間データに対して集約処理を行い，最終的な結果を出力する．Reduce フェーズの処理の流れを図 4 に示す．

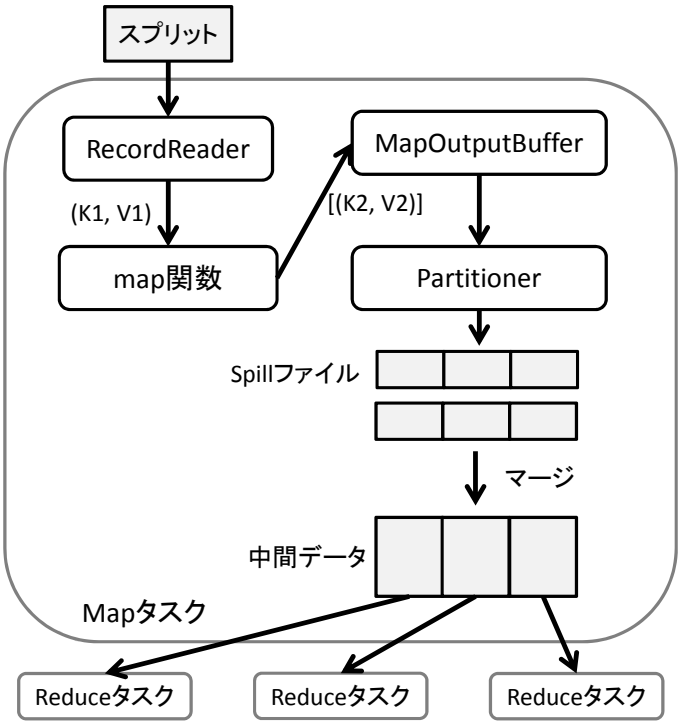

図 3: Map タスクの流れ

Reduce タスクでは中間データのうち，そのタスクに対応するパーティションを入力として処理する．そのため，まず必要なパーティションを自分のノードにコピーするシャッフル処理を行う．Map タスクが実行されていたノードから HTTP 通信を利用してパーティションがコピーされる．Reduce タスクは，デフォルトで全 Map タスク数のうちの 5%が完了した時点で実行が開始される．そのため Map フェーズと Reduce フェーズは同時に実行されている期間があり，その間は完了した Map タスクの中間データのシャッフルを行う．全ての必要なパーティションのコピーが完了すると，これらのパーティションは Key によってソートされマージが行われる．ここでも，同一の Key を持つ複数の Key-Value ペアは，1 つの Key と複数の Value からなる Key-Values ペアに変換される．マージされた中間データは Key-Values ペアごとに reduce 関数に入力され集約処理が行われ，最終的な結果となる Key-Value ペアは RecordWriter へ出力される．RecordWriter は，reduce 関数の出力する Key-Value ペアを全て集めると，HDFS 上にファイルとして出力する．

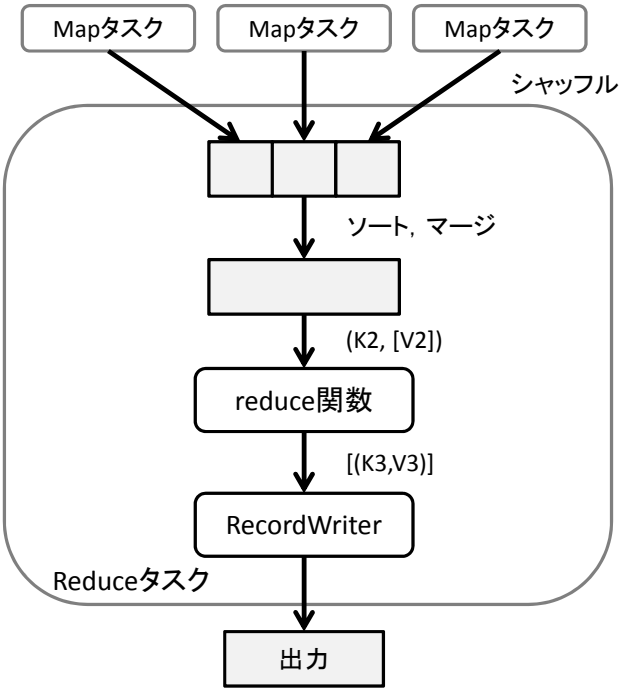

図 4: Reduce タスクの流れ

2.4 Hadoop のタスクスケジューリング

前述のとおり Hadoop MapReduce では，ユーザから投入されたジョブは，複数の Map タスクおよび Reduce タスクに分割され，JobTracker によって TaskTracker にスケジューリングされることで実行される．Hadoop では，JobTracker と TaskTracker は定期的に Heartbeat 通信を行っている．Heartbeat はそれぞれの TaskTracker が JobTracker に向けて実行しているタスクの進捗や TaskTracker の状況を報告し，それに対して JobTracker が HeatbeatResponse という応答を送信するという形で動作する．TaskTracker は，自身がタスクを実行できる状態にある場合，Heartbeat 通信によって JobTracker にタスクを要求する．JobTracker が選択したタスクを HeartbeatResponse に格納し TaskTracker に送信することでタスクをスケジューリングする．

Map タスクと Reduce タスクでは，異なったスケジューリング手法が取られている．以下でそれぞれについて述べる．

2.4.1 Map タスクのスケジューリング

Map フェーズでは，HDFS 上の入力ファイルを処理して中間データを出力する．Map タスクは HDFS によってクラスタ上の各ノードに分散配置されている入力ファイルを分割したスプリットに対し処理を行う．Hadoop のスケジューラは，タスクを要求してきた TaskTracker に，そのノードに配置されているスプリットに対応する Map タスクをスケジューリングしようと試みる．このようにスケジューリングされたタスクは data-local タスクと呼ばれ，タスクが実行されるノードに入力スプリットが存在しているため，ローカルディスクから入力データを読み込んで処理を行うことができる．

