平成24年度　　卒業研究論文

データ通信の最適化を目指した HadoopMap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21年度入学　21115081番

曽我恵里

平成25年2月12日

目 次

1 はじめに

ブログ・SNS・Twitterなどインターネット上における多種多様なサービスの展開と，インターネット利用者の爆発的な増加により，世界中でデータ量は膨大に増え続け，それに伴いデータの処理も大規模化している．膨大なデータを高速に処理できる性能の高いコンピュータは高価である．また，コンピュータ1台を使用してサービス全体を管理する場合，そのコンピュータの障害発生時にはサービス全体を停止させなければならない．そこで，安価なコンピュータを複数台連携させ冗長構成にすると，障害発生時のサービスの停止を防ぐことができる．さらにデータの保存や処理を分散して行うことで，全体の性能を向上させることができる．

しかし，複数台のコンピュータを連携させ性能を向上させる場合，全てのコンピュータで矛盾が起きない，かつ並列実行や負荷分散，ネットワーク転送とディスク利用の最適化，ノードの故障時の対処，ノード追加による処理性能の拡張などについて実運用に耐えることのできるアプリケーションを開発する必要がある．これらが正常に動作するように，1ユーザや1企業が検討し全て実装し，そしてそれを運用することは容易ではない．このような課題に対し，複数のコンピュータを用いた分散処理における複雑な機能があらかじめ実装されたフレームワークであるHadoopが注目され，利用されるようになった．([1][2][3][4])

Hadoopは大規模データの分散並列処理を支えるプラットフォームである．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と分散ファイルシステムGoogle File Systemのオープンソース実装である．

Hadoopを基盤としているサービスは，単一のユーザに利用されるよりも複数のユーザに同時に利用されることが圧倒的に多いと考えられる．このような状況において，ユーザは他のユーザのサービス利用状況やHadoopの動作状況を考慮していない．そのため，1度に多くのジョブがHadoopが動作するシステムに投入されると複数のジョブでリソースを分け合う必要がある．よって，複数のジョブが投入された場合でも，ユーザが処理を要求してから結果を返されるまでのレスポンスの遅さを感じさせないために，Hadoop内部において，Hadoopからみた実行単位であるタスクのスケジューリングが重要となる．

本論文では，複数のユーザが Hadoop を利用し，複数のジョブが同時に投入された場合にも，性能低下につながるネットワーク通信をできるだけ抑えることで実行時間を削減することを目的として，複数ユーザの Hadoop の利用に適したスケジューリング手法を提案する．Hadoop に実装されているジョブスケジューラの 1 つであるキャパシティスケジューラの拡張機能として，タスクの割り当てを要求したノードに配置されているスプリット数を考慮したスケジューリング手法を実装した．なお，スプリットは MapReduce 処理に対する入力データが Map 処理を行うタスク数分に分割されたものである．そして，複数のユーザが同時にジョブを投入した場合に，ネットワークを介さずに処理されるタスクの増加量と単一ジョブの実行時間についての評価を行った．

本論文では，第 2 章では Hadoop の詳細と，Hadoop に複数ジョブが投入された場合のタスクスケジューリングの問題点を挙げ，第 3 章で本研究の提案とその実装を述べる．第 4 章で性能評価と考察を行い，最後に第 5 章で今後の課題を述べ，本論文をまとめる．

2 研究背景

2.1 Hadoop の概要

Hadoop とは大規模なデータを処理するための並列分散処理プラットフォームである．Hadoop は，信頼性，拡張性を持ちながら，分散処理と分散（データ）ストレージを実現するためのオープンソースソフトウェアであり，Apache Software Foundation のプロジェクトの一つとして公開され，現在も Apache Hadoop プロジェクトにより開発が進められている [5]．

Hadoop は大きく分けて，分散ファイルシステムである「Hadoop Distributed File System(HDFS)[6]」と分散処理を担う「Hadoop MapReduce[7][8]」で構成されており，以下でこの 2 つについて述べる．

2.2 Hadoop Distributed File System

Hadoop Distributed File System(HDFS) とは，Googleが開発した分散ファイルシステム Google File System をオープンソースで実装したものである．

HDFS の管理を行うマスターノードである NameNode と，実際の I/O 処理を行うスレーブノードである複数の DataNode で構成される．HDFS クライアントがファイルを書き込む場合，クライアントは NameNode に問い合わせる．対象のファイルはブロックと呼ぶ固定長に区切られた塊に分割され，NameNode の指示によりクラスタ内の DataNode に分散され，書き込まれる．NameNode はファイルとその位置情報の管理の他に DataNode の監視も行う．HDFS の特徴として大容量，高スループット，耐障害性，スケーラビリティ，信頼性などが挙げられる．HDFS ではブロックを複数のノードにレプリケーションしているが，これにより 1 つのノードが何らかの理由でダウンしブロックを失ったとしても，他のノードに同じブロックが保持されているため，分散ファイルシステム全体としてデータを失う危険性を回避しているのである．

HDFS の利便性は 2 つの観点から説明することが可能である．1 つは透過性である．クライアントの立場から見た場合，ユーザはこのファイルシステムを透過的に扱うことが可能である．すなわち，ファイルシステムの裏側で複数のサーバが動作していることを考慮する必要はなく，ローカルファイルシステムを扱うかのようにファイルに対して命令を発行することができる．もう 1 つは拡張性である．HDFS は分散ファイルシステムを構成するスレーブノードを追加することで基本的な I/O 処理能力（スループット）の向上を図ることができる．よってディスク容量が足りなくなれば，スレーブノードを追加するだけで容量の増加が可能である．

2.3 Hadoop MapReduce

2.3.1 Hadoop MapReduce の概要

Hadoop MapReduce は Googleが開発した並列分散処理フレームワーク MapReduce をオープンソースとして実装したものである．MapReduce はクライアントが実行要求する処理単位であるジョブを，並列分散処理が可能な独立したタスクの集合に分割

し，そのタスクを複数のノードに分散して処理する．MapReduce においての処理は大きく Map フェーズと Reduce フェーズの 2 つに分けられる．この 2 つのフェーズについては 2.3.2 節で詳しく述べる．MapReduce はデータを Key と Value の組み合わせである KeyValue ペアで扱う．MapReduce は JobTracker（マスターノード）と複数の TaskTracker（スレーブノード）で構成される．JobTracker はクライアントからのジョブの受付，TaskTracker へのタスクの割り当て，TaskTracker の管理など，分散処理全体を制御するための動作を担う．一方，TaskTracker は割り当てられたタスクの実行を担う．また TaskTracker においては Map 処理スロットと Reduce 処理スロットがあり，1 つのスロットに対し 1 つのタスクが割り当てられる．

MapReduce の特徴は，データ処理に関して高いスケーラビリティを持つことである．Map 処理や Reduce 処理は，それぞれ処理対象のデータを分割することにより複数のスレーブノードで，同じ処理を並列に実行することが可能である．Map への入力データに分割可能なデータを使用するため，Map 処理を実行するスレーブノードを増やせば，分割したデータをそれらのノードで並列に処理させることができる．このようにして，MapReduce は容易に高いスケーラビリティを得ることができる．

MapReduce のデータフローを図 1 に示す．MapReduce では各 TaskTracker 上の Map 処理スロットと Reduce 処理スロットにタスクが 1 つずつ割り当てられて処理が行われる．Map・Reduce の各フェーズでは入力・出力はいずれも Key と Value のペアのシンプルなデータ構造をとる．

