平成23年度　　卒業研究論文

Hadoopのメニーコア環境に適したスロット数の動的調整手法

指導教官
松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成20年度入学 20115053番

藏澄 汐里

平成24年2月8日

目 次

1 はじめに

インターネット上における多種のサービスの展開と，インターネットを利用する人口の爆発的な拡大により，世界各地でデータ量は膨大に増え続け，それに伴い処理も大規模化している．膨大なデータを処理できるようなパフォーマンスの高いコンピュータは高価な上に，このようなコンピューター台を使用してサービス全体を管理する場合，障害発生時にサービス全体を停止しなければならないという欠点もある．そのため，安価なコンピュータを複数台連携させ，処理やデータを分散し，全体のパフォーマンスを向上させるという方法がとられてきた．

しかし，分散プログラミングは，ネットワーク通信のトラブル，ノードの故障などへの対処，及び，処理を複数台のコンピュータに分散させるための記述も必要であり，コーディングが繁雑である．ユーザがそういった分散のための処理を意識することなくシンプルにコーディングを行うために，本来のアルゴリズム以外の記述を大幅に省略可能なHadoopが用いられるようになった．Hadoopへの注目度は高く，企業・個人共に利用需要は急激に増えてきている．

また，従来コンピュータはCPUのクロック周波数を上げることで処理能力を高めてきたが，このクロック周波数の上昇率の伸び悩み，上昇に伴う消費電力の増加，熱の発生などが問題となり，マルチプロセッサ・マルチコアプロセッサのように，コアを複数にすることで処理能力を高める方向に方針が変更された．そして，並列コンピューティングという強い基盤の存在から，本格的なメニーコアの時代に入ろうとしている．今後は数十コアがワンチップに搭載されることが期待される．

メニーコア環境を有効利用するためには並列プログラミングすることが必要であるが，並列処理特有のデータの同期及びデータの分割などの問題もあり，分散プログラミング同様容易ではない．コーディングをシンプルに行うために分散処理と同様Hadoopを並列処理にも利用することはできるが，ネットワーク上の多数のノードを用いて通信し処理を分散させることを前提として開発されているため，一つのノード内で並列に処理を行う場合には最適ではない部分もある．

そこで本研究では，まず，Hadoopをメニーコア環境でも有効活用できるように最適化することを目的とし，メニーコア環境に適したスケジューリング手法の基礎検討を

行った．Hadoopに標準で使用されているタスクスケジューラの拡張機能として，システム内の負荷状況の検知によるスロット数の動的調整機能を実装した．そして，Business IntelligenceのようなI/O処理よりもCPU内の演算処理が主なプログラムをシミュレートするためにHadoopに標準で付属しているSortプログラムにランダム時間の遅延を挿入し，このプログラムをHadoop上で実行し，24コアマシンにおいてどの程度外部プログラムの実行によるHadoopタスクの並列実行制御の乱れの影響を低減できるか評価を行った．

本論文では，第2章ではHadoopの詳細とメニーコア環境においてHadoop上で並列処理を実行する場合の問題点を挙げ，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全体をまとめる．

2 研究背景

2.1 Hadoopの概要

Hadoopとは大規模データの並列分散処理を支えるプラットフォームである．Apacheによるオープンソースプロジェクトの一つであり，世界規模の開発貢献者コミュニティによって開発され，現在も続けられている．

数千ノードの利用及びペタバイト級のデータを扱うことを可能にし，本来ならば何日もかかるような大規模な処理を現実的な時間で完了することを実現した．

Hadoopは大きく分けて，大量のデータを管理する分散ファイルシステム Hadoop Distributed File Systemと，並列分散処理フレームワーク Map Reduceから構成されている．下記でこれら二つについて述べる．

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．Hadoopを構成する要素の一つとして，組み込まれている．大容量，スケーラビリティ，高スループット，耐障害性などの特徴がある．

HDFSの利便性として二つの特徴が挙げられる．まず，透過性である．ユーザはHDFSを透過的に利用することができ，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なく，あたかも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．そして，もう一つの特徴は拡張性である．HDFSは拡張が非常に容易であり，サーバの台数を増やすことだけでI/O処理性能向上を図ることができる．ディスク容量の増加もサーバを追加することで可能となる．

さらに，信頼性の確保も行っている．HDFSはファイルを分割した一定サイズのブロックをレプリケーションし，それらを複数のサーバに分散して保管する．これにより，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．

2.3 Hadoop Map Reduce

2.3.1 Hadoop Map Reduceの概要

Hadoop Map Reduceとは，Googleが開発した並列分散処理フレームワークMap Reduceをオープンソースで実装したものである．HDFSと同様にHadoopの構成要素の一つとして，組み込まれている．

Map Reduceは，クライアントが実行要求する作業単位であるジョブを並列分散処理可能な独立したタスクの集合に分割しそのタスクを多数のノードに分散して実行する．分散させるノードの数・並列実行させるコア数が多い程タスクの分割数を増やせるため，大規模な並列分散処理を可能としている．一つのJobTracker（マスターサーバ）と多数のTaskTracker（スレーブサーバ）から構成されており，タスクの管理・スケジューリングをJobTrackerが司り，配置されたタスクの実際の実行をTaskTrackerが担当する．処理は大きく，MapとReduceの二つのフェーズに分かれていて，詳細は2.3.2節で述べる．入力にはキー・バリューのペアのデータ集合を扱う．