もし，タスクを要求した TaskTracker のノードに処理可能なスプリットが存在しない場合は，そのノードが属しているラック内に存在するスプリットに対応する Map タスクをスケジューリングしようと試みる．このようにスケジューリングされたタスクは rack-local タスクと呼ばれる．rack-local タスクも存在しない場合は，そのラックの外まで範囲を広げてスプリットを探し，タスクをスケジューリングする．このようなタスクは non-local タスクと呼ばれる．rack-local タスクと non-local タスクがスケジューリングされると，TaskTracker がそのスプリットを処理する際，自身のノードにスプリットが存在しないためネットワークを介してそのスプリットが配置されているノードからスプリットをコピーする必要があり，通信のオーバヘッドが発生する．

よって，Hadoop はネットワークを介したスプリットのコピーのオーバヘッドが少なくなるよう，data-local タスク，rack-local タスク，non-local タスクの順で Map タスクをスケジューリングする．

2.4.2 Reduce タスクのスケジューリング

Hadoop における Reduce タスクのスケジューリングは，非常に単純である．TaskTracker が Heartbeat 通信時にタスクを要求したとすると，JobTracker は未実行の Reduce タスクを一つ取り出し，その TaskTracker にスケジューリングする．

2.5 Hadoop の Reduce タスクのスケジューリングの問題点

Hadoop では Reduce タスクをスケジューリングする際には，タスクを要求してきた TaskTracker に無条件に Reduce タスクが割り当てられる．Reduce タスクのスケジューリングは，Map タスクのスケジューリングにおいて data-local タスクから割り当てるといった，実行に効果的であるデータローカリティを考慮したスケジューリングではない．Reduce タスクの入力データは，全ての Map タスクがそれぞれ出力したその Reduce タスクに対応するパーティションという中間データの集まりである．そのため，Reduce タスクを実行するには，Map タスクの実行された全てのノードから必要なパーティションをネットワークを介してシャッフル，つまりコピーする必要がある．しかし，それぞれのノードが保持しているパーティションの量が均等であるという保証はなく，偏りが生じる場合がある．この偏りは，ジョブの処理内容に依存する．Hadoop の Reduce タスクのスケジューリング手法では，パーティションサイズのばらつきによっては，不必要により多くのデータをシャッフルしなければいけない状況が発生し得る．例えば，ある Reduce タスクが必要としているパーティションの 9 割が一つのノードにある場合，そのノードに Reduce タスクをスケジューリングするとローカルディスクから 9 割のパーティションが得られるが，別のノードにスケジューリングした場合は，9 割のパーティションをネットワークを介してシャッフルする必要があり時間を要する．さらに，ネットワークの帯域も浪費される．

また，ネットワークの状況を考慮していないという問題点も指摘できる．一般的にクラスタはラック構成となっていると考えられる．ラック構成の場合は，ラック間の通信帯域はラック内の複数のノードによって共有されるため，ラック内の通信帯域に比べ制限される．そのため，ラック間でパーティションを多くシャッフルする必要があるような Reduce タスクのスケジューリングが行われた場合，パフォーマンスが低下することが予想される．さらに，ネットワーク負荷の高いノードおよび負荷の高いスイッチに接続されるノードはネットワークパケットの欠落や遅延が発生することがある．このようなノードにタスクをスケジューリングすることもパフォーマンスを低下させる原因となる．

3 既存研究

Hadoop の Reduce タスクのスケジューリング手法の改良に関する既存研究として，CoGRS[4] がある．CoGRS(Center-of-Gravity Reduce task Scheduler) は，各ノードが Reduce タスクの実行に必要なパーティションをどれだけ保持しているかによって重心ノードを定義している．重心ノードは，パーティションをコピーするノード間のネットワーク距離とコピーされるデータ量から計算される，シャッフルに要するコストが最小となるノードである．ネットワークの距離は Hadoop のネットワークトポロジの設定から得られる．CoGRS では，この重心ノードに Reduce タスクをスケジューリングすることにより，ラック間のシャッフルデータ転送量を削減する．

重心ノードを算出するために，Total Network Distance(TND) を定義しており，これは，Reduce タスク R が実行に必要としているパーティションを保持しているノードとタスクをスケジュールしようとする対象のノードとのネットワーク距離の総和としている．図 5 を用いて説明する．

例えば，TT1 と TT2 が Reduce タスク R の実行のために必要なパーティションを保持しているとする．この時，JobTracker に Reduce タスクを要求している TaskTracker が TT1，TT2 および TT4 であるとする．ネットワーク距離をノードやスイッチ間のリンクを 1 として計算する．TT1 もしくは TT2 に R をスケジューリングした場合は，片方のパーティションのみをコピーすればよいので，TND は 2 となる．一方，TT4 にスケジューリングした場合は，それぞれのパーティションを別のラックからコピーする必要があり TND は 8 となる．

CoGRS では，TND をもとに Reduce タスクごとに重心ノードを求める．Reduce タスク R の重心ノードは，パーティションのシャッフルに要するコストが最小となるノードであるので，R のパーティションを保持しているノードの中から選ばれる．R のパーティションを持っていないノードで R を実行した場合，全てのパーティションをコピーする必要がありコストが大きいからである．R のパーティションを保持しているノード全てに対して，パーティション P の重さによって重み付けした Total Network Distance(WTND) を計算する．P の重さ w は，R の実行に必要な全パーティションサイズに対するそれぞれのノードの保持するパーティション P の割合で定義される．

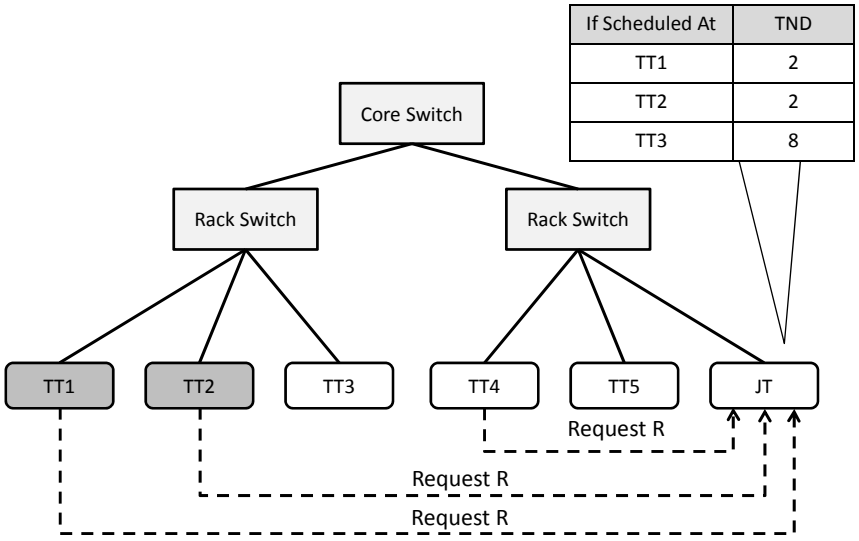

図 5: TT1 と TT2 が Reduce タスク R の実行に必要なパーティションを持っている場合 (TT=TaskTracker,JT=JobTracker).