2.3.2 Hadoop MapReduce のデータフロー

Hadoop は，MapReduce ジョブへの入力データを入力スプリット（あるいは単にスプリット）と呼ばれるほぼ固定長の断片に分割する．Hadoop は各スプリットに対して 1 つの Map タスクを生成する．生成された Map タスクでは，ユーザが定義した Map 関数が各レコードに対して実行される．Map 処理が行われたデータは中間データとして出力される．Reduce タスクが 1 つの場合は，Map タスクの出力全てが 1 つの Reduce タスクの入力として集約される．図 1 のように複数の Reduce タスクが存在する場合，Map タスクはその出力を各 Reduce タスクに対応するパーティションに分割する．な

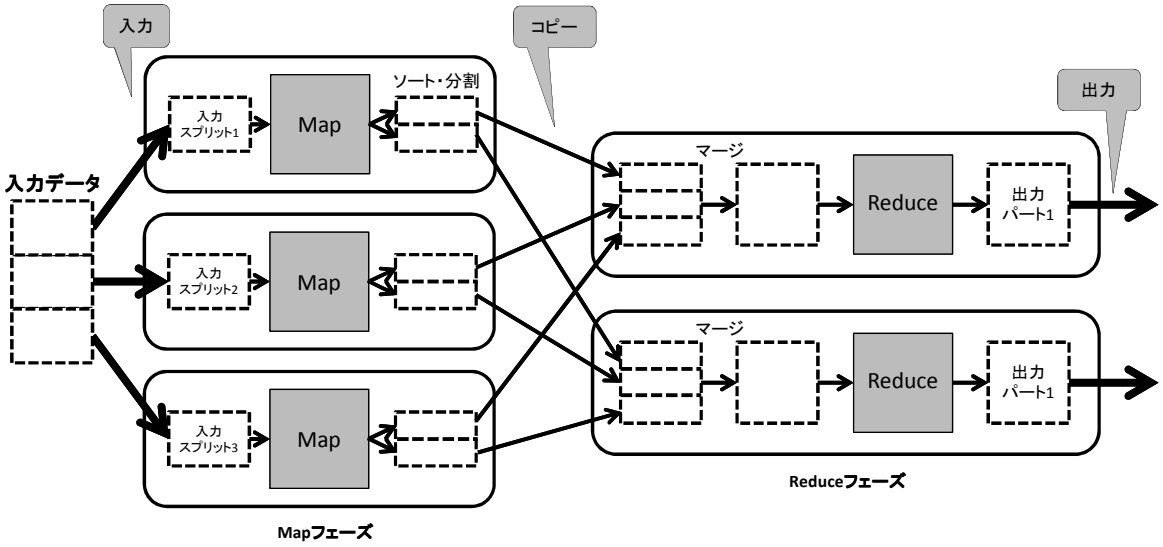


図 1: MapReduce データフロー

お，この操作をパーティション化と呼ぶ．Reduce フェーズにおいて Map タスクの中間出力をパーティションごとに集約し，Reduce タスクの配置されたノード上でマージを行う．マージされたデータはユーザが定義した Reduce 関数に渡され，Reduce 処理が行われる．Reduce 処理が行われたデータは，通常 HDFS に出力される．

2.3.3 Map フェーズ

Map タスクは入力スプリットから生成される．まず，クライアントによってジョブの分割が要求されると，InputFormat を実装するクラスの getSplits メソッドが呼ばれ，ストレージ上の全ての入力データから InputSplit という入力スプリットが生成される．InputSplit は，正確には入力データを分割した実体そのものではなく，バイト単位のサイズ情報とストレージ上の位置情報を持つ．同時に，InputSplit からレコード単位で Key，Value を取得するための RecordReader も生成される．

クライアントは，この InputSplit を JobSplitWrite クラスの createSplitFiles メソッ

ドに渡し，クラス内外の多数のメソッドを用いて，InputSplit のファイルへの書き出しと，InputSplit から SplitMetaInfo の生成及びファイルへの書き出しを行う．

一方，JobInProgress クラスで initTasks メソッドにより Map タスクの初期化が行われる．TaskSplitMetaInfo クラスの createSplits メソッドで書き出した SplitMetaInfo をファイルから再び読みだして TaskSplitMetaInfo を生成し，MapTask という Map タスクの実体が生成される．このように入力データからタスクへの入力スプリットを割り当てている．

JobTracker と TaskTracker は定期的に HeartBeat 通信を行っている．TaskTracker は HeartBeat 通信を用いて自分が新しいタスクを実行する準備ができているかどうかを JobTracker に知らせる．もし準備ができていれば，JobTracker はタスクスケジューリングを行い，タスクの実行準備ができたことを通知した TaskTracker に新しいタスクを割り当てる．JobTracker は割り当てるタスクを HeartBeat を用いて通知する．

図 2 のように，ユーザの定義した Map 関数は MapRunner によって呼び出され実行される．MapRunner では RecordReader によって割り当てられた入力スプリットからレコードを 1 つ取得し，1 つの KeyValue ペアを Map 関数に渡し Map 処理を行う．Map 処理が行われた KeyValue ペアはバッファに出力される．

Map 関数の出力は，そのデータが最終的に渡される Reduce タスクに対応するパーティションに分割される．各パーティション内では Key によるソートが行われる．combiner 関数が与えられた場合はソートの出力に対してその関数が実行される．combiner 関数はメモリバッファからあふれた場合に Map タスクの出力をコンパクトにするために用いられる．そしてパーティションで区切られたデータはそれぞれ別の Reduce タスクの入力として与えられる．

2.3.4 Reduce フェーズ

Map から出力された中間データは，その Map タスクを実行した TaskTracker のローカルディスク上に書き出されている．図 3 のように，Map から出力された中間データの各パーティションを入力として，Reduce タスクを実行しようとしている TaskTracker は，パーティションをクラスタ上の複数の Map タスクから入手しなければならない．

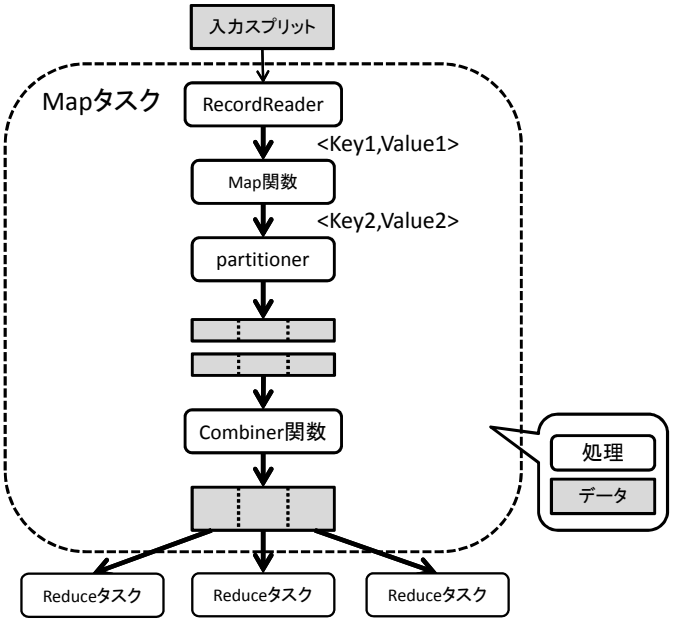


図 2: Map タスクの流れ

しかし，Reduce タスクの入力に必要なパーティションを出力する各 Map タスクの終了時刻が異なっている可能性があるため，Reduce タスクはそれぞれの Map タスクが終了したら，すぐに Map タスクの出力のコピーを開始する．Reduce タスクは任意の数のコピースレッドを持ち，Map の出力を並列に取得することができる．パーティションは，十分に小さい場合は Reduce を行う TaskTracker ノードの主記憶（メモリ）にコピーされ，そうでない場合は補助記憶（ディスク）にコピーされる．メモリ内のバッファが閾値のサイズまで使われるか，パーティションの数が閾値に達すると，その内容はマージされ，ディスクに書き込まれる．全てのパーティションをコピーし終えると，Reduce タスクは各パーティションのソート順序を保証しながらマージを行う．マージされたデータは各 Key ごとに Reduce 関数に渡される．Reduce 関数の出力は出力ファイルシステムに直接書き出される．このファイルシステムには通常 HDFS が使われる．