Map Reduceの一つの特徴として，非常にシンプルにコーディングできることができるという点が挙げられる．コーディングに最低限必要な記述は，Map処理のための関数，Reduce処理のための関数，入力先及び出力先のデータ・ディレクトリ及びフォーマットの指定，そしてジョブに関するパラメータだけである．分散を行うための処理，ノードの故障・データ転送のトラブルなどへの対処は，Map Reduceの枠組に含まれており，ユーザは本来のアルゴリズムのみの記述に専念することができるという利点がある．

2.3.2 Map Reduceのデータフロー

Map Reduceのデータフローを図1に表す．Map Reduceでは，実行するタスクごとにJava Virtual Machine（JVM）を起動し，JVM上でタスクの処理が行われる．Map・Reduceの各フェーズでは，入力・出力いずれもキー・バリューのシンプルなデータ構造をとる．

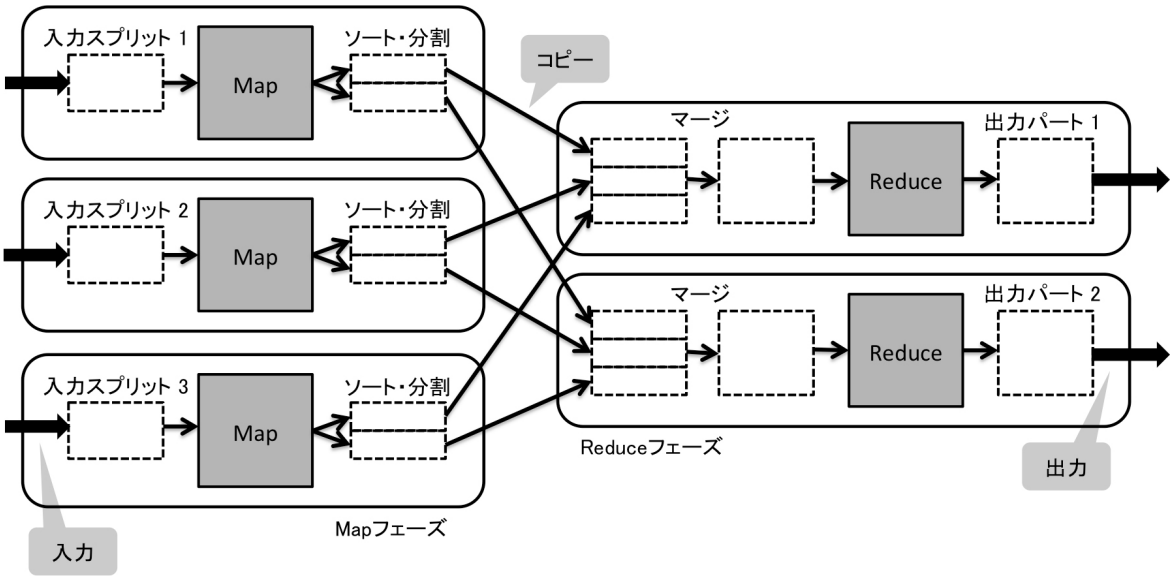

図 1: Map Reduce のデータフロー

Map フェーズでは，まず，ジョブへの入力データが設定した Map タスクの数分に分割される．これは入力スプリットと呼ばれる．Map Reduce は入力スプリット一つに対して一つの Map タスクを生成する．この入力スプリットが，JobTracker によって Map タスクが割り当てられた，TaskTracker の動作するノードで起動した JVM に渡され，JVM 上で Map 処理が行われる．処理が完了したタスクから，処理を行ったデータをソートし中間データとして出力する．

次に，Map タスクの出力を集約する．Reduce フェーズでは，完了した Map タスクによって出力された中間データから Reduce タスクが必要とする部分のデータを取得する．集められたデータはマージされ，これを入力として，Map 処理と同様 JVM 上で Reduce 処理が行われる．この結果が，最終出力として書き出されるか，もしくは再びマージされ次の Map フェーズの入力データとして渡される．

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（これは入力データのフォーマットに準ずる）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入力スプリット（InputSplit）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キー・バリューを取得するための RecordReader も生成される．

クライアントは，この InputSplit を JobSplitWriter 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 タスクの初期化作業が行われる．SplitMetaInfoReader クラスの readSplitMetaInfo メソッドにより，書き出した SplitMetaInfo をファイルから再び読み出して TaskSplitMetaInfo が生成される．この TaskSplitMetaInfo から TaskInProgress が Map タスクの数だけ生成され，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 HeartBeat によって常時通信を行っている．この HeartBeat 毎に JobTracker はタスクスケジューリングを行っていて，割り当てられた Map タスクは TaskTracker により実行に移される．

ユーザの定義した Map 関数は，直接的には 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一つ取得し，キー・バリューを Map 関数に渡し Map 処理を行う．つまり，レコードの数だけ Map 関数は呼び出される．Map 処理が行われたキー・バリューは単純にキー・バリューのペアを出力することを目的としている OutputCollector に渡され，出力される．