WTND は次式で計算される．

$$WTND\ Per\ Reduce\ Task\ R = \sum_{i=0}^{n} ND_{iR} \cdot w_i$$

ここで，n は R が必要としているパーティションの数，ND_{iR} はパーティション i のシャッフルに関するノード間のネットワーク距離，w_i はパーティション i の重さを表す．CoGRS は，それぞれのノードついてに計算された WTND を比較し，WTND が最も小さいノードをシャッフルのコストが小さい重心ノードとする．

全ての未実行の Reduce タスクに関して重心ノードの計算を行った後，タスクを要求してきた TaskTracker が重心ノードと計算されたタスクの集合から 1 つのタスクを選びスケジューリングする．このとき，その TaskTracker が最も多くのパーティションを保持しているタスクが選択される．一方，タスクを要求した TaskTracker が重心ノードとなる未実行タスクが存在しない場合は，それぞれのタスクの重心ノードの TaskTracker の空き Reduce スロット数を確認する．空き Reduce スロット数は，それぞれの TaskTracker がその時点で実行可能な Reduce タスクの数で，タスクを開始すると 1 減少し，終了すると 1 増加する．重心ノードの TaskTracker に空き Reduce スロットが存在しないタスクがある場合，つまり現時点でその TaskTracker が Reduce タス

クを追加で実行できないタスクがある場合は，そのタスクのうちでタスクを要求した TaskTracker に最もネットワーク距離が近い TaskTracker が重心ノードとなるタスクを代わりにスケジューリングする．

CoGRS は，Reduce タスクの必要としているパーティションサイズと，そのノードからパーティションをコピーを行う際のネットワーク距離をもとにデータローカリティを考慮したスケジューリングを行う．ネットワーク距離には，ユーザが Hadoop に静的に設定されたネットワークトポロジを利用する．そのためネットワークトポロジが正しく設定されていない環境では，期待した動作が行われないことが考えられる．また，クラウドコンピューティング環境で Hadoop を利用するなど，ネットワークトポロジを把握できない場合も同様である．

4 提案と実装

2.5 節で述べたとおり，Hadoop では，Reduce タスクのスケジューリングは単純にタスクを要求する TaskTracker に Reduce タスクをスケジューリングするというものである．この手法は，データローカリティやネットワークの状況を考慮しておらず，パフォーマンスが低下する可能性がある．そこで本論文では，Hadoop のパフォーマンス向上のために，ネットワーク遅延を測定することによりクラスタ内のネットワークの状況を動的に把握し，Reduce タスクをスケジューリングする手法を提案する．

4.1 提案手法

Hadoop を実行するクラスタ内の各ノード間の往復遅延を測定することによりネットワークの状況を把握し，Reduce タスクのスケジューリングに測定した遅延情報を利用する．ネットワークの遅延を定期的に測定することによってネットワークの負荷状況を適宜把握し対応する．ネットワークに負荷がかかっている場合，他に比べ大きな遅延が測定されるため，そのようなノードへは Reduce タスクを割り当てる優先度を下げる．それは，Reduce タスクでは，実行に必要なパーティションをクラスタ全体から

コピーするためネットワークの負荷の高いノードに割り当ててしまうと，シャッフルに時間がかかってしまうためである．

また，Hadoop は Amazon EC2[5] に代表されるクラウドコンピューティング環境上で運用されることも多い．このような環境では，VLAN や SDN といった仮想ネットワーク技術が利用されており，実際のネットワークトポロジを把握し Hadoop に設定することは困難である．2 つのノードが別のラックに格納され，ノード間の距離が物理的に大きい場合などは，スイッチを複数経由し遅延は大きくなると考えられるため，この遅延を利用することでネットワークトポロジを把握できない環境でもデータローカリティを考慮したスケジューリングが可能となる．

4.1.1 Reduce タスクのスケジューリング

本手法は，既存手法である CoGRS を独自実装したものをベースとし，ネットワーク距離を利用していた部分に測定した往復遅延を用いる．CoGRS と同様に Reduce タスク R のパーティションの重さを用い，重み付けした Total Delay(WTD) を以下の式で定義し，パーティションを持っているノードに関してそれぞれ計算する．

$$WTD\ Per\ Reduce\ Task\ R = \sum_{i=0}^{n} RTT_{iR} \cdot w_i$$

ここで，n は R が必要としているパーティションの数，RTT_{iR} はパーティション i のシャッフルに関するノード間のネットワーク往復遅延，w_i はパーティション i の重さを表す．本手法では，WTD がもっとも小さいノードに Reduce タスク R の重心ノードと定義する．ただし，重心ノードは必ず Reduce スロットが空いているものとし，計算された重心ノードのスロットが空いていない場合は次に WTD が小さいノードを重心ノードとする．そして，タスクを要求した TaskTracker には，その TaskTracker のノードが重心ノードであるタスクが存在する場合のみ，その中で最も多くのパーティションを保持しているタスクをスケジューリングする．

この実装は CoGRS のものとは異なっている．CoGRS と同様の実装を行った場合，ネットワークの負荷を考慮しないスケジューリングが行われ，Hadoop のパフォーマンスが低下する可能性があるからである．CoGRS では，まずタスクを要求した TaskTracker

