集約処理を用いた MapReduce 最適化手法の提案と実装

小沢 健史[†] 鬼塚 真[†]
福本 佳史[†] 盛合 敏[†]

本稿では，MapReduce で行う処理のうち，部分集約が可能な処理を高速化する手法を示す．部分集約が可能な処理とは，集約時に結合法則と交換法則が成立する処理のことを指す．部分集約ができる処理に対して，既存研究では特有の処理系を新たに作成することにより高速化を行っていた．しかし，これらの手法は MapReduce の仕組みを大幅に変更する必要があることから，Hadoop に組み込むのは 困難であった．そこで本研究では，Hadoop への実装コストが低く抑え，高速化をおこなう Map Multi-Reduce の提案を行う．Map Multi-Reduce は，MapReduce に Record Reduce と Local Reduce の 2 つの機能を追加した，MapReduce の拡張版である．提案手法の実装を行うにあたり行った Hadoop への変更量は，Record Reduce で 約 200 行，LocalReduce で約 300 行と小さい．このように少ない変更量にも関わらず，ディスク IO とネットワーク IO が削減され，実験により 2TB WordCount を行う際に，処理速度が 1.7 倍になることを確認した．また，100GB のデータに対して WordCount を行った際に，最大で Map 処理と Reduce 処理間のデータの受け渡しを 50% に削減できることを確認し，より大きな入力データに対して，データの受け渡しコストをより削減できる可能性があることを示す.

MapReduce optimization using mapper-side aggregation

Tsuyoshi OZAWA,[†] Makoto ONIZUKA,[†]
Yoshifumi FUKUMOTO[†] and Satoshi MORIAI[†]

In this paper, we propose a MapReduce optimization by using mapper-side aggregation designed for aggregation queries - the queries consisting of the aggregation operations that satisfy both the commutative and the associative law. The mapper-side aggregation has been applied in different platforms, however, it is difficult for related work to be embedded within existing MapReduce framework like Hadoop, because its mechanism of task scheduling or monitoring is different and MapReduce framework does not provide inter-process communication facilitiy. To solve this problem, we prototype Map Multi-Reduce, while preserving MapReduce semantics with small modification against Hadoop. Map Multi-Reduce is an extension of MapReduce consisting of two features - Record Reduce and Local Reduce. Record Reduce aggregates the intermediate data of MapTask in memory and is implemented in only 200 LOC. Local Reduce aggregates the the outputs of multiple MapTasks in same machines and is implemented in only 300 LOC. Map Multi-Reduce improves 1.7 times faster in WordCount processing against 2TB dataset and cuts down shuffle cost by 50% against 100GB dataset.

1. はじめに

記憶装置・計算機・インターネットの発展により，産業界において取り扱うデータの量が爆発的に増加しつつある．蓄積された大量のデータを安く，実用的な速度で処理するために，安価な計算機クラスタを用いて効率的に分散計算を行うフレームワークに注目が集まっている.

† NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center

特に，MapReduce は最も普及している分散処理フレームワークの 1 つであり，Google，Facebook，Yahoo! をはじめとする多くの企業に利用されている[3),4),9),10)]．MapReduce では，並列処理をおこなう Map 関数と，Map 関数の結果を集約する Reduce 関数を記述するだけで，計算機の故障や，計算機間の同期処理を意識することなく，大規模な分散処理を行うことができる[4)]．Hadoop は，MapReduce のオープンソース実装であり，Yahoo! や Cloudera，Hortonworks が主導で開発を進めている[2)]．Hadoop は，単純なオンプレミス環境だけでなく，クラウド環境上の

ⓒ 2012 Information Processing Society of Japan

サービスとしても提供され，多くの企業で利用されている[1]．

MapReduce には，タスクを実行するために 2 種類のプロセスがある．1 つは Mapper ，もう 1 つは Reducer である．Mapper は Map 関数を実行する MapTask を処理し，Reducr は Reduce 関数を実行する ReduceTask を処理する．MapReduce で行われる処理の中には，Reducer が 1 つであることを強いる処理がある．例えば，WordCount を行った後に，上位数件を表示する処理などである．このような処理の場合，全ての MapTask の結果が 1 つの Reducer へ転送されるため，Reducer が行う処理が増大し，過負荷になってしまい処理に時間がかかるという問題がある．