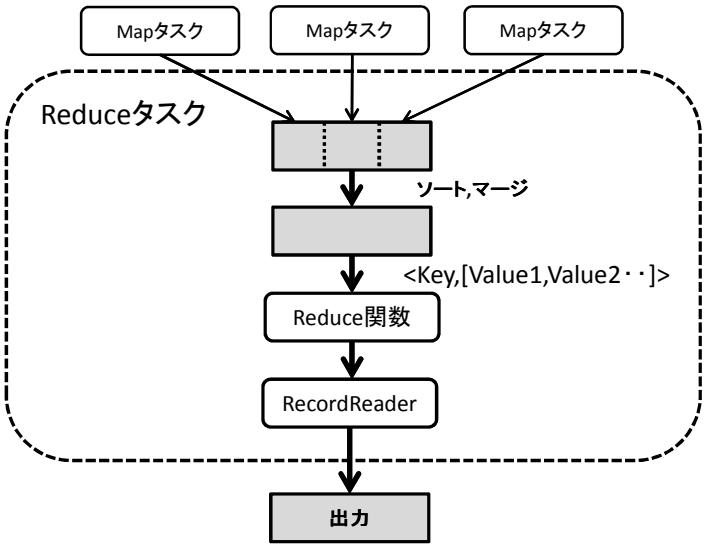


図 3: Reduce タスクの流れ

2.4 Hadoop のタスクスケジューリング

2.4.1 Hadoop のジョブスケジューラ

Hadoop には現在 3 つのジョブスケジューラが実装されていて，ユーザはどのスケジューラを使用するか設定することが可能である．

- ジョブキュータスクスケジューラ
  First In First Out(FIFO) キューに基づくスケジューラで，デフォルトで設定されているスケジューラである．このスケジューラでは JobTracker がワークキューからジョブを取得し，それらのジョブの中で最も古いジョブから実行する．このスケジューラにはジョブの優先度などの概念はないが，実装が容易で効率的なスケジューラである．

- キャパシティスケジューラ
  複数ユーザに対応したスケジューラである．このスケジューラではいくつかのキューが作成される．各キューには，保証されたキャパシティが割り当てられる．キャパシティとは，各キューが占有できるクラスタ内の Map 処理スロットおよ

び Reduce 処理スロット数の割合である．つまり，全てのキューのキャパシティをスロット数に変換すると，その合計はクラスタ全体の処理スロット数となる．各キューを利用することのできる人や組織を設定ファイルで定義することができるため，キューに対するアクセス制御は厳格である．このアクセス制御により，キューへのジョブ送信機能やキュー内のジョブの表示機能，変更機能を制限することが可能である．また，このスケジューラではジョブに対して優先度をつけることはできるが，実行中のタスクの一時中断はサポートしていない．

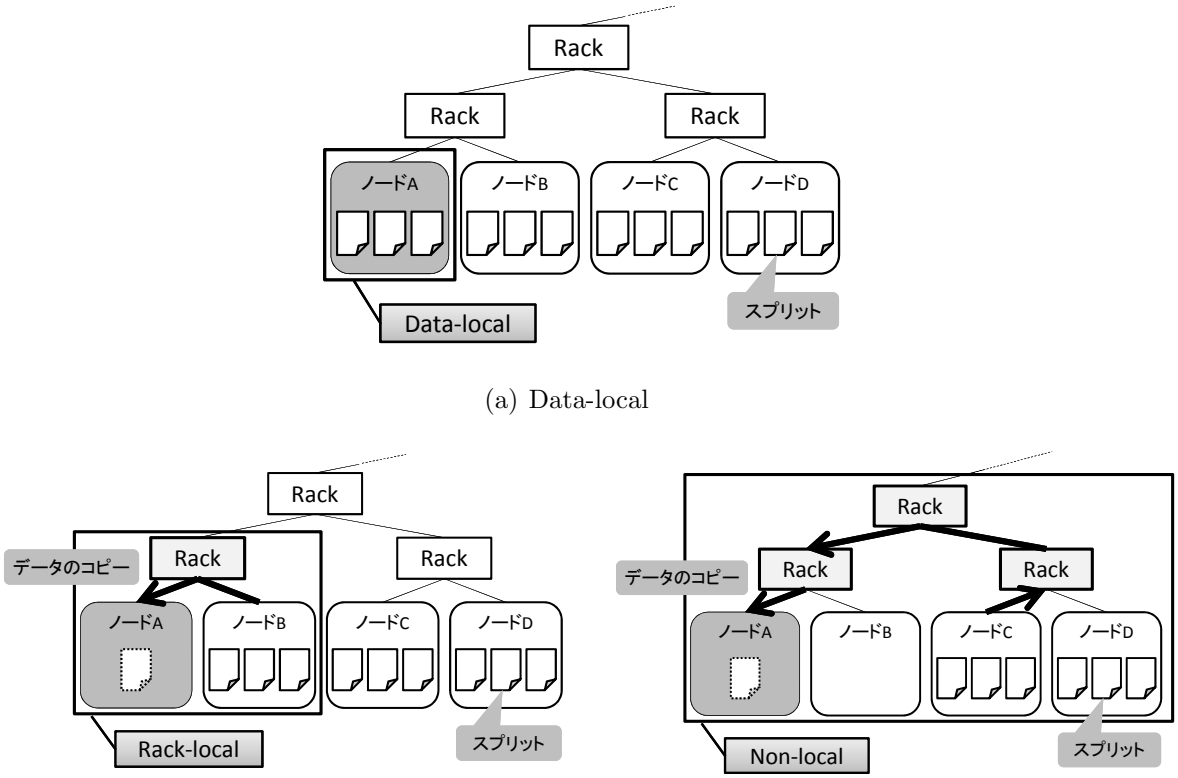
- フェアスケジューラ
  複数ユーザに対応したスケジューラである．長期的に見て各ユーザに公平にクラスタの処理能力を配分しようとする．Hadoop の実装では，ジョブを配置するプール群を作成し，スケジューラがジョブを配置するプールを選択する．公平性を確保するために各ユーザには 1 つずつプールが割り当てれられる．ジョブが 1 つしか投入されていない場合は，そのジョブはクラスタの全能力を占有するが，複数のジョブが投入されている場合は，各ユーザに公平にクラスタの能力が配分されるように空いているタスクスロットがジョブに割り当てられていく．こうすることであるユーザが多くのジョブを送信した場合でも，そのユーザは他のユーザと同じ分だけクラスタの能力が割り当てられる．つまり送信したジョブの量と無関係となる．

これらのスケジューラでは，TaskTracker がタスクの割り当てを要求した際のタスクの選択方法は共通している．はじめに図 4(a) のように対象ノード (仮にノード A とする) に配置されているスプリットに対応するタスク (このタスクを Data-local タスクと呼ぶことにする) を探す．Data-local タスクが存在する場合，スケジューラは対象の TaskTracker に Data-local タスクを割り当て，これによりローカルの処理が可能である．しかし，Data-local タスクを割り当てられない場合は，図 4(b) のように対象ノードが属する Rack 内の他のノード (仮にノード B とする) に配置されているスプリットに対応するタスク (これを Rack-local タスクと呼ぶことにする) を割り当てる．Rack-local タスクを処理する場合にはノード B からノード A へスプリットのコピーが必要となる．