Map タスクによる出力を中間データと呼び，この中間データは Partitioner により複数のパーティションに分割される．

2.3.4 Reduceフェーズ

Reduceタスクは，JobInProgressクラスのinitTasksメソッドによりMapタスクの初期化作業が行われた際，同時に生成される．Mapタスクに相当するTaskInProgressが生成された後，実際に生成されたMapタスクの数などを引数として渡しReduceタスクに相当するTaskInProgressも生成され，ReduceTaskというタスクの実体が生成される．

タスクスケジューリングにより空きスロットにこれ以上Mapタスクを配置できなくなるか，Reduceタスクを配置する前に完了していなければならないMapタスクの数（completedMapsForReduceSlowstart）以上のMapタスクが完了していた場合，Reduceタスクが空きスロットに配置されるようになる．Mapタスクと同様，TaskTrackerにより実行に移されるが，Reduceタスクの入力とする中間データのパーティションがMapタスクから出力されるのを待たなければならない．このため，Reduceタスクは実行開始からJobTrackerに繰り返し問い合わせを行い，HTTP経由でMapperから，Mapタスクの出力の位置つまり必要なパーティションの位置を取得すると，パーティションの存在するノードからReduceタスクの配置されているノードへコピーし，ResourceEstimaterによってそれらのパーティションをマージして，Reducerがレコードを Reduce関数に渡しReduce処理を行う．

Reduce処理が行われたキー・バリューはMap処理と同様にOutputCollectorに渡される．処理結果が最終出力の場合，Reduceタスクの数だけ出力としてファイルへの書き出しを行い，複数回のMap Reduce処理が行われるジョブの場合，この出力が再びマージされ次のMapフェーズの入力として渡される．

2.4 Hadoop をメニーコア環境で動作する場合の問題点

Hadoop には，完全分散モード・疑似分散モード・スタンドアロンモードの三つの動作モードがあり，用途によって使い分けられている．完全分散モードとは，ネットワーク上の多数のノードを用いて通信し処理を分散させたい場合に使用するモードで，一般的に使用されているモードと考えられる．スタンドアロンモードとは，Hadoop を用いてユーザプログラムを試験的に動作させるために用意されているモードで，Map Reduce 共に一つのプロセス内で動作する．疑似分散モードとは，一つのノード内で擬似的に処理を分散させたい場合に使用するモードであり，Hadoop を用いて一つのノード内で並列処理を行う場合はこのモードが使用される．

しかし，第 1 章でも述べた通り，Hadoop は一つのノード内で並列処理を行う場合には最適ではない部分もある．既存仕様の Hadoop をメニーコア環境で最適に動作させるには大きく分けて二つの問題点がある．以下でこの二つの問題点を挙げる．

2.4.1 中間データの取扱

Hadoop では，2.3 節で述べた通り，出力された中間データはファイルとして書き出され，Reduce タスクが必要とするパーティションをノード間でコピーし取得している．メニーコア環境で Hadoop を動作させる場合，当然データのやりとりはノード内で行われる．この場合ノード間でファイルをコピーする必要はなく，Map の出力をファイルとして書き出すことは不必要だと考えられる．

さらに，標準で使用される HDFS は，多数のノードを利用することに適した分散ファイルシステムであり，レプリケーションのように冗長化など分散のためのオーバーヘッドも大きく，ノード内でデータを管理し動作させるには適していないと考えられる．

2.4.2 タスクスケジューリング

Hadoop には三つの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，標準で設定されているスケジューラである．分割したタスクを優先順位を付けたリストで管理し，該当ノードにおいてローカルに配置されている入力スプリットか，もしくは近くに位置する他のノードに配置されている入力スプリットに対応するタスクを割り当てる．プリエンプションは実装されていない．

- フェアスケジューラ

  マルチユーザに対応したスケジューラである．長期的にみて各ユーザに公平にクラスタの処理能力を配分する．ユーザ毎で用意されるジョブのプール内で FIFO スケジューリングを行い，プリエンプションも実装されている．

- キャパシティスケジューラ

  フェアスケジューラと同様マルチユーザに対応したスケジューラであるが，フェアスケジューラとは異なるアプローチをとる．多くの階層化されたキューから構成され，各キューにはそれぞれキャパシティが設定されており，各キュー内で FIFO スケジューリングを行う．

これらの三つのスケジューラはネットワーク上の多数のノードにタスクを配置する場合に有効なスケジューラであり，一つのノード内で複数のタスクを実行することに関しては単純なスケジューリングしか行われていない．一つのノード内で複数のタスクを並列実行する場合を考慮すると，より有効なスケジューリング手法を組み込むことができる可能性があると考えられる．

さらに，各タスクの処理時間の差による処理終了待ちに関しても改善の余地があるといえる．タスクのコア毎の処理時間軸を図 2 に示す．Hadoop では，処理を均等にするためタスクに割り当てられる入力スプリットはほぼ同サイズに分割されているが，キー・バリューに対する Map 関数及び Reduce 関数内での処理が複雑であると，処理時間の差は広がり，これに伴い処理終了の差が生まれてしまう．例えば，ある Map タスクの処理が非常に長引いた場合，他の Map タスクの処理を完了したコアはアイドル状態のまま図 2 のように処理の遅れている Map タスクの処理終了を待ちつづけてるという状態を引き起こす．メニーコア環境で処理を行う場合，この状態では CPU リソー