が重心ノードとなるタスクを探し，存在すればその中から最もその TaskTracker が多くパーティションを持っているタスクをスケジューリングする．要求した TaskTracker が重心ノードとなるタスクが存在しなければ，重心ノードの Reduce スロットが空いておらず，その時点で重心ノードにスケジューリングできないタスクの中で，重心ノードとタスクを要求した TaskTracker のノードとの距離が最も小さいタスクを代わりにスケジューリングする．既存手法の距離を利用する部分を往復遅延を利用するように単純に変更した場合を考える．重心ノードとなれるタスクが存在しなかった場合，重心ノードのスロットが空いていないタスクの中でタスクを要求した TaskTracker と重心ノードとの間の往復遅延が最も短いタスクを代わりにスケジューリングすることになる．もし，ネットワーク負荷の高いノードの TaskTracker がタスクを要求し重心ノードとなるタスクが存在しなかったとすると，その負荷の高い TaskTracker の WTD に関係なく他のタスクが代わりにスケジューリングされ，パフォーマンスが低下してしまう．そのため前述のように，重心ノードの Reduce スロットが空いていない場合は，WTD が小さい順に重心ノードを決定することで，ネットワーク負荷が高いような WTD が大きな TaskTracker にタスクが割り当てられないようにしている．

4.1.2 遅延の測定

各ノード間のネットワーク遅延は，TCP の通信により測定する．ICMP パケットを利用し遅延を測定する手法を検討したが，Hadoop が記述されている言語である java では ICMP を直接扱えないことや，UNIX の ping コマンドを利用すると測定に時間がかかるといった問題があったため TCP を利用することとした．1 回の遅延測定で 1 往復の測定値を利用する場合は精度が低いため，複数回の測定の平均を利用することとしているが，ping コマンドを利用する場合，ICMP パケットの送信間隔は最小でも 200ms と制限されてしまい測定に時間がかかってしまう．これに対し，java プログラムから直接 TCP パケットを用いて往復遅延を測定する方法では，高速に複数回の測定を行うことができ，精度の高い値が得られる．さらに，TCP は Hadoop の通信に用いられているため，ICMP より実際の通信に近い測定が可能であると考えられる．

TCP での測定において，接続を確立後にクライアントがサーバにリクエストを送信

した時刻から，サーバからのレスポンスを受信した時刻までの間を往復遅延として利用する．図 6 に，遅延測定のシーケンスを示す．クライアントはサーバに対して TCP のコネクションを確立した後，SETUP メッセージを送信する．サーバはクライアントに対して応答できる状態となると，READY メッセージを返答することで遅延測定可能な状態になる．クライアントは現在の時刻を取得，記憶し，直後にサーバに REQUEST メッセージを送信する．これを受けたサーバは RESPONSE メッセージを返答し，それを受信したクライアントはその時の時刻を取得する．通信前後に取得した時間の差を計算し，サーバクライアント間の往復遅延とする．複数回測定する場合は，この REQUEST メッセージと RESPONSE メッセージによるやり取りを繰り返す．測定を終了する場合は，クライアントから FIN メッセージを送信し，サーバも FIN メッセージで応答した後にコネクションを切断する．

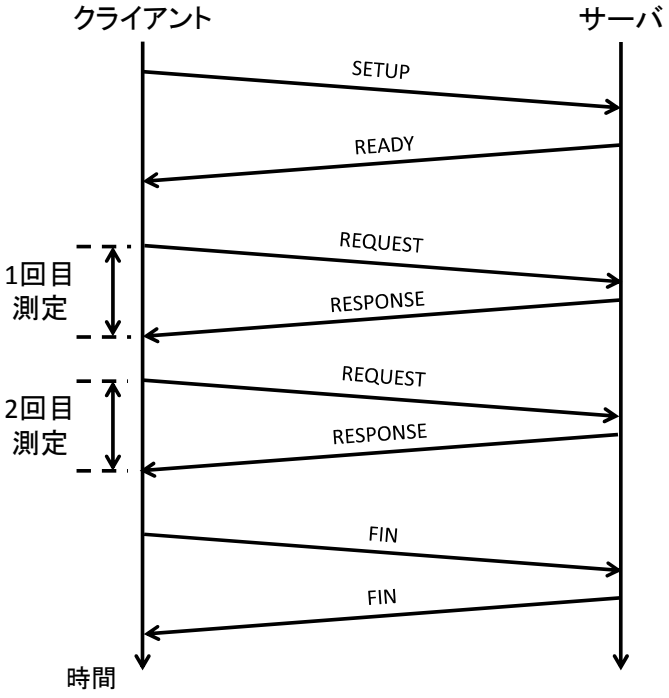

図 6: TCP を用いた遅延測定のシーケンス（2 回測定する場合）

4.2 実装

提案手法を，Hadoop バージョン 1.0.3 に実装した．提案手法を実装した Hadoop の動作を図 7 に示す．追加実装した部分は，JobTracker が定期的に TaskTracker に遅延測定を要求する部分，測定された遅延情報を管理する部分，および TaskTracker 間で往復遅延を測定する部分である．

4.2.1 JobTracker への実装

TaskTracker へ遅延測定を要求する機構を JobTracker に実装した．この機構には，TaskTracker が JobTracker に対して行なっている Heartbeat 通信を利用した．Heartbeat 通信は，JobTracker がそれぞれの TaskTracker に対し死活監視を行うための通信であり，TaskTracker が JobTracker に向けて通信を行い JobTracker が応答するというように動作する．Heartbeat 通信は実際には RPC によって実装されており，TaskTracker が JobTracker のメソッド heartbeat をコールすることで行われる．実行中のタスクや TaskTracker の状況を表す TaskTrackerStatus を引数に渡し，その TaskTracker にスケジューリングするタスクなどの情報である HeatbeatResponse を戻り値として返すことによりお互いの情報を交換している．