さらに Rack-local タスクも割り当てられない場合は図 4(c) のようにさらに上の階層の Rack，またはノード A が属するクラスタ全体の他のノード (仮にノード C とする) に配置されているスプリットに対応するタスク (これを Non-local タスクと呼ぶことにする) を割り当てる．Non-local タスクを処理する場合も，Rack-local タスクを処理する場合と同様にノード C からノード A へスプリットのデータのコピーが必要となる．

Rack-local や Non-local で処理するタスクが増加すると，他のノードからデータをコピーする量が増加するので，実行時間の増加やネットワークトラフィックの増加を引き起こし，性能低下へとつながる．

よってタスクを割り当てる際は，タスクを要求したノードに配置されているスプリットに対する Data-local タスクを割り当てることでローカルに処理を行うことが理想的である．



(a) Data-local

(b) Rack-local

(c) Non-local

図 4: タスクの選択方法

2.4.2 複数ユーザで Hadoop を利用する際の問題点

複数ユーザで Hadoop を利用する際，各ユーザは他のユーザの利用状況や，Hadoop の動作状況は考えていない．そのため各ユーザがユーザ同士の利用状況や Hadoop の動作状況を把握していなくても，Hadoopが効率的に仕事を行うことができるようにスケジューリングを行う必要がある．また，複数ユーザ対応のスケジューラであるキャパシティスケジューラまたはフェアスケジューラを用いる必要がある．

キャパシティスケジューラでは TaskTracker からタスク割り当ての要求が来ると，図 5のように，はじめにどのキューからジョブを選択するかを決定するために，毎回キューのソートを行う．なお，キャパシティスケジューラでは各キューにキャパシティが設定されており (キャパシティは設定ファイルで変更が可能)，キャパシティの合計が 100 となるように設定されている．使用しているキューは queuesforAssigningTasks というリストの中に格納されており，このリストの中に入っているキューはキュー内のジョブの実行中タスク数を各キューのキャパシティで割った割合で昇順にソートされ，スケジューラはソートされたリストの先頭にあるキューを取り出す． 例えば，各キューが全て同じキャパシティを持っている場合には，キュー内のジョブの実行中タスクが少ない順にソートされている．また，キュー内のジョブの実行中タスクが等しい場合にはキャパシティが大きいキュー内のジョブから取り出されることになる．このようにキューが取り出され，次にキュー内にあるジョブを先頭から見ていき，未処理のタスクがあればそのジョブのタスクを選択し，対象の TaskTracker に割り当てる．

キャパシティスケジューラで各キューに設定されているキャパシティの割合だけリソースを使うことができるため，キャパシティに応じて公平性を保つことはできるが，各キューに入っているジョブの実行に必要なデータの位置情報は考慮していない．仮に，ある TaskTrackerがタスク割り当ての要求を行った際に，あるキューの進捗が遅れているために，そのキューが持つジョブのタスクを割り当てようとしたとする．このとき，その TaskTracker の動作するノードにはジョブのタスクに対応するスプリットがないと，TaskTracker は Rack-local タスクまたは Non-local タスクを処理せざるを得ない状況となってしまう．このように各キューに入っているジョブの実行に必要な入力データの位置情報を用いてスケジューリングを行わないと，Rack-local 及び Non-local

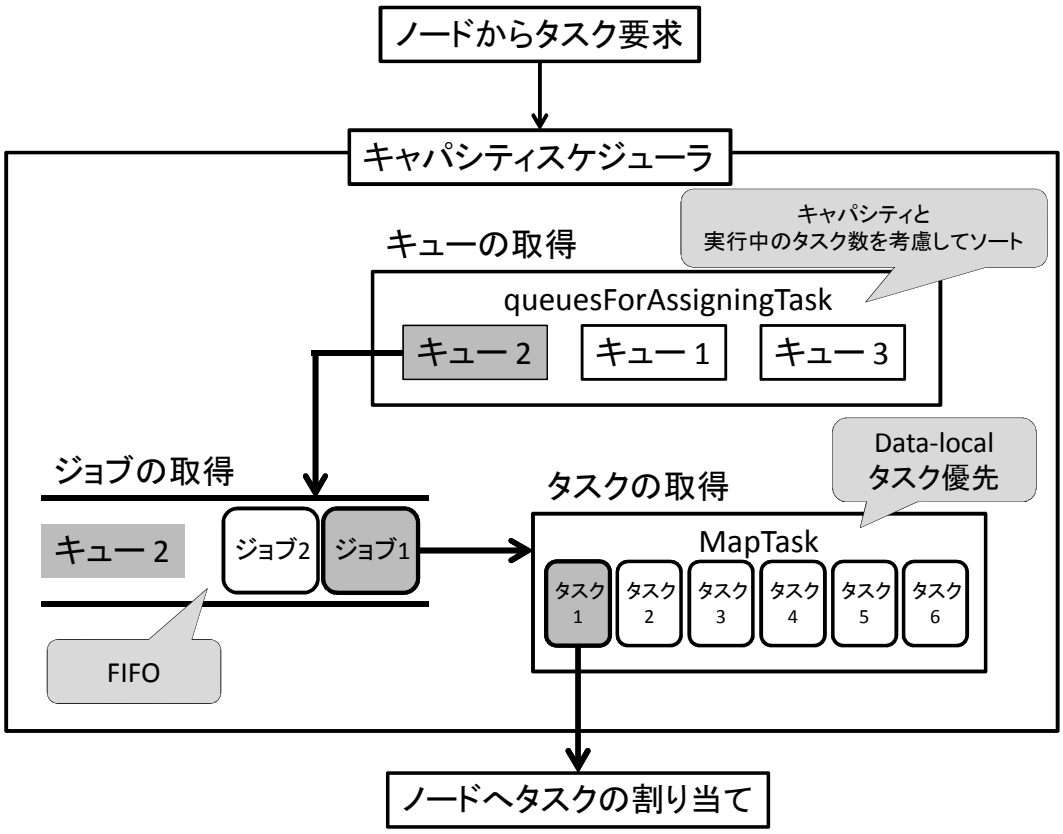


図 5: キャパシティスケジューラのタスク割り当て

の処理を増やしてしまう可能性がある．Data-local タスクを増やし，ジョブの実行時間を削減するためには，ジョブのスプリットの位置情報を用いてスケジューリングを行う必要があると考える．

3 提案と実装

3.1 提案手法

既存の Hadoop のキャパシティスケジューラでは，各キューのキャパシティと実行中のタスク数によって取り出すキューを選択している．本研究では 2.4.2 節で述べたスケジューリングに関する問題の解決に向け，各ジョブのスプリットの配置情報を用い

るスケジューリング手法を提案する．提案する Hadoop のキャパシティスケジューラに組み込む追加機能とその処理の流れを図 6 に示す.

キャパシティスケジューラでは，どのキューにジョブを投入するかを指定することが可能である．本研究ではキューを 3 つ用意し，各キューにそれぞれ別のユーザがジョブを 1 つずつ投入した場合を想定し，評価を行っているので，以下ではジョブ 1/2/3 の 3 つのジョブが各キューに投入されたとして説明を行う.