スを有効に利用することができない．CPU リソースを有効利用しスループットをより上昇させるには，スケジューラ側で適切なタスクの配置・再配置などを行い，こういった処理終了の差による CPU リソースのアイドル時間を可能な限り減少させる必要があると考えられる．

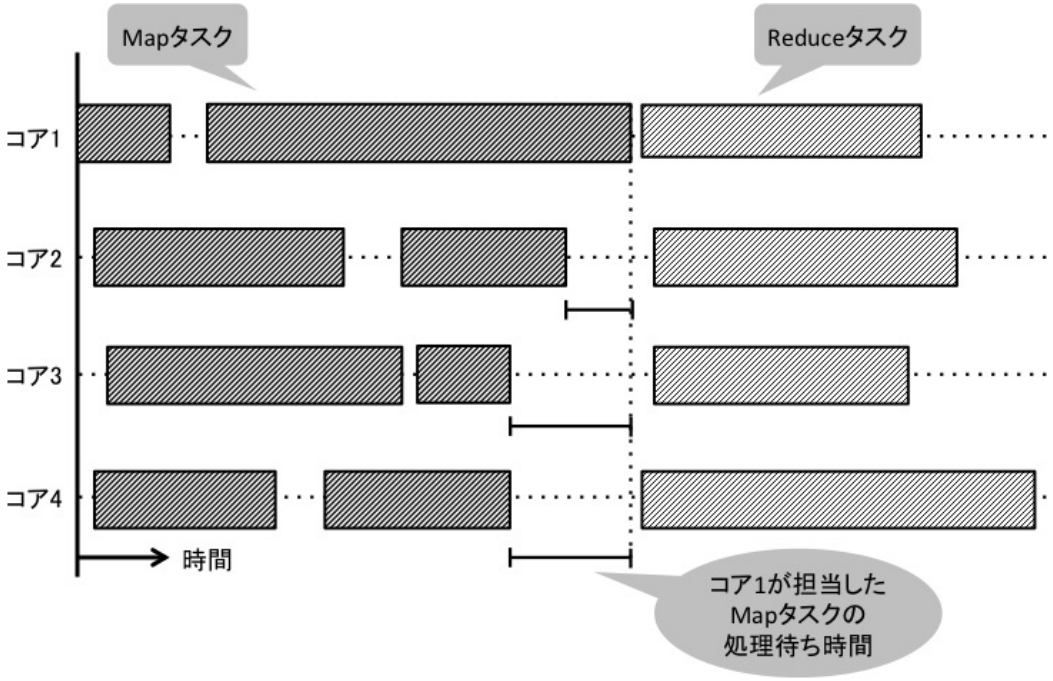

図 2: タスクのコア毎の処理時間軸

3 提案と実装

3.1 提案

既存仕様の Hadoop では，設定ファイルからスロット数を読み込み，初期段階で設定した後はスロット数は固定である．本研究では，2.4.2 節で述べたタスクスケジューリングに関しての問題解決に向け，システム内の負荷状況の検知によるスロット数の動的調整手法を提案する．提案する Hadoop に組み込む追加機能とその流れを図 3 に示す.