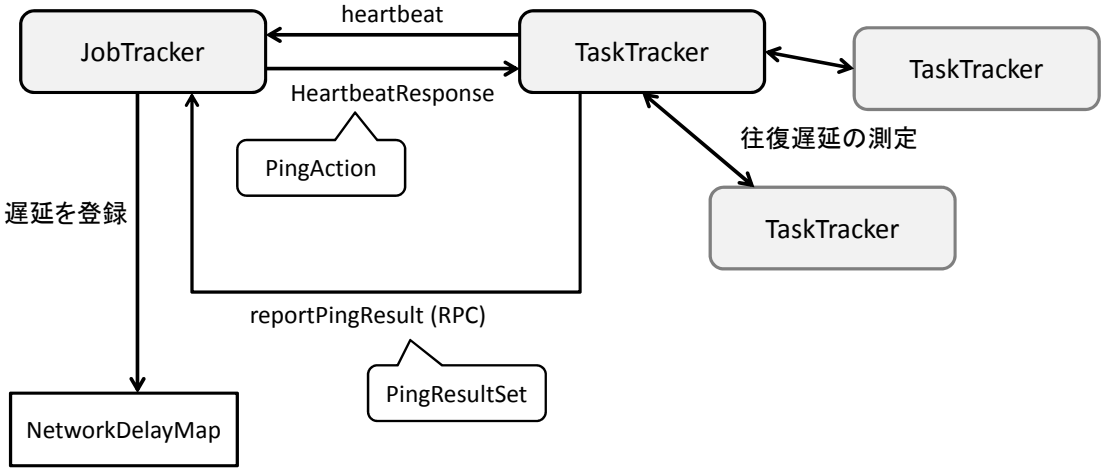

図 7: 提案を実装した Hadoop の往復遅延測定の動作

JobTracker の起動時に開始される新たなスレッドを実装し，そのスレッドにより定期的に TaskTracker に遅延測定を要求するようにした．この要求は，PingAction というクラスを定義し，これを HeartbeatResponse に格納することで TaskTracker へ伝達する．PingAction には，その TaskTracker が遅延測定を行うべき相手ノードの IP アドレスが格納されている．クラスタ内のすべての TaskTracker ノード間の遅延が測定されるように JobTracker は，PingAction を生成する．

それぞれの TaskTracker は遅延測定を終えると，結果を JobTracker へ伝達するために RPC を利用し，JobTracker の reportPingResult メソッドを呼び出す．このメソッドでは，引数で渡された測定結果 PingResultSet から遅延データを取り出し，JobTracker 自身の保持している表 NetworkDelayMap に格納し管理する．

4.2.2 TaskTracker への実装

提案手法では，TaskTracker は TCP 通信を用いて他の TaskTracker との間の往復遅延を測定する．そこで，まず TaskTracker に PingServer というサーバプログラムを実装した．PingServer は TaskTracker 起動時に生成されるスレッドとして動作を開始し，後述するクライアントプログラム PingClient からの接続を待つ．クライアントから接続されると別スレッドを生成し，クライアントからの REQUEST メッセージに対して RESPONSE メッセージを返答する（図 6 参照）．

Heartbeat 通信時に JobTracker が TaskTracker に遅延測定を要求した場合，TaskTracker では HeartbeatResponse から PingAction が得られる．PingAction にはその TaskTracker が遅延を測定すべき相手ノードの IP アドレスが記されているので，それにしたがって測定を行う．遅延測定は，スレッド PingThread を実装し，このスレッドによって行われる．PingAction を受け取った場合，記されている IP アドレスの TaskTracker に対して遅延測定を行うクライアント PingClient を生成する．PingClient は ping メソッドを実装しており，これを呼び出すと PingServer に対して接続を行い遅延を測定し，戻り値として返す．ping メソッドの引数に整数を指定することで測定回数を指定でき，その回数測定した往復遅延の平均値が返ってくるようになっており，デフォルトでは 20 回としている．測定回数が少ない場合は，平均値にばらつきが多く見られ

たので，平均値が安定し，且つ測定に大きな時間がかからない回数として，経験的に20回を選んだ．PingActionに記されているすべてのノードとの間の遅延を測定すると，TaskTrackerは結果を格納するPingResultSetを生成しJobTrackerのreportPingResultメソッドをRPCによって呼び出し，結果を送信する．

4.2.3 Reduceタスクスケジューラの実装

Nativeの Hadoopでは，JobTrackerはTaskTrackerからHeatbeat通信時にタスクを要求された場合，実行中のジョブを表すクラスJobInProgressのobtainNewReduceTaskメソッドを呼び出す．obtainNewReduceTaskメソッドは，findTaskFromListメソッドにより未実行のReduceタスクのリストからReduceタスクを取り出し，それを返す．JobTrackerは返されたReduceタスクを，LaunchTaskActionクラスのインスタンスに格納する．最後にTaskTrackerからのHeartbeat通信の応答として，HeartbeatResponseにLaunchTaskActionを格納して，TaskTrackerに返すことでReduceTaskをスケジューリングしている．

このスケジューリング部分を変更し，実装した遅延測定機構により得られた往復遅延の値を用いて，Reduceタスクをスケジュールするスケジューラを実装した．Reduceタスクのスケジューリング手法は既存研究であるCoGRSを独自実装し，ネットワーク距離を重心ノードの計算に利用する代わりにJobTrackerが管理するネットワークの遅延値を利用する．Nativeの Hadoopの実装であるfindTaskFromListメソッドを呼び出す部分を，新たに実装したfindReduceTaskByNetworkDelayメソッドを呼び出すように変更した．findReduceTaskByNetworkDelayメソッドでは，測定した遅延情報をもとにタスクを要求してきたTaskTrackerに割り当てるReduceタスクを選択する．未実行のReduceタスク全てについて重み付けしたTotal Delay(WTD)を計算し，重心ノードを求める．タスクの重心ノードは，そのReduceタスクに対応するパーティションを保持しているTaskTrackerに対してそれぞれWTDを求める．そして，計算されたWTDを昇順でソートし，最もWTDが小さいTaskTrackerのReduceスロットに空きが存在するか確認する．空きが存在する場合は，このタスクを実行可能であるため，そのTaskTrackerを重心ノードとし，空きが存在しない場合は，次にWTDが小さい

TaskTracker の Reduce スロットを同様に確認し，重心ノードを求める．全ての未実行 Reduce タスクの重心ノードを算出した後，タスクを要求している TaskTracker が重心ノードとなる Reduce タスクのリストを作成する．その TaskTracker の保持するパーティションの中で最も多く保持しているパーティションに対応する Reduce タスクをリストから選択し，findReduceTaskByNetworkDelay メソッドから返す．