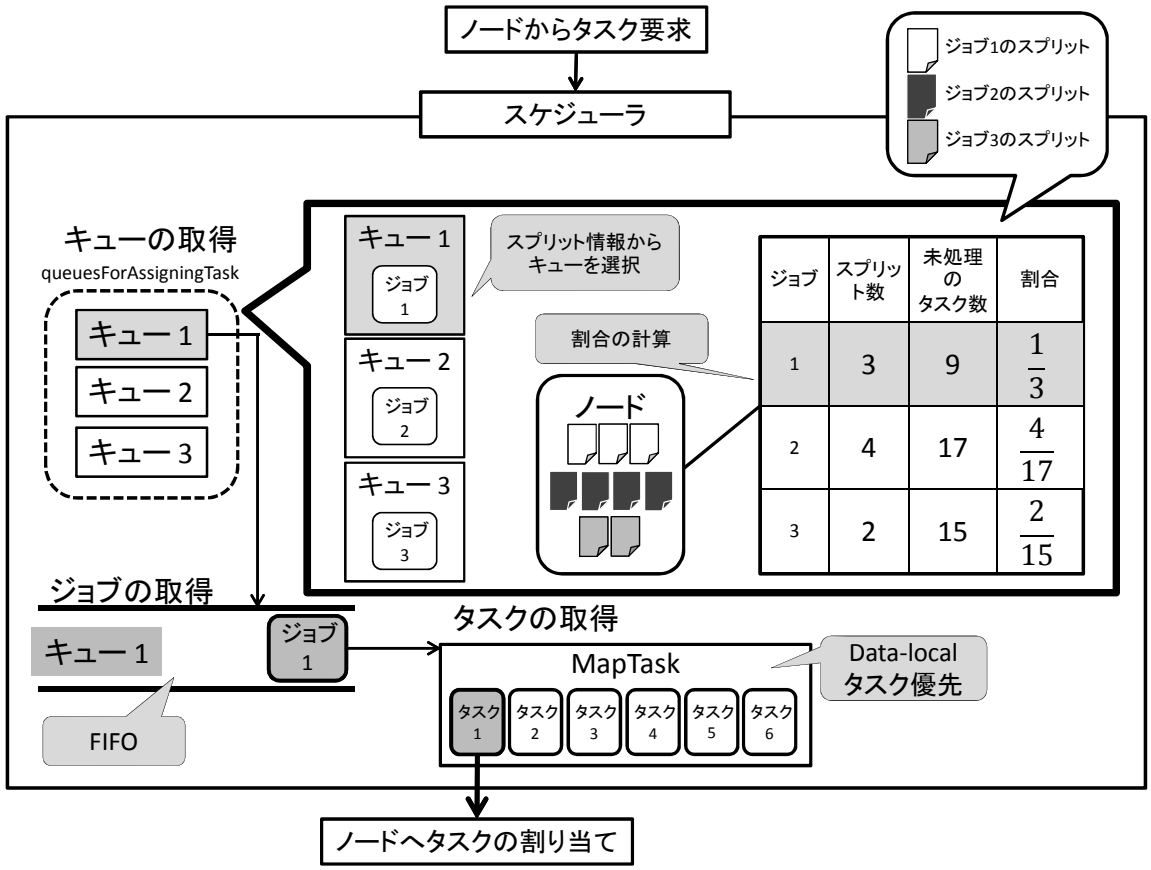


図 6: 提案するスケジューラの概要

あるノードの TaskTracker がタスク割り当ての要求を行ったとき，スケジューラはそのノードに置かれているスプリットに注目する．ジョブが複数あった場合に各ジョブで，それぞれの持つ未処理のタスクのうちいくつのタスクを処理することができるかという割合を計算し，その割合が最も大きいジョブのタスクを割り当てる．Map タ

スクとスプリットは一対一に対応しているので，各ノードに配置されている処理が行われていないスプリットの総数がそのノードでローカルに処理することのできる未処理のタスク数となる．

図 6 を用いて説明する．キュー 1,2,3 にそれぞれジョブ 1,2,3 が投入されたとする．タスクを要求したノードに配置されている未処理のスプリット数がジョブ 1 は 3，ジョブ 2 は 4，ジョブ 3 は 2 で，各ジョブの未処理のタスク数がジョブ 1 は 9，ジョブ 2 は 17，ジョブ 3 は 15 だとすると，(ノードの未処理スプリット数)/(ジョブの未処理タスク数) の割合はジョブ 1 が最も大きい．なお，本論文ではこの割合を SplitRate と呼ぶ．そのためスケジューラは SplitRate が最も大きいジョブ 1 が入っているキュー 1 を選択し，ジョブ 1 のタスクを対象のノードに割り当てる．

単純に対象のノードに配置されている各ジョブの未処理のスプリット数で比較してしまうとスプリット数が多い，すなわち Map タスク数が多いものほど各ノードに配置されるスプリットの数は多くなり，優先して処理が行われてしまうことになる．また，進捗率の高いジョブも未処理スプリット数は少なくなるため，進捗率の高いものほど優先度は低くなってしまうことになる．よって Map タスクの数によって自動的に優先順位が定まってしまうことがないように，SplitRate を計算することで比較を行っている．このように SplitRate を計算し，最も割合の大きいジョブのタスクを割り当てるようにスケジューリングを行い，各ノードのスプリット総数のバランスをとることで，Data-local タスクの増加を図る．

3.2 実装

既存仕様の Hadoop のキャパシティスケジューラを用いて，実際に Map タスクを割り当てるスケジューラの部分を変更する．スプリットの配置を考慮したスケジューラを実装するにあたり，JobInProgress で，createCache メソッドを用いてジョブ毎に SplitTable という連想配列を準備する．Key を各ノードのホスト名，Value を各ノードが持つスプリット情報である SplitList というリストで管理する．これによりどのノードがどのスプリットを持つのか，また，そのノードが持つスプリットの総数を把握することができる． また，スプリットと Map タスクを関連付ける連想配列も準備する．

JobTracker と TaskTracker は定期的に HeartBeat 通信を行っており，TaskTracker は JobTracker に TaskTracker の状態を示す TaskTrackerStatus を渡している．TaskTrackerStatus は TaskTracker のホスト名やメモリ容量，最大 Map スロット数，最大 Reduce スロット数などの情報を保持しているオブジェクトである．JobTracker は TaskTrackerStatus からその TaskTracker で処理することのできる最大 Map スロット数と，現在 Map 処理に使われているスロット数の情報を取得する．最大 Map スロット数と使用中スロット数の差がタスク割り当て可能なスロット数 (availableSlots) となり，availableSlots が 0 よりも大きい場合にタスク割り当ての処理が開始される．はじめにデフォルトのキャパシティスケジューラのスケジューリング方法に従って queueForAssigingTasks というリストに入っているキューがソートされる．次に assignTasks メソッドが呼び出される．実際にキュー，ジョブ，タスクが選択されるのは assignTasks メソッド内なので assignTasks メソッド内に提案手法の実装を行う．実装の手順は以下の通りである．

Hadoop でタスクスケジューリングを行う際に，タスクのタイプは MAP，REDUCE，SETUP，CLEANUP の 4 種類に区別される．図 7 のように，assignTasks メソッドにおいて，タスクのタイプが MAP である場合，どのキューのジョブを取り出すかを選択するために getMaxQueue メソッドを呼び出す．getMaxQueue メソッド内では各キューの SplitRate を計算する getMaxJobSplitRate メソッドを呼び出す．getMaxJobSplitRate メソッド内では未実行のタスク数を取得する getUnlaunchedTask メソッドと，未処理のスプリット数を取得する getRawSplit メソッドを呼び出し，それぞれの値を取得した後，実際に計算を行う．このようにしてジョブを取り出すキュー (maxQueue) を決定すると maxQueue の中からジョブを取り出し，そのジョブのタスクから Data-local タスクを優先的に取得し，割り当てる．このような処理が availableSlots が 0 より大きい限り繰り返される．