この問題の対策は 2 つある．1 つ目は，in-mapper combining というテクニックを利用して Reducer への負荷を減らす手段である[7]．in-mapper combining は，ユーザがプログラムを書き直すことで，Map 関数の中で集約処理を行う．in-mapper combining では，ほとんどの処理をメモリ内で完結し，集約処理を行うことができるため，Mapper の ディスク IO と Mapper - Reducer 間の 通信量を最小限に抑えることができる．2 つ目の方法は，Combiner という機能を利用する方法である．Combiner は， Hadoop で提供されている基本機能であり，1 つの MapTask の中で出力される中間ファイルに対し，集約処理を行うための仕組みである．Combiner を利用することにより Mapper と Reducer の間の IO を減らすことができる．しかし，in-mapper combining，Combiner どちらを用いても，集約処理の適用範囲は 1 つの MapTask の結果に閉じているため効果が限定的である．特に計算機がマルチコアであるような環境では，多数の MapTask が 1 台の計算機で実行されることが多い．すると，MapTask の中間出力ファイルは同一計算機内で複数生成されるにも関わらず，それらを計算機ごとに集約することができない．よって，これらの手法だけでは計算機の増加/入力データ量の増加に伴い，Reducer に渡されるデータ量も増加してしまい，Reducer が処理のボトルネックになってしまう - つまり，スケーラビリティに限度があるという問題点が生じる．また，in-mapper combining は，Combiner を用いた手法よりも良い性能を示すことが報告されている[7] が，ユーザがメモリ管理を自前で行う必要があるため，メモリ不足に注意を払う必要がある，という問題がある．

そこで本稿では，既存の Combiner を強化した，Map Multi-Reduce を提供する．Map Multi-Reduce は，2 つの機能を Hadoop に追加した，Hadoop の拡張版である．1 つめの機能は in-mapper combining をフレームワークとして提供するための機能である．これにより，フレームワーク側で細かなメモリ管理を行うことができるため，メモリ不足で MapTask が停止することを防止することができる．2 つめの機能は MapTask を実行した計算機毎に集約処理を行う機能である．これにより，データの規模増大に対して，高いスケーラビリティをもつ MapReduce を実現することができる．

本手法は，Combiner を利用できる処理において有効である．Combiner を利用できる処理とは，つまり，MapReduce で処理を記述した際に，演算順序が依存しない一可換法則と結合法則を同時に満たすー処理である．処理全体が演算順序に依存しない処理の組み合わせである代表例として，複数の単語数を数え上げる WordCount や，単語間の共起頻度計算がある．また，処理の一部が演算順序に依存しない計算の例として，平均や分散計算などがある．

本稿では，Map Multi-Reduce の実装と評価を行い，2TB のデータ処理下において，1.7 倍の処理速度向上が見込めることが確認した．また，計算機毎に集約を行う機能により， 100GB のデータを処理する際に Mapper - Reducer 間の通信量を 50% 削減できることを確認し，より大きなデータに対してより多くの通信量を削減できることを示す.

以下，2 章にて，MapReduce の基本的な挙動と in-mapper combining/Combiner を用いた MapReduce 処理の効率化とその問題点について述べる．3 章にて，in-mapper combining/Combiner の問題点を解決した Map Multi-Reduce を提案する．4 章で，Map Multi-Reduce の実装について述べ，5 章では，実験による高速化の評価方法を示す．6 章で関連研究について述べ，7 章で本論文をまとめる．

2. 前 提 知 識

2.1 MapReduce

図 1 を用いて，本研究が改善の対象とする MapReduce について説明する．MapReduce ジョブは Map フェーズと Reduce フェーズの 2 フェーズから構成される．Map フェーズでは入力データ（D）を読み込んで Key/Value ペア (K_1, V_1) を生成し，それを入力として各ペアに対してユーザが定義した Map 関数を実行し，新たな Key/Value ペアリスト (K_2, V_2) を中間データとして出力する．Reduce フェーズでは，ま