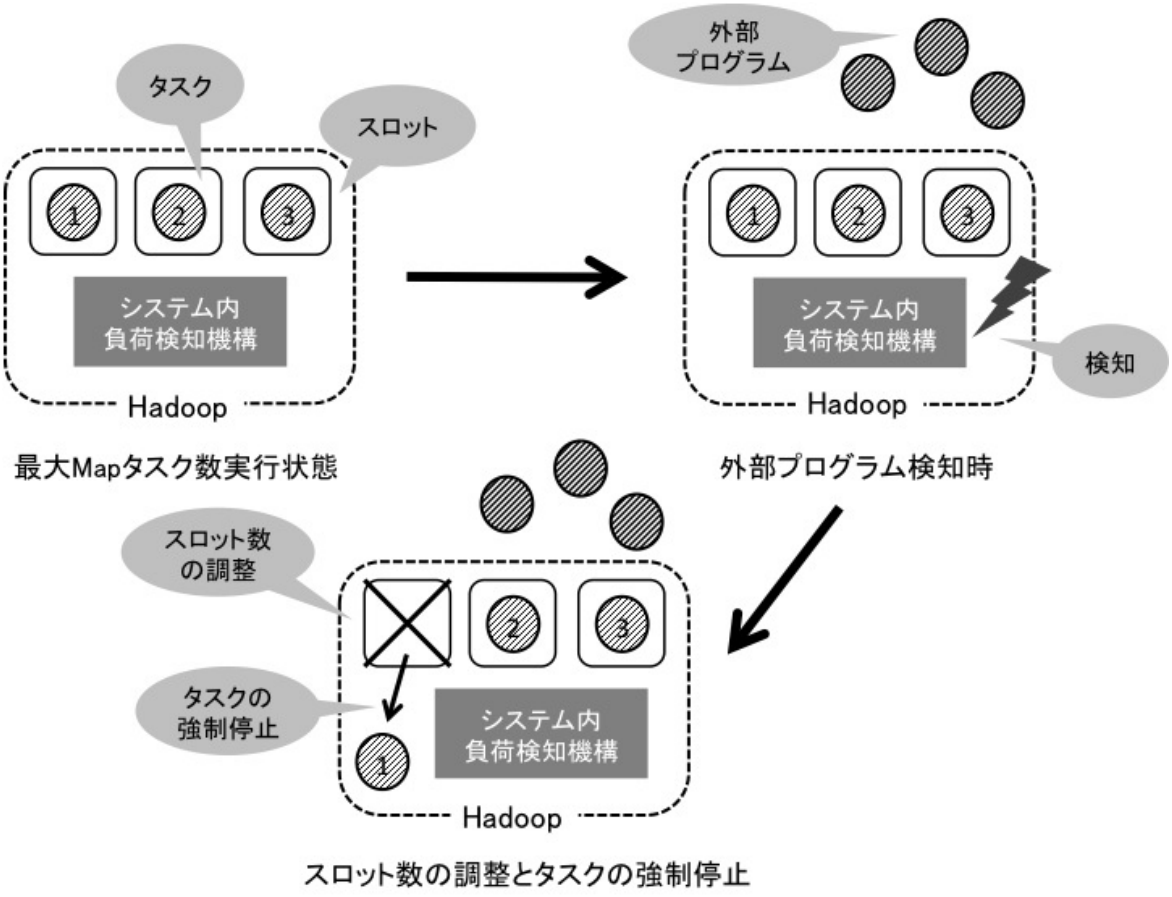

図 3: 提案する追加機能

Hadoop にシステム内の負荷状況を検知する機構を組込み，検知したプログラムから近似的にスレッド数を割り出し，このスレッド数を用いて，ノード内で同時並列実

行可能な最大 Map タスク数（Map スロット数）を動的に調整する．スケジューラは呼び出しの度に Map スロット数を読み込むため，最新の Map スロット数におけるタスクスケジューリングが行われる．さらに，Map スロット数より多い Map タスクを実行していた場合は，Map タスクを強制停止しスロット数に合わせて実行中の Map タスクの数を調整する．これにより，スレッド数の過多によるスレッド管理のオーバーヘッドを減らし，外部プログラムの実行による Hadoop タスクの並列実行制御の乱れの影響を低減し，スループットの降下抑制を図る．

3.1.1 起動スレッドについて

Map スロット数の動的調整において，1 スロットつまり 1Map タスクにどれくらいのスレッドが起動実行されているかが重要となる．

そこで，Map スロット数を変動させ，初期割り当て実行時の TaskTracker スレッドグループのスレッド数，つまり純粋に Map スロット数だけ Map タスクを実行しているときの実行スレッド数を調査した．表 1 がその結果である．

Map スロット	14	18	22	24	26
スレッド数	72	88	104	112	120

表 1: 最大 Map タスク実行時スレッド数

調査結果から考慮すると，この時点では 1Map タスクにつき 4 スレッド起動されていることが分かる．Map スロット数の削減は最低でもこの値よりスレッド数が多い場合に行うことが好ましいと考えられる．この結果を元に次の実装を行う．

3.2 実装

既存仕様の Hadoop の拡張機能として本研究の実装を行う．この追加実装による処理の流れを図 4 に示す．

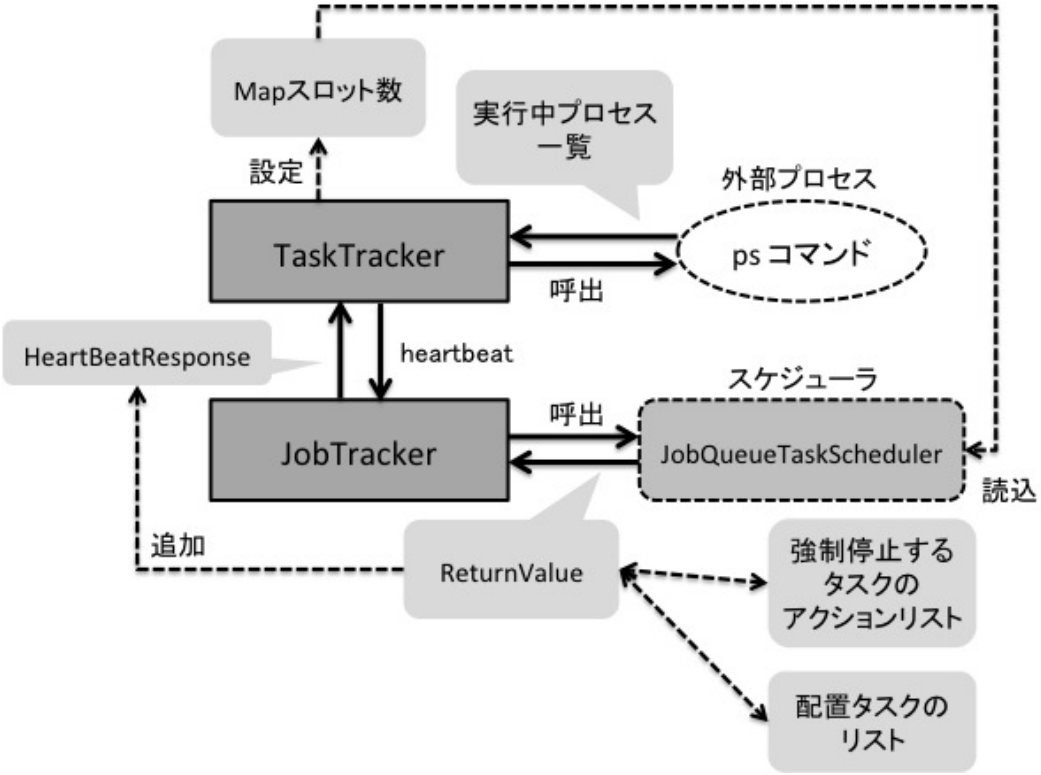

図 4: 追加実装による処理の流れ

3.2.1 システム内の負荷状況の検知と Map スロット数の調整

システム内の負荷状況を検知する機構をノード内で直接タスクを管理する TaskTracker に組み込む.