以下で各メソッドの詳しい処理を説明する．

- getMaxQueue

  CapacitySchedulerQueue クラスの queuesForAssigingTasks というキューの入ったリストの中からキューを 1 つずつ取り出し，各キューの SplitRate を比較し，最も値が大きいものを選出する．SplitRate は getMaxJobSplitRate メソッドを呼

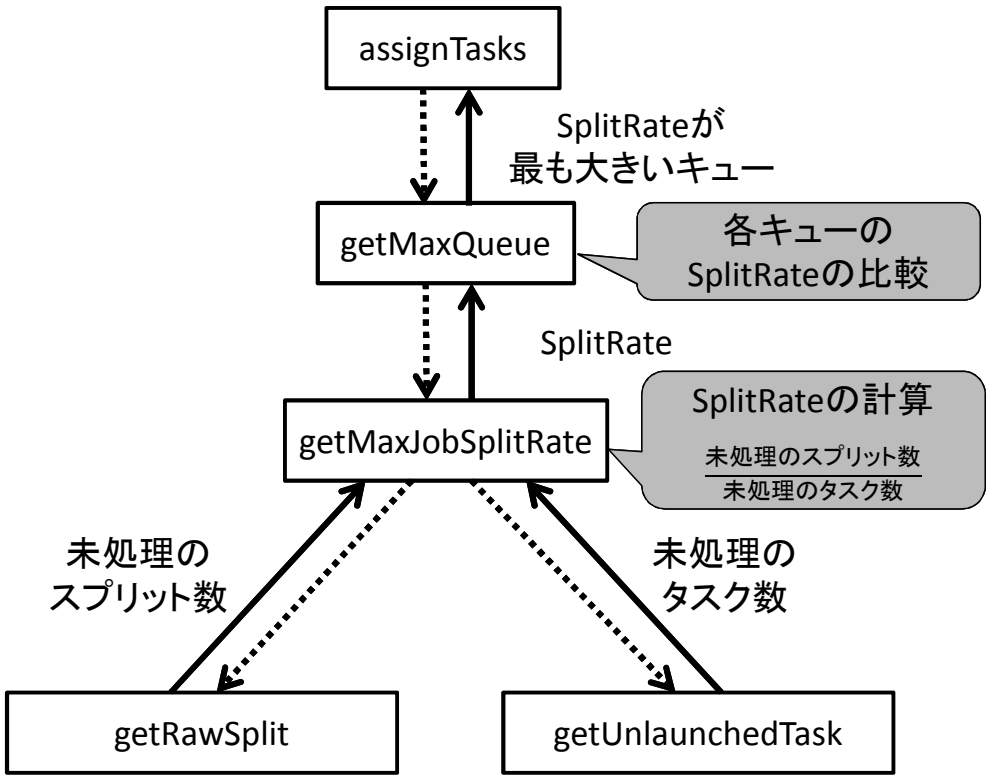


図 7: 提案するスケジューラの実装

び出し取得する. 戻り値は未処理タスクの中のスプリットの割合が最も多いジョブの入ったキューである.

- getMaxJobSplitRate
  キューに入っているジョブの中で実行中のものを取り出し，getUnlaunchedTask メソッドと，getRawSplit メソッドを呼び出し，SplitRate を計算する．キューに入っているジョブを比較し，最も大きい値を選出し戻り値として返す．キャパシティスケジューラでは各キューの中でのジョブの実行方法は FIFO となっているので，基本的には先頭のジョブが実行中として取り出されてそのジョブの SplitRate が計算され戻り値として返される.

- getUnlaunchedTask
  対象のジョブの未処理のタスクの総数を返す．各ジョブのタスクの総数は numMapTasks という変数で管理されている．未実行のタスク数を表す変数は用意されていないので，新たに mapcount という TaskTracker の処理スロットに配置が完了した Map タスクの数を管理する変数を用意した．ジョブが投入された時点では 0 に初期化しており，新たに Map タスクをスケジューリングし，タスクを runningCache という実行中のタスクを管理するマップに追加した際に mapcount をインクリメントする．よって numMapTasks と mapcount の差を未実行のタスク数として管理を行う．

- getRawSplit
  対象のノードに配置されている対象のジョブの未処理のスプリット総数を返す．ここでは，はじめに作成しておいた連想配列 SplitTable を用いる．対象の TaskTracker のホスト名を取得し，SplitTable のキーにホスト名を入れバリューを splitList というリストに入れる．splitList が null でない場合，すなわち未処理のスプリットが 0 でない場合に，splitList のサイズを返す．これにより未処理のスプリットの総数を返すことができる．

getMaxQueue メソッドは CapactityTaskScheluler クラスの assignTasks メソッド内で呼び出されている．assignTasks メソッドは CapacityTaskScheduler クラスの addMapTasks メソッドで呼び出されている．addMapTasks メソッド内では assignTasks メソッドを呼び出す前にキューの入ったリスト (queuesForAssigningTasks) 内の全てのキューを実行中のタスク数とキャパシティを考慮してソートを行っている．そのため，提案したスケジューラにおいて全てのジョブの SplitRate が等しいときには，デフォルトのキャパシティスケジューラのキューの選択方法に従ってキューが選択されるようになっている．

4 評価

本提案を実現するために既存仕様の Hadoop に対して提案手法の実装を行い，ベンチマークプログラムを用いて性能評価を行った.

4.1 評価環境

実行環境を表 1 に示す. マスターノード 1 台とスレーブノード 14 台の計 15 台でクラスタを構成しており，2 つのラック構成に設定した．データのレプリカ数は設定ファイルで設定可能であるが，本評価ではレプリカ数は 2 と設定した.

表 1: 実行環境

	マスターノード	スレーブノード
Hadoop のバーション	hadoop-1.0.3	hadoop-1.0.3
ファイルシステム	HDFS	HDFS
CPU	(AMD Opteron-12-core 6168/1.9GHz)x2	Core i5 750/2.66GHz
メモリサイズ	32GB	8GB
OS	CentOS 5.7	Ubuntu
ノード数	1	14

評価対象のベンチマークプログラムとして Hadoop に標準で付属している Sort プログラムと Intel の提供する HiBench の中の PageRank プログラム ([9]) を採用した．各プログラムを 5 回ずつ実行し，その平均を計算した．各プログラムの入力データのサイズは全てのユーザで統一し，Map タスク数のみ変更した．各プログラムの Map タスク数を表 2 に示す.

Sort プログラムの入力データサイズを 5GB に設定した．PageRank プログラムは Web ページのランク付けをするプログラムであり，Web ページのリンク関係と現在のランクを整理する stage1 と，ページごとに被リンク先のランクと被リンク数からランクを計算する stage2 の 2 つの段階があり，各段階で MapReduce が行われる．stage1 の Reduce の出力が stage2 の Map の入力となっている [10]．入力データサイズは 5000000 ページのデータとした.