5 評価と考察

実装した提案手法の有効性をベンチマークプログラムを実行することで評価した．

5.1 評価環境

評価環境を表 1 に示す．表中の項目は計算機 1 台あたりのものを示す．JobTracker ノード 1 台と TaskTracker ノード 13 台からなるクラスタ上で評価を行った．ネットワーク環境は，2 つのスイッチを用いて構成されており，JobTracker ノードと TaskTracker ノード 6 台の計 7 台のラック 1 と，残りの TaskTracker ノード 7 台のラック 2 の 2 つのラックから構成されている．各ノードとスイッチは 1Gbps のリンクで接続され，2 つのスイッチは直接 1Gbps のリンクで接続されている．

表 1: 評価環境

	JobTracker	TaskTracker
OS	CentOS 5.7	Ubuntu 11.04
CPU	2 x AMD Opteron 6168 1.9GHz	Intel core i5 750 2.67GHz
メモリ	32GB	8GB
ノード数	1 台	13 台
ネットワーク	1Gbps Ethernet	

5.2 ベンチマークプログラム

提案手法を評価するために，Hadoop でベンチマークプログラムを実行し処理時間を計測した．ベンチマークプログラムとして，Intel の提供する Hadoop 向けベンチマー

クである HiBench[6] の PageRank プログラムを利用した．PageRank は，Web ページのリンク関係を表すグラフから，ページの重要度をランク付けするグラフアルゴリズムである．このプログラムは，Stage1 と Stage2 の 2 つの MapReduce ジョブから構成されており，ページの重要度が収束するまでこの 2 つのジョブを繰り返すイテラティブアプリケーションである．Stage1 の Reduce の出力は Stage2 の Map の入力となり，Stage2 の Reduce の出力は次のイテレーションの Stage1 の Map の入力となる．Stage1 は，現在のイテレーションのそれぞれのページのスコア（ページの重要度）とそのページのリンク関係から，リンク先のページに与えるスコアを計算する．そして，Stage2 でそれぞれのページが Stage1 でリンク元から与えられたスコアを集計し，スコアを更新する．

本評価では，HiBench に付属のツールによりページ数 500 万のデータセットを生成し，PageRank をイテレーション回数を 1 回として実行した．データセットのサイズは約 1GB である．以降の評価において，PageRank の Stage1 と Stage2 タスク数は表 2 の通りとする．Stage1 の Map タスク数は 195，Reduce タスク数は 10，Stage2 の Map タスク数は 10，Reduce タスク数は 10 としている．Stage1 の Map タスク数は，処理の適度な分散を考慮しクラスタの TaskTracker の数である 13 の整数倍としている．Reduce タスクの数は，一般的に TaskTracker の数以下に設定し，今回は 10 としている．これは，クラスタサイズと同じにした場合，全てのノード間でシャッフルが行われ，スケジューリングの効果がわかりづらくなると考えたためである．Stage2 の Map タスクと Reduce タスクの数は，Stage1 の Reduce タスクの数によって，ベンチマークプログラムが決定している．

表 2: ジョブのタスク数

	Stage1	Stage2
Map タスク数	195	10
Reduce タスク数	10	10

5.3 ネットワークの負荷に基づくスケジューリング

提案手法は，各ノード間の往復遅延を測定することによりネットワークの負荷を考慮し Reduce タスクのスケジューリングを行う．そこで，まず最初にネットワークの一部に負荷をかけた状態で Hadoop で PageRank を実行し，本手法によるスケジューリングの有効性を評価した．Hadoop を実行するバックグラウンドでは，クラスタを構成するノード間で通信を行いネットワークに負荷を与えた．負荷生成を行うノードは，ラック 2 のノードを利用し，同一ラック内のノードを相手に通信を行う．この送受信を行うノードを 0 台，2 台，4 台，6 台と変化させながら，Native，CoGRS，および提案手法を実装したそれぞれの Hadoop でそれぞれ PageRank プログラムを実行し，処理時間を測定した．評価結果を図 8 に示す．

グラフは，Native，既存手法 CoGRS，提案手法を実装した Hadoop で PageRank を 5 回ずつ実行した際の処理時間の平均値を示し，5 回の測定の最大値と最小値の間の範囲を誤差表示によって示している．横軸は，バックグラウンドでネットワークに負荷をかけているノードの数をである．ノード数 0 は，バックグラウンドトラフィックを生成しておらず，ノード数 2 では 1 ノードが送信ノード，もう 1 ノードが受信ノードとして動作している．ノード数 4,6 も同様に送信ノード，受信ノードが半数ずつ動作している．縦軸は，それぞれの Hadoop における PageRank の処理時間である．

Native と比較して CoGRS と提案手法は，全てのバックグラウンド通信ノード数において処理時間が削減されている．CoGRS と提案手法を比較すると，バックグラウンド通信ノード数が 0 の場合を除いて，提案手法の方が CoGRS より処理時間が削減されていることがわかる．通信ノード数が 0 の場合は，CoGRS に比べ提案手法は処理時間が約 0.1%増加し，通信ノード数が 2，4，6 の場合は，処理時間がそれぞれ約 7%，5%，16%削減している．