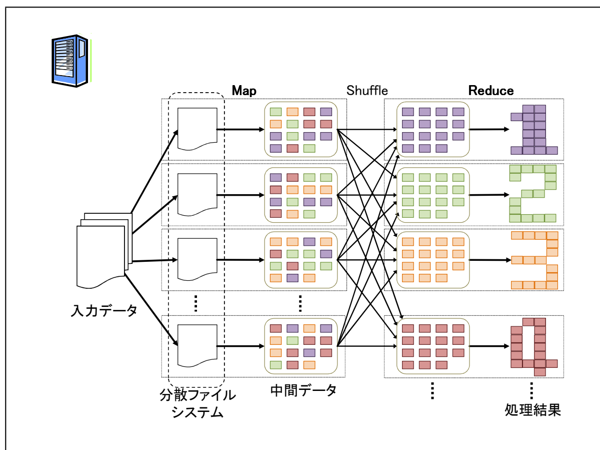

図 1 MapReduce の概要

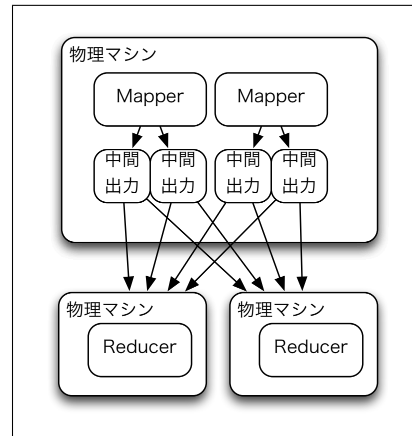

図 2 中間出力の集約を処理を行わないときの処理フロー

ず中間データを Key 毎にグルーピング（Shuffle）する．そして各グループを入力としてユーザが定義した Reduce 関数を実行し，結果として Key/Value ペアリスト (K_3, V_3) を出力する．この一連の流れを式として表現すると，以下のようになる．

$D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2\}$

$shuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\}$

$reduce(K_2, \{V_2\}) \rightarrow (K_3, V_3)$

MapReduce 処理は複数のノード（コンピュータ）をネットワークで相互接続したクラスタ内で行われる．クラスタはマスタ（Controller）ノードと複数のスレーブノードで構成されており，クラスタ上に DFS（分散ファイルシステム）が構築され，入力データが格納される．MapReduce ジョブを開始すると，まずマスタが入力データをユーザの指定したサイズで分割し複数の InputSplit を生成する．次にマスタが InputSplit の数だけ MapTask を生成し，各スレーブに割り当てる．MapTask が割り当てられたスレーブでは Mapper が起動され，ユーザが定義した Map 関数が実行されて，Key/Value 形式の中間データが出力される．中間データは同じ Key 値を持つものが一つのスレーブに集まるようにネットワークを介して相互に移動（shuffle）される．マスタはユーザが定義した数の ReduceTask を生成し，各スレーブに割り当てる．このとき，中間データは Key 毎に分配される．ReduceTask が割り当てられたスレーブでは Reducer が起動され，ユーザが定義した任意の Reduce 関数が実行されて，新たに Key/Value 形式の処理結果が DFS に出力される．

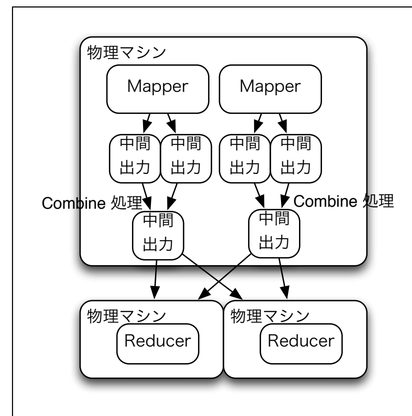

図 3 中間出力の集約を Combine 処理を用いて行ったときの処理フロー

2.2 Mapper 毎の集約処理による MapReduce 処理の最適化

2.2.1 Combine 処理

Combine 処理は， Hadoop で提供されている基本機能であり，1 つの MapTask の中で出力される中間出力に対し，集約処理を行うための仕組みである．図 2 に，集約処理を行わない MapReduce 標準の処理フローを，図 3 に，Combine 処理を利用したときの処理フローを示す．Combine 処理に利用できるクラスは，Reduce クラスを継承したクラスのみである．Combine 処理の前提として，Reduce 処理を行うことによりデータサイズが減ることが多いという前提がある．例えば，WordCount の場合，

$(K_1, V_1) = (apple, (1, 1, 1, 1, 1))$

$(K_2, V_2) = (sweet, (1, 1))$

となっているとき，Reduce 関数を Combine 処理として適用することで

ⓒ 2012 Information Processing Society of Japan

```
1  class Mapper
2    method Setup
3      H = new AssociativeArray
4
5    method Map(docid a; doc d)
6      for all term ∈doc d do
7        H{t} ←H{t} + 1
8
9    method Cleanup
10     for all term t ∈H do
11       Emit(term t; count H{t})
```

図 4 WordCount における in-mapper combining の例

$(K_1, V_1) = (apple, (5))$
$(K_2, V_2) = (sweet, (2))$

と変換することができ，データ量を減らすことができる．この性質により，Mapper と Reducer の間の IO を減らすことが可能となる．Hadoop では，Combine は MapTask の一部として実現されており，

$$MapTask \begin{cases} D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2\} \\ localshuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\} \\ combine(\{K_2, \{V_2\}\} \rightarrow \{K_2, V_2\} \end{cases}$$

$$Shuffle \left\{ shuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\} \right.$$

$$ReduceTask \left\{ reduce(K_2, \{V_2\}, P_r) \rightarrow (K_3, V_3) \right.$$

と表すことができる．Combine 処理が性能向上に寄与する場合，Map 関数の出力データ量を $|V_m|$，Combine 関数の出力データ量を $|V_c|$ とすると，$|V_m| > |V_c|$ が成立する．

ただし，現在の Hadoop 上の仕組みでは，Combine 処理の適用範囲は 1 つの MapTask の中に閉じているため，計算機内で実行された複数の MapTask の出力を集約することはできない．

2.2.2 in-mapper combining

in-mapper combining は，Map 関数の中で集約処理を行うように，ユーザがプログラムを書き直す最適化手法である．in-mapper combining の処理フローを図 5 に， WordCount における in-mapper combining の例を図 4 に示す．Combine 処理と in-mapper combining の違いは，Combine 処理 が Hadoop の管理するバッファ上の中間出力を対象とするのに対し，in-mapper combining は Map 関数内のユーザが管理するメモリ上で集約処理を行ってしまい，集約済みの値を Reducer に渡す部分である．

メモリを利用することで，ほとんどの処理をメモリ内で完結させることができるため，Mapper の ディスク IO と Mapper - Reducer 間の IO の多くを減らすことができる．

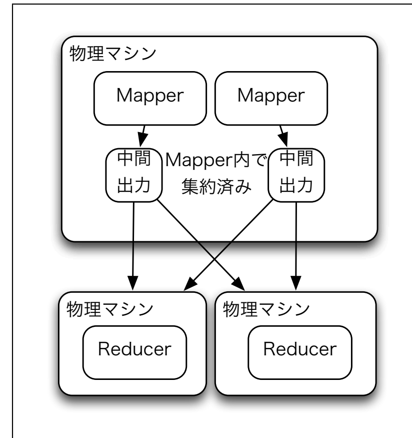

図 5 中間出力の集約を in-mapper combining を用いて行ったときの処理フロー

Hadoop では，in-mapper combining は map 処理の一部として実現されるので，集約済みの値を V_2' とおくと

$D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2'\}$
$shuffle(\{K_2, V_2'\}) \rightarrow \{K_2, \{V_2'\}\}$
$reduce(K_2, \{V_2'\}) \rightarrow (K_3, V_3)$

と表すことができる．in-mapper combining が高速化に寄与するのは，Map 関数の出力データ量を $|V_m|$，in-mapper combining による出力データ量を $|V_{im}|$ とすると，$|V_m| > |V_{im}|$ が成立する場合のみである．

in-mapper combining による集約処理の適用範囲は，Combine 処理同様 1 つの MapTask に閉じているため，計算機内で実行された複数の MapTask の出力を集約することはできない．

また，ユーザが自らメモリ管理を行う必要があるため，実装の敷居が高い．特に Hadoop 上の MapTask に メモリリークが発生した場合，Out of Memory 例外が発生し，無限に retry が実行され，強制終了するしかなくなる場合がある．

3. Map Multi-Reduce

上記のような問題を解決するため，Map Multi-Reduce を提案する．Map Multi-Reduce には，大きく分けて 2 つの機能がある:

(1) In-mapper combining 相当の機能を API として備え，フレームワーク側で提供する Record Reduce 機能.
(2) マシン毎に Combine 処理を行う Local Reduce 機能.

以下では，その詳細について述べる．

3.1 Record Reduce

Record Reduce は，フレームワーク側で in-mapper combining を提供するための機能である．Record Reduce を用いることにより，ユーザは in-mapper combining のときに必要であった MapTask 内のメモリ管理を行う必要がなくなる．Record Reduce には，2 種類の設計が考えられる:

(1) 特定のキーに関連づけられた値 values に対して，Combiner を呼び出す方法.
(2) 2 値の集約関数を呼び出し，逐次集約する方法.

Combiner を呼び出す方法とは，ユーザがプログラム内で指定した Combiner クラスを利用することで，Combine 処理を実行する方法である．現在の Hadoop の場合，ユーザが指定した jar ファイルを java プロセスにロードし，Reflection API 経由で集約関数を実行する．この方法は，Combiner をユーザが設定してさえいれば，ユーザからは透過的に実行できるという利点がある．一方で，Combine 処理を行うために，一旦出力したオブジェクトや Iterator オブジェクトを生成する必要があったり，Reducer 内でオブジェクト全体に対して繰り返し処理を行う必要があったり，Reflection API 経由で Combine 関数を呼び出す必要があるため，実行コストが高いという欠点がある．

2 値の集約関数を呼び出す方法とは，集約関数の極端な形を用いて Record Reduce を行う方法である．この方法は，逐次集約を行うためメモリ消費量が少なく，Combiner を呼び出す方法と比較して実行コストが低いが，ユーザが新たに関数定義を行う必要がある．2 値の集約関数とは，これまでに集約した結果 V_{inter} と，新たに emit された値 V_{new} を用いて，逐次集約を行う関数のことで，

$V_{inter} \leftarrow binaryCombine(V_{inter}, V_{new})$

で定義される関数のことである．ただし，binaryCombine 関数はユーザが定義する関数であり，binaryCombine 関数が呼び出される度に，V_{inter} の値は，binaryCombine 関数の中で emit された値に更新される．

例として，WordCount の例を示す．binaryCombine の中身は，ユーザにより

$V_{inter} \leftarrow V_{inter} + V_{new}$

と定義されている必要がある．このとき，キー apple に対する中間値 V_{inter} が

$V_{inter} = 13$

という値になっているとすると，次の emit は

$map(K1, V1) = (apple, binaryCombine(13, 1)) = (apple, (13+1)) = (apple, (14))$ となる．V_{inter} の値は，emit された値に更新されるため，この例では 14 に更新される．この処理を繰り返すことにより，Mapper 内での集約関数を実現することができる．

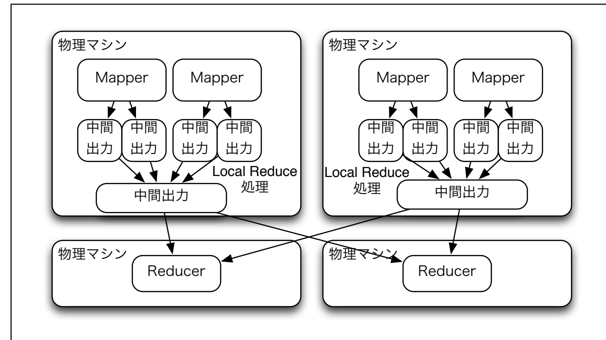

図 6 中間出力の集約を Local Reduce を用いて行ったときの処理フロー

3.2 Local Reduce

Local Reduce は，マシン毎に集約処理を行うことで，Reducer の負荷を分散するための機能である．Local Reduce のイメージ図を，図 6 に示す．Local Reduce では，Combine 処理を複数の Mapper の中間出力に対して適用することで，Reducer に渡す中間出力のファイルを小さくする．

Reducer は，本来以下の処理を行う．

(1) Mapper の出力した中間ファイルを受け取る．
(2) 自分が受け取る予定の中間ファイルを全て受け取ったら，キーでソートを行う．
(3) ユーザの定義した Reduce 関数を呼ぶ.

Local Reduce を用いることにより，1., 2. の処理の一部を，Mapper 側で行うことが可能となる．そして，Record Reduce や Combiner では不可能であった，複数の Mapper 間の計算結果のまとめあげを行うことができるので，Reduce 処理の高速化を行うことができる．

4. Hadoop 上への Map Multi-Reduce の実装

我々は，MapReduce のオープンソース実装である Hadoop 上に提案手法である Map Multi-Reduce を実装した．要点は，性能を最大限に出すための 2 通りの Record Reduce の設計・実装と，最小限の変更で Local Reduce を実現するための設計・実装である．本節では，これらの設計・実装の方法について述べる．

4.1 Record Reduce の実装

4.1.1 Combiner による Record Reduce の実現

Hadoop の場合，Mapper の引数に与えられる Map-

```
1 interface binaryCombinable<K, V> {
2   binaryCombine(K key, V currentValue, V
        newValue, MapContext context);
3 }
```

図 7 二値集約関数のインタフェース定義

Context クラス経由で，現在の MapReduce ジョブにユーザが指定した Combiner のクラス名を取得することができる．しかしこれだけでは不十分で，Combiner