評価内容は Map タスクの内訳と，各ジョブの実行時間である．Map タスクの内訳は Data-local タスクがどれだけ増加したかを示し，それによりジョブの実行時間がどれだけ削減できたかを各ジョブの実行時間で示した．

また，スケジューラはキャパシティスケジューラに提案手法の実装を行ったもので，キューの設定を行うことができる．本研究では各ユーザに 1 つずつキューを用意し，各キューのキャパシティは全て等しくなっている．

表 2: ベンチマークの Map タスク数

	Map タスク数		
ユーザ	Sort	PageRank(stage1)	PageRank(stage2)
1	80	120	80
2	100	140	100
3	120	160	120

4.2 評価結果

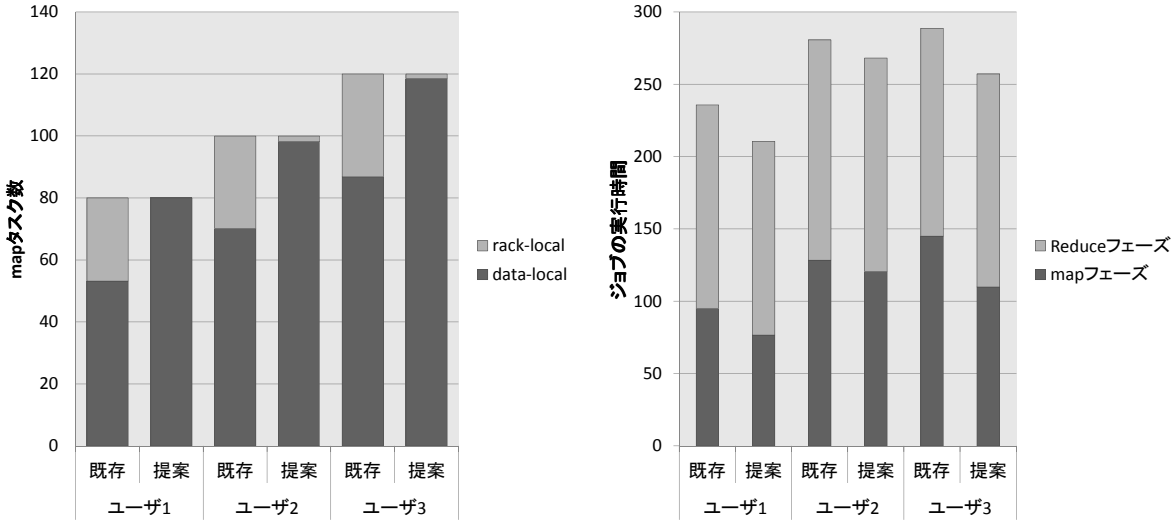
評価結果を図 8 から図 10 に示す．左のグラフ (a) は全てタスクの内訳を示し，右のグラフ (b) はジョブの実行時間の内訳を示している．各グラフの中ではユーザ 1，ユーザ 2，ユーザ 3 を既存の Hadoop の評価結果と提案手法を追加実装した Hadoop の評価結果を比較している．タスクの内訳を示すグラフにおいては濃い色の部分が Data-local タスクの割合，薄い色の部分が Rack-local タスクの割合を示している．ジョブの実行時間の内訳を示すグラフにおいては，濃い色の部分が Map フェーズにおいてかかった時間を示し，薄い色の部分 Reduce フェーズにかかった時間を示しており，濃い色と薄い色の部分の合計がジョブ全体の実行時間を示している．

まず Sort プログラムにおいては全ユーザで平均の Rack-local タスクの割合が大幅に減少し，Data-local タスクの割合は大幅に増加した．3 つのユーザで各 5 回ずつジョブを実行したが，その中でほぼ全てのジョブの Map タスクは Data-local で処理が行われていた（図 8(a)）．ジョブの実行時間については全てのユーザにおいて削減されている．Map フェーズの実行時間に注目しても削減されていることがわかる (図 8(b))．ユーザ

3 に最も効果が現れ，ジョブの実行時間が 11%，Map フェーズの実行時間が 24%削減された．

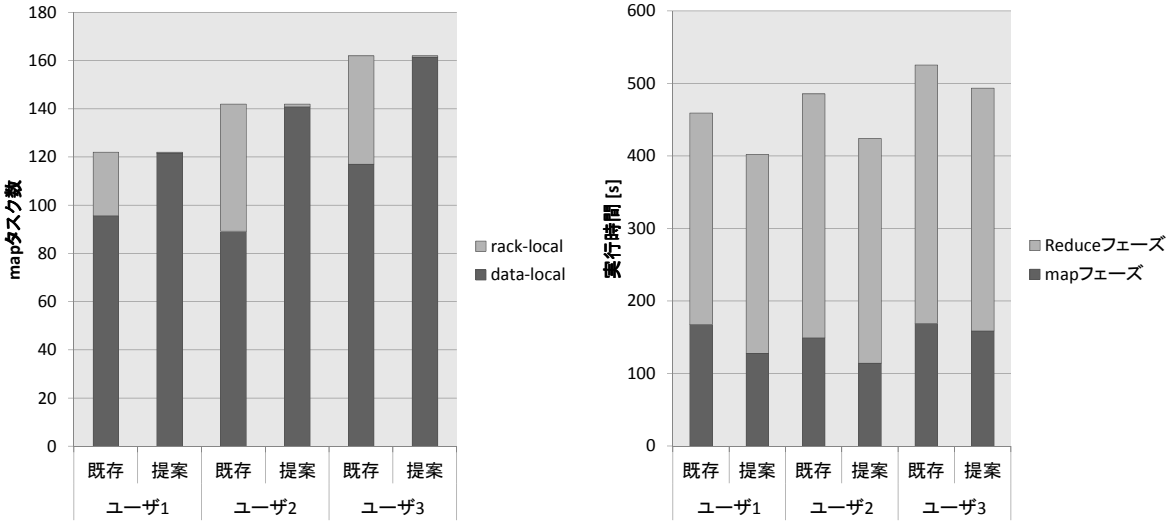
PageRank プログラムの stage1 においては図 9(a) に示すように，Sort プログラムと同様に全てのユーザにおいて Map タスクに占める Data-local タスクの割合は大幅に増加し，ジョブ全体の実行時間と Map フェーズの実行時間についても共に削減された (図 9(b))．ユーザ 2 に最も効果が現れ，ジョブの実行時間が 13%，Map フェーズの実行時間が 23%削減された．

PageRank プログラムの stage2 においては全てのユーザにおいて Map タスクに占める Data-local タスクの割合は大幅に増加しているが (図 10(a))，実行時間に注目するとユーザ 2 とユーザ 3 については共に実行時間は削減されているが，ユーザ 1 のジョブ全体の実行時間がわずかに増加している (図 10(b))．しかし，Map フェーズの実行時間はわずかに削減されている．ユーザ 2 で最も効果が現れ，ジョブの実行時間が 6%，Map フェーズの実行時間が 17%削減された．