まず，暫定外部スレッド数と初期設定 Map スロット数を格納するメンバ変数を新たに用意する．Map スロット数を格納するメンバ変数は存在するが，実装にあたりこの値を動的に変化させる必要があるため，固定変数として読み出せるように個別に用意する．TaskTracker のデーモン起動時に呼び出されるメソッド内で初期化が行われるが，このとき設定ファイルから読み出される Map スロット数を初期設定 Map スロット数にも格納しておく．暫定外部スレッド数は 0 に初期化しておく．

TaskTracker が JobTracker へ HeartBeat を送りそのレスポンスを受け取る仕組みは，TaskTracker が JobTracker の heartbeat メソッドを呼び出し，その引数として Task-

TrackerStatus（TaskTracker の状態）などを渡す．そして，その戻り値として HeartBeatResponse を受け取る．heartbeat メソッド内でスケジューラが呼び出されるため，システム内の負荷状況を検知するプログラムをこの直前に記述する．

次に，独立した外部プロセスでコマンドを実行することができる Runtime クラスの exec メソッドを用いて，ps コマンドを実行する．ps コマンドの実行により，ノード内で動作しているプロセス情報の一覧が取得できる．この出力をパイプで渡し複数回の文字列操作を経て，最終的にはステータスが R・R+，つまり実行中か実行待ちのプロセスだけを出力する．exec メソッドで実行した戻り値として Process クラスのオブジェクトが生成されるため，この Process から InputStream の生成を経て，BufferedReader を生成する．

最後に，この BufferedReader から一行ずつ出力を取得しカウントし，このカウントを実行中の外部プロセスのスレッド数と近似的に認識する．heartbeat メソッドは頻繁に呼び出されるため，Map スロット数の激しい変動を抑えるために，暫定外部スレッド数と比較して変化があった場合は暫定外部スレッド数を取得したスレッド数に書き換え，一致した場合は暫定外部スレッド数を取得したスレッド数に書き換えた後，Map スロット数をセットする．具体的には，暫定外部スレッド数が 4 以上である場合，その数に応じて初期設定 Map スロット数から一定数を減算する．この値を setMaxMapSlots メソッドを用いて Map スロット数に格納する．

3.2.2 オーバースロットによるタスクの強制停止

システム内の負荷状況の検知による Map スロット数の調整に連動し，スケジューラ呼出時に調整した Map スロット数より多い Map タスクを実行していた場合は，Map タスクを強制停止しスロット数に合わせて実行中の Map タスクの数を調整する．この実装として，JobTracker の heartbeat メソッド内のタスク処理と，デフォルトで設定されているタスクスケジューラであるジョブキュータスクスケジューラ (JobQueueTaskScheduler) を拡張する．

- ReturnValue

  既存の JobQueueTaskScheduler は戻り値を Task のリストのみとしている．今回の実装にあたり，戻り値を Task のリスト（assignedTasks）と TaskTrackerAction というタスクを強制停止するためのアクションを含む TaskTracker がタスクに対して行うアクションのリスト（killList）にする必要があるため，これらをメンバ変数とする戻り値のクラスとして ReturnValue を新たに作成した．JobQueueTaskScheduler はスケジューリングを行った結果を ReturnValue のメンバ変数に格納し，この ReturnValue を戻り値として返す．

- JobQueueTaskScheduler

  スケジューリングを行う JobQueueTaskScheduler のメイン部分である assignTasks メソッドの拡張を行う．

  assignTasks では，availableMapSlots に空き Map スロット数を算出して格納し，この availableMapSlots の数だけ可能な限りのタスクの配置を行う．availableMapSlots は未処理タスク数に連動しており，そのときに最低限必要な空き Map スロットが格納される．これに対し，純粋な空き Map スロットを表す availablePureMapSlots を用意し，単純に Map スロット数から実行中 Map タスク数を減算した数を格納する．

  システム内の負荷状況の検知による Map スロット数の調整が行われ Map スロット数が削減されると，実行中の Map タスク数が Map スロット数を上回り，availablePureMapSlots がマイナス値となる．このマイナス値の絶対値をオーバースロット数とみなし，オーバースロット数だけ実行中の Map タスクの強制停止を次の手順で行う．

  まず，実行中のジョブから実行中の全ての Map タスクの TaskInProgress のリストを取得し，配列に格納する．double 型の二次元配列を用意し一つ目のインデックスを共通として，TaskInProgress の getProgress メソッドを用いて取得したタスクの進捗と，TaskInProgress の配列のインデックスを格納する．

次に，この二次元配列をタスクの進捗を基準に昇順にソートする．ソート時に TaskInProgress の配列のインデックスも同時にスワップする．

最後に，昇順にソートされた二次元配列から順にオーバースロット数だけ，TaskInProgress の配列のインデックスを取り出し，該当する TaskInProgress を取得する．進捗の少ない Map タスクを選択することで，強制停止するリスクを軽減する．TaskInProgress から最新の TaskAttemptID を取得し，該当 Map タスクが完了していないかチェックをしてから TaskAttemptID を引数に KillTaskAction を生成し，killList に追加する．

- JobTracker

  JobTracker は heartbeat メソッド内で JobQueueTaskScheduler の assignTasks メソッドを呼び出す．この戻り値の ReturnValue から killList を取得して killTasksList に追加する．

  killTasksList に登録されているタスクを強制停止するためのアクションはタスクを起動するアクションなどとひとまとめにされ，HaertBeatResponse の要素として TaskTracker に渡され，アクションが実行される．

4 性能評価

本提案を実現するために既存仕様の Hadoop に対して追加実装を行い，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 2 に示す．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 プログラムを採用し，I/O 処理よりも CPU 内の演算処理が主なプログラムをシミュレートするためガウス分布によるランダム時間の遅延を挿入し改変している．また，中間データの取扱に関する問題点について，HDFS より最適な Hadoop に実装されている LocalFileSystem を採用し，これを RAM ディスク上に展開した．これによりネットワーク及びディスクの I/O を無くし，後に行う予備評価実験及び評価実験の実行においてボトルネックとならないようにした．

Hadoop のバージョン	hadoop 0.20.203
ファイルシステム	LocalFileSystem
CPU	(AMD Opteron 12-core 6168 / 1.9GHz) x2
メモリサイズ	32GB
OS	CentOS 5.7
カーネル	Linux 2.6.18

表 2: 実行環境

本研究の提案の目的はスレッド数の過多によるスレッド管理のオーバーヘッドを減らし外部プログラムの実行による Hadoop タスクの並列実行制御の乱れの影響を低減することである．このため，最適な起動スレッド数で処理を行うパラメータを知る必要がある．入力データを 500MB，ベンチマークプログラムに挿入するランダム時間の生成に用いるガウス分布を平均 0 分散 500,000 とし，予備評価として既存仕様の Hadoop を用いて Map スロット数及び Reduce スロット数を調整し実行時間を計測した．計測結果から分かった最適パラメータを表 3 に示す．表 3 の最適パラメータのときベンチマークプログラムの実行時間は約 130 秒であった．ガウス分布のパラメータをこの値

にした理由は，本研究の提案はある程度 CPU 処理時間の長いタスクである場合に効果を発揮すると考えており，予備実験を繰り返し評価に使用するのに適切だと判断したパラメータであったからである．

Map スロット数	24
Map 分割数	36
Reduce スロット数（Reduce 分割数）	6

表 3: 最適パラメータ

4.2 評価結果

ベンチマークプログラムを用いて評価実験を行った．入力データを 500MB とし，ベンチマークプログラムのランダム時間の生成に用いるガウス分布を平均 0，分散 500,000 と設定した．外部プログラムとして無限ループを回す C 言語プログラムを作成しこれを使用した．ジョブの投入を行ってから 10 秒後（Hadoop が起動してから Map タスクの配置を開始する目安の時間）に外部プログラムの実行を開始し，Map タスクの強制停止及び再配置が行われ，Map タスクを強制停止するオーバーヘッドも含め，調整された Map スロット数でどれだけ外部プログラムの実行による Hadoop タスクの並列実行制御の乱れの影響を低減することができるか評価した．表 4 は，削減 Map スロット数と外部スレッド数のパターンを示す．各パターンをそれぞれ 5 回ずつ試行し，計測した実行時間の平均をとった．尚，外部スレッド数の下限は 3.1.1 節で調査したスレッド数を基準にしている．

削減 Map スロット数	1	2
外部スレッド数	4〜11	8〜15

表 4: 評価パターン

評価結果を図 5 に示す．左のグラフはシステム内の負荷状況検知時の削減 Map スロット数を 1，右のグラフはシステム内の負荷状況検知時の削減 Map スロット数を 2

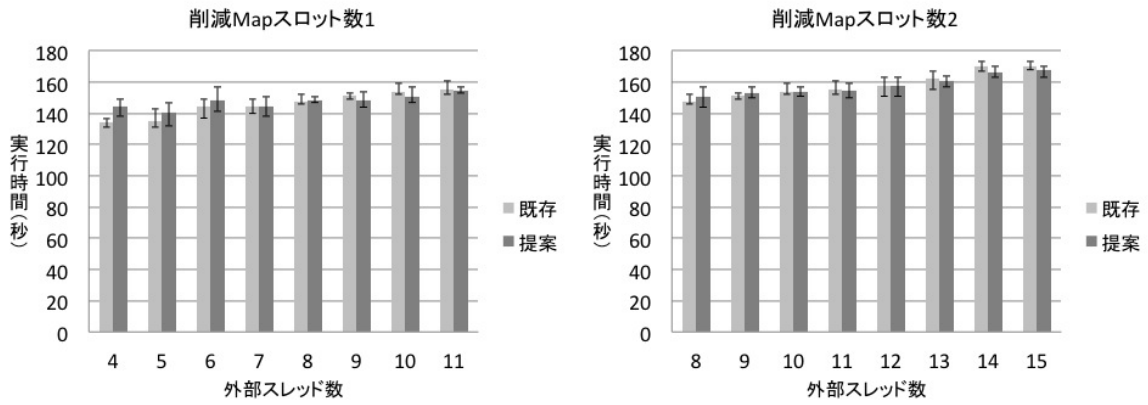

図 5: 評価結果

に設定したパターンで，提案手法と既存手法の Hadoop の実行時間を比較した．左の薄い色のグラフが既存手法，右の濃い色のグラフが提案手法である．誤差表示には観測した最高値と最低値を用いた．

まず，削減 Map スロット数を 1 に設定したパターンでは，外部スレッド数が 9 以上で既存手法に比べ数ミリ秒から数秒の影響低減がみられた．外部スレッド数が 7 であるときもわずかに影響低減がみられるが，外部スレッド数が 8 であるときは実行時間が上回っているため分散による観測外の誤差であると考えている．同様に，外部スレッド数を増やすにつれ実行時間が単純に右上がりにならないのは分散による少ない誤差であるだろう．外部スレッド数が 4 及び 5 であるとき既存に比べ大幅に遅延しているのは，外部スレッド数がこの程度であると Map スロット数を削減する必要性が低く，さらにタスクの強制停止によるオーバーヘッド（1Map タスクにつき強制停止してから再配置まで約 6 秒かかる）がカバーしきれなかったことによると考えられる．

次に，削減 Map スロット数を 2 に設定したパターンでは，外部スレッド数が 10 以上で，削減 Map スロット数を 1 に設定したパターンと同様に既存手法に比べ数ミリ秒から数秒の影響低減がみられた．

4.3 考察

既存手法と提案手法の実行時間を比較すると，削減 Map スロット数を 2 に設定したパターンでは全体的に影響低減することができている．このパターンで効果がみられなかった外部スレッド数が 9 であるときについては，削減 Map スロット数を 1 に設定したときは影響低減していることから，外部スレッド数が 9 より多い場合は本研究の提案が有効であることが評価結果から分かった．

しかし，これ以上削減 Map スロット数増やす場合は，非常に多くの外部プログラムが実行されているときにしか効果は見込まれないだろう．これはスレッド管理のオーバーヘッドの削減のための Map スロット数の動的調整による実行スレッド数の削減が，タスクの並列実行による効果を上回ることが困難であるということに起因すると考えられる．

本来外部プログラムは意識的に動作させるものでない場合も多く，ユーザがそういった Hadoop の実行環境での外部プログラムの実行状態を意識することなく Hadoop 側が動的にスロット数の調整を行い影響低減を実現することが好ましい．パラメータとして削減 Map スロット数と対応する外部スレッド数を設定しているが，今回の実験パターンを例として，外部スレッド数が 9・10 の場合は削減 Map スロット数を 1，外部スレッド数が 11 以降の場合は削減 Map スロット数を 2 といったように，外部スレッド数から適切な削減 Map スロット数を算出し調整できるようにする必要がある．これについては，具体的な算出式を提案できるように今後さらなる調査を続けていきたい．

5 まとめと今後の課題

本研究では，Hadoop を用いたプログラム実行時の外部プログラムの実行によるスレッド数過多に起因するスレッド管理のオーバーヘッドの削減手法として，システム内の負荷状況の検知によるスロット数の動的調整手法を提案した．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 プログラムを改良し CPU 処理を仮想的に変動させたプログラムを用いて評価を行った．その結果，一部の評価パターンにおいて外部プログラムの実行による Hadoop タスクの並列実行制御の乱れの影響を低減することができた．

本研究の今後の課題として，外部スレッド数から適切な削減 Map スロット数を得るための算出式など，より最適なスロット数の動的調整手法を提案することが考えられる．また，検知したプロセス数を実行スレッド数とみなしているため，マルチスレッドのプログラムなどに対しても正確なスレッド数を検知できるよう改善が必要であると考えられる．さらに，Hadoop のスケジューリングにおいて，タスクの処理時間の差による処理終了待ちの回避など他にも最適化の余地がある．Hadoop をメニーコア環境で動作する場合のタスクスケジューリング全体を考慮し，さらなる高速化を目指したい．

謝辞

本研究を進めるにあたり，多大なご尽力，ご指導を頂いた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，松井俊浩准教授に深く感謝いたします．また，日常の議論を通じて多くの知識や示唆を頂き，ご協力をして頂いた松尾・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熊崎宏樹氏，伊藤翼氏，佐藤貫大氏，中居新太郎氏，水野航氏，里見優樹氏に感謝いたします．そして，研究生活を支えてくれた両親に感謝いたします．

参考文献

[1] Tom White, “Hadoop”, オーム社, 2011.

[2] 大田一樹・下垣徹・山下真一・猿田浩輔・藤井達朗・濱野賢一郎, “Hadoop 徹底入門”, 翔泳社, 2011.

[3] ApacheHadoopProject, http://hadoop.apache.org/

[4] ApacheHadoopProject:HadoopDistributedFileSystem, http://hadoop.apache.org/hdfs/

[5] ApacheHadoopProject:HadoopMapReduce, http://hadoop.apache.org/mapreduce/

[6] ApacheHadoopProject:HadoopCommon, http://hadoop.apache.org/common/

[7] HadoopWiki, http://wiki.apache.org/hadoop/FrontPage