5.4 トポロジを把握できない環境下での有効性

提案手法では，クラウドコンピューティング環境上などの Hadoop を実行するクラスタのネットワークトポロジを把握できない環境においても，測定された遅延値をもと

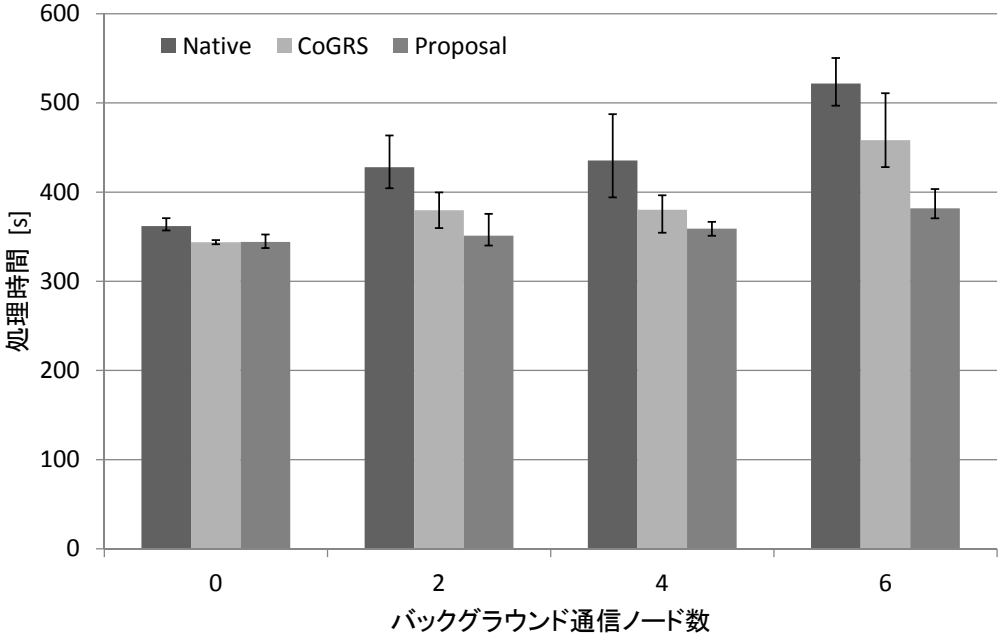

図 8: ネットワークの負荷による PageRank の処理時間

にローカリティを考慮したスケジューリングを可能とすることを目指す．そこで，トポロジを把握できない状況における本手法の有効性を評価するために，Hadoop にネットワークトポロジの設定を行わずに PageRank プログラムを用いた評価を行った．Hadoop は，ネットワークトポロジが設定されなかった場合は，全てのノードが default-rack と呼ばれる同一のラックに属しているものとして扱う．そのためこの評価では，Hadoop は実際は 2 つのラックから構成されている評価環境を，全ノードが 1 つのラックに属している環境として認識する．この場合，CoGRS で利用している 2 ノード間のネットワーク距離は，どのような 2 つのノード間の距離は 2 となり，有効に働かないと考えられる．トポロジを設定しない状態で Native，CoGRS，および提案手法を実装した Hadoop でそれぞれ PageRank プログラムを実行し，処理時間を計測することにより評価した．評価結果を図 9 に示す．

グラフは Native，既存手法 CoGRS，提案手法を実装した Hadoop で PageRank を実行した際の処理時間を示している．提案手法は，Native の処理時間に比べると削減されているが，CoGRS と比べた場合は増加している．Native に対しては約 8 %の処理時

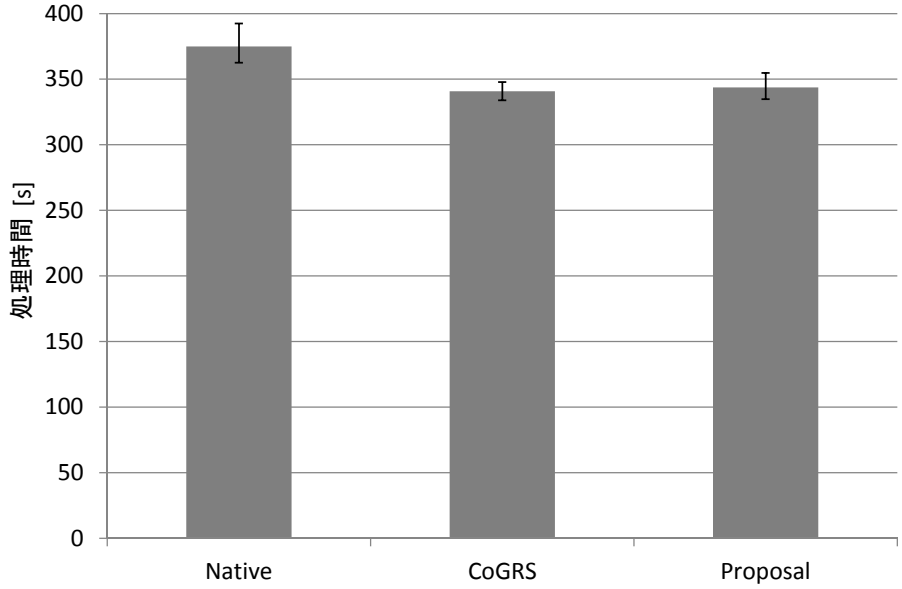

図 9: トポロジを設定しない場合の PageRank の実行時間

間削減，CoGRS に対しては約 1%の処理時間増加となった．

5.5 考察

全ての場合において，Native に比べ既存手法 CoGRS，提案手法を実装した Hadoop の処理時間は削減されている．これは，Reduce タスクのスケジューリングにデータローカリティが考慮されているからである．図 8 のネットワークの負荷に対する評価結果では，バックグラウンド通信ノード数が 0 の場合を除いて CoGRS に比べ，提案手法は処理時間が削減されている．これは，本手法で測定された遅延情報により負荷の高いノードへの Reduce タスクの割り当てが少なくなっているためだと考えられる．バックグラウンド通信ノード数が 0 の場合では，各ノード間の遅延がほぼ均一となっており，提案手法による効果があまり現れなかったといえる．一方，バックグラウンド通信ノード数が 6 の場合に，提案手法は Native，CoGRS の両方に対して最大の処理時間削減率となっており，それぞれ約 27%，16%となった．ネットワーク負荷の高いノードが多く存在しており，その結果 Native や CoG ではそのようなノードに Reduce タスクがスケジューリングされることが多くなるが，提案手法では測定された遅延情

報を用いて，そのような遅延の大きいノードへのスケジューリングを避けるため，より効果が現れたと考えられる．

図 9 のトポロジを把握できない環境下での評価結果では，CoGRS に比べ処理時間が約 1%長くなっている．これは提案手法にオーバヘッドがあると考えている．提案手法は，タスクを要求した TaskTracker が重心ノードとなる場合にのみ Reduce タスクをスケジューリングし，ネットワーク負荷の高いノードの TaskTracker にスケジュールしないようにしている．一方，既存手法 CoGRS では，タスクを要求した TaskTracker が重心ノードとなるタスクが存在しなかった場合，重心ノードで実行できないタスクを探し，そのタスクをその TaskTracker に代わりにスケジューリングする．この実装の違いにより，提案手法は WTD の小さい順に Reduce タスクをスケジューリングできるが，CoGRS に比べ全ての Reduce タスクをスケジュールするまでに要する時間が長く必要になる場合がある．CoGRS では，ある TaskTracker が Heartbeat 通信によりタスクを要求し，重心ノードとなるタスクが存在しなかった場合に，直ちに代わりのタスクをスケジューリングするのに対し，提案手法ではタスクをスケジューリングしない．そして，次の Heartbeat 通信が行われた際に再び重心ノードを計算する．この 2 回の Heartbeat 通信までの間に，他の TaskTracker の空き Reduce スロットがなくなり，重心ノードを計算し直すことでタスクを要求した TaskTracker が重心ノードとなるタスクが存在している可能性があり，存在した場合そのタスクをスケジューリングする．提案手法では，何度も重心ノードを計算し直さなければタスクがスケジューリングさず，全てのタスクをスケジューリングするのに時間を要する場合がある．

実際に，Stage1 の全ての Reduce タスクをスケジューリングするのに要した時間を調べると，既存手法では平均で 7.2 秒であったが，提案手法では 28.8 秒であった．Stage2 に関しては，CoGRS，提案手法ともに 3.9 秒であった．Stage1 の Reduce タスクを全てスケジューリングする時間が CoGRS に比べ提案手法は 4 倍以上要していることが確認できた．Stage2 では差が見られなかったが，これは PageRank の Stage2 の処理の特徴によるためである．Stage2 の Reduce タスクの数は Map タスクの数と同じであり，それぞれの Map タスクは 1 つの互いに異なるパーティションの中間データのみを生成するため，Map タスクと Reduce タスクが 1 対 1 で対応する．各 Reduce タスクの重心

ノードは対応する Map タスクが実行されていた TaskTracker のノードとなり，これは全て別のノードとなるので，Reduce スロットの空きがなくなる状況が発生しない．そのため，提案手法でも全ての TaskTracker が 1 回ずつ Reduce タスクを要求するだけでスケジューリングが完了するので，既存手法とスケジューリングに要する時間は同じとなる．

Stage1 における大きなオーバヘッドは，遅延測定結果の精度も影響していると考えている．現在，1 回の遅延測定において，複数回の測定の平均値を結果として利用しているが，それでも測定結果にばらつきが見られた．PageRank の Stage1 の Map タスクの出力する中間データは，いずれのパーティションサイズも同程度で偏りがないため，重心ノードを求める際に計算する WTD の値は，遅延の値に大きく影響を受けることになる．もし，他に比べ遅延の値が小さく計測されたノードがあった場合，多くのタスクがそのノードを重心ノードとすることが予測される．重心ノードが同じタスクが多く存在すると，WTD が小さい順に Reduce タスクをスケジューリングする際に重心ノードの再計算が多く行われ，全ての Reduce タスクをスケジューリングするのに大きな時間が必要になると考えられる．

6 今後の課題

本研究では，クラスタ内のノード間の往復遅延を測定し，その情報を利用するタスクスケジューラの実装を行った．しかし，現時点では Reduce タスクのみをスケジューリングの対象としているため，Map タスクは Native の Hadoop のタスクスケジューリング手法が適用され，ネットワークの負荷が考慮されない．もし，ネットワーク負荷の高いノードにスケジューリングされた Map タスクが，ある一つのパーティションを多く出力するような偏った中間データを生成した場合，そのパーティションに対応する Reduce タスクもデータローカリティによりその負荷の高いノードにスケジューリングされてしまう．そのため，遅延情報を利用したスケジューリングを Map タスクにも適用する必要がある．

クラウドコンピューティング環境上などのネットワークトポロジを把握できない場合を想定した評価では，提案手法の有効性が示されなかった．本手法の Reduce タスク

のスケジューリングにかかかるオーバヘッドが確認されたため，これを削減することは今後の課題である．

また，現在の遅延測定手法では複数回の測定を行ない，平均化した場合でも結果にばらつきが見られるので，遅延測定手法を改善することや，クラスタ内のボトルネックリンクが特定し，よりネットワークの状況をスケジューリングに反映するために，ネットワークの可用帯域推定手法も検討したい．さらに発展した内容としては，遅延の推移や実行されるジョブの特徴などをプロファイリングするといった高度なスケジューリング手法を検討し，更なる Hadoop の高速化を目指したい．

7 まとめ

本研究では，Hadoop のパフォーマンスの向上を目的として，ネットワークの往復遅延を測定しネットワークの状況を考慮した Reduce タスクスケジューリング手法を提案した．提案手法の有効性を確認するために，クラスタのネットワークに負荷をかけた状態で HiBench ベンチマークの Pagerank プログラムを実行し，処理時間の評価を行った．評価の結果から，多くの場合において，提案手法はジョブの実行時間を削減できており，Reduce タスクスケジューリング手法のベースとした既存手法 CoGRS と比較し，最大で 16%の処理時間を削減した．今後の課題として，Map タスクスケジューリングに遅延を本手法をに適用することや，遅延の測定精度の向上などが考えられる．

謝辞

本研究を進めるにあたり多大な尽力を頂き，日頃から熱心な御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授，梶岡慎輔助教，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，日常の議論を通じて多くの知識や示唆を頂いた松尾・津邑研究室，齋藤研究室，松井研究室の皆様，特に研究に関して貴重な意見を頂いた藏澄汐里氏，水野航氏，曽我恵里氏に深く感謝致します．

参考文献

[1] Hadoop. available online on `http://hadoop.apache.org/`.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system, 2003.

[4] M. Hammoud, M.S. Rehman, and M.F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pp. 49 –58, jun 2012.

[5] Amazon elastic compute cloud（amazon ec2）. available online on `http://aws.amazon.com/jp/ec2/`.

[6] Hadoop benchmark suite (hibench). available online on `https://github.com/intel-hadoop/HiBench`.