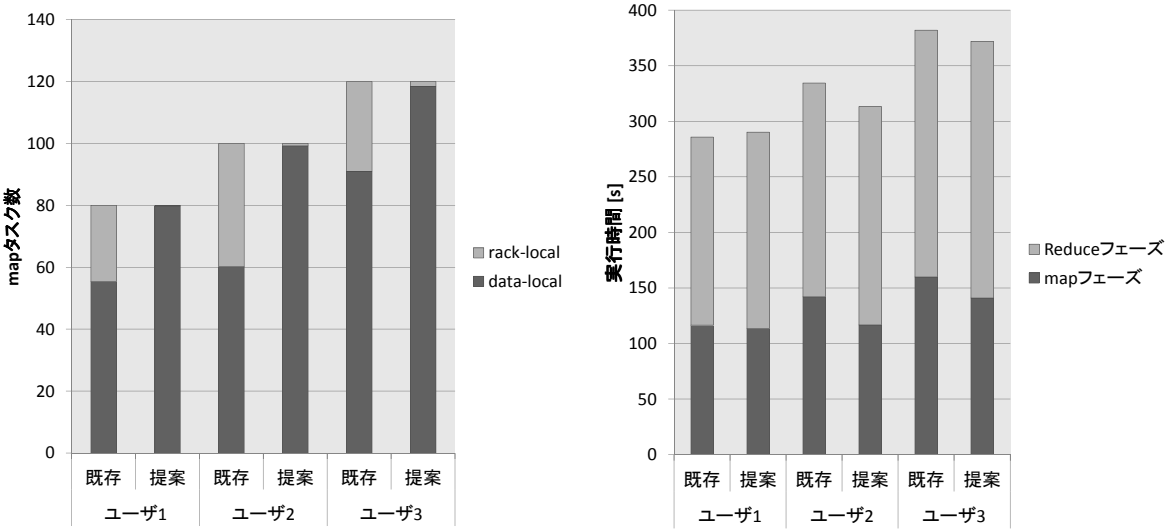
(a) タスクの内訳　　(b) 実行時間の内訳

図 8: Sort



(a) タスクの内訳 (b) 実行時間の内訳

図 9: PageRank stage1



(a) タスクの内訳 (b) 実行時間の内訳

図 10: PageRank stage2

5 考察

キャパシティスケジューラを用いたHadoopと比較して提案手法を実装したスケジューラを用いたHadoopではほぼ全てのジョブについて実行時間を削減することができた．Mapフェーズの実行時間が削減されたことがジョブ全体の実行時間の削減へとつながっていると考えられる．これはMapタスクの大部分がRack-localタスクではなくData-localタスクで処理されたため，Rack-localタスクで必要な他ノードからのデータのコピー時間が削減されたからであると考えられる．ほとんどのジョブは実行時間は削減することができたが，ユーザ1が実行したPageRankプログラムのstage2のジョブのみ全体の実行時間はわずかに増加している．このジョブにおいてもMapフェーズはわずかであるが削減されているので，Reduceフェーズにおいての実行時間が増加しているからであると考えられる．他のジョブにおいてもReduceフェーズの実行時間が増加しているジョブもあれば，逆に削減されているジョブもある．これはMapタスクのスケジューリング手法を変更したことでMapフェーズの出力する中間データの配置が大きく変わり，Reduceフェーズに影響が出たためだと考えられる．

図9(a)において既存のHadoopで実行したユーザ2のジョブのData-localタスクの割合は他のユーザのジョブよりも少ない．既存のHadoopでは各ユーザで実行ごとにData-localタスクの割合のばらつきが大きかった．これは，既存のHadoopでのスケジューリングの方法がスプリットの配置を考慮していないため，ネットワークの遅延などで各タスクの進捗具合が変化し，その結果ジョブごとに毎回大きく異なる割合となったことが理由であると考えられる．そのため今回のPageRankプログラムのstage1の評価においては，ユーザ2は他のユーザに比べてData-localタスクが割り当てられることが多かったが，再評価を行ったり実行数を増やしたりすると，ユーザ1やユーザ3のData-localタスクの割合が減少することも十分に考えられる．

また，ほぼ全てのジョブにおいてデータは同じサイズであるにも関わらずMapタスクが少ないジョブの実行時間が短いのは，ジョブのMapタスクが多いほどタスク割り当ての回数が多くなり，タスクを割り当てる度に必要なタスクのセットアップや，JobTrackerからTaskTrackerへのHeartBeat通信などのオーバーヘッドが大きくなるためであると考えられる．しかし，図8(b)の提案手法を実装したHadoopでは，Map

タスク数が 120 であるユーザ 3 のジョブの実行時間の方が Map タスク数が 100 であるユーザ 2 のジョブの実行時間よりも短いことがわかる．既存のキャパシティスケジューラを用いた Hadoop ではユーザ 3 の方がユーザ 2 よりもジョブの実行時間は長くなっている．これは各キューのキャパシティよりも各ノードに配置されているスプリット数を考慮してタスクを割り当てているために，各ノードがタスクの要求を行った際にユーザ 3 の投入したジョブ 3 がユーザ 2 の投入したジョブ 2 の SplitRate よりも大きく，ジョブ 3 のタスクがジョブ 2 のタスクよりも早く割り当てられてしまったからであると考えられる．本研究の提案手法では，単純に各ノードに配置されている Split 数だけでなく SplitRate を用いているため，Map タスク数が多いジョブのタスクが優先されることはないが，各ノードの Split の配置によっては各ジョブの進捗に影響が出てしまう可能性があると考えられる．

6 今後の課題

本研究では Data-local タスクを割り当てるようなスケジューリングを行うことで実行時間を削減することを目的としたスケジューラを提案したため，各キューのキャパシティは全て等しく設定した．しかし，キャパシティスケジューラでは各キューのキャパシティを設定することで，ユーザが各ジョブの優先度を自由に変更できるものである．具体的には，早く実行したいジョブはキャパシティが多く割り当てられているキューにジョブを投入し，早く実行させる必要のないジョブはキャパシティの割り当てが少ないキューに投入する．今回提案したスケジューラでは本来ユーザが自由に静的に設定できるキャパシティを考慮してスケジューリングを行っていないため，各キューに設定されたキャパシティが異なっていても，各ジョブの優先度をつけることができない．よってこのような場合に対処できるように，キャパシティの値などを計算式に含めることで，各ノードに配置されているスプリット数と各キューに割り当てられているキャパシティの両方を考慮してタスクの割り当てを行うスケジューリング手法を提案することが今後の課題である．

7 まとめ

本研究では，複数ユーザが Hadoop を用いて同時に複数ジョブが実行される場合，ノードからノードへのデータのコピーのオーバーヘッドを削減する手法として，各ジョブのスプリットの配置を考慮したスケジューリング手法を提案した．提案の有効性を確認するため，Hadoop に標準で付属している sort プログラムと Intel が提供している Hibench の中の PageRank プログラムを用いて評価を行った．その結果，全てのジョブにおいて Data-local タスクの割合を増加させることができ，ジョブの実行時間を最大 13%，Map フェーズの実行時間を最大 24%削減することができた.
今後の予定として，キャパシティの値を計算式に含めることで各キューのキャパシティも考慮して，Map タスクを割り当てることができる手法を提案することが考えられる.

謝辞

本研究を進めるにあたり，ご指導を頂いた卒業論文指導教員の松尾啓志教授, 津邑公暁准教授, 齋藤彰一准教授, 松井俊浩准教授, 梶岡慎輔助教に感謝致します．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, ならびに松井研究室の皆様，特に研究に関して貴重な意見を下さった藏澄汐里氏と水野航氏，山崎一樹氏に深く感謝致します.

参考文献

[1] Tom White. *Hadoop*. オーム社, 2011.

[2] 太田一樹/下垣徹/山下真一/猿田浩輔/藤井達朗. Hadoop 徹底入門　オープンソース分散処理環境の構築. 翔泳社, 2011.

[3] 田澤孝之/横井浩/松井一比良. はじめての Hadoop ～分散データ処理の基本から実践まで. 技術評論社, 2012.

[4] *Hadoop Wiki*. http://wiki.apache.org/hadoop/FrontPage Available online on 2013.2.11.

[5] *ApacheHadoopProject*. http://hadoop.apache.org/ Available online on 2013.2.11.

[6] *ApacheHadoopProject:HadoopDistributedFileSyetem*. http://hadoop.apache.org/hdfs/ Available online on 2013.2.11.

[7] 三木大知. パターンでわかる Hadoop MapReduce -ビッグデータのデータ処理入門. 翔泳社, 2012.

[8] *ApacheHadoopProject:HadoopMapReduce*. http://hadoop.apache.org/mapreduce/ Available online on 2013.2.11.

[9] *gitHub HiBench*. https://github.com/hibench/ Available online on 2013.2.11.

[10] 西田圭介. Google を支える技術. 技術評論社, 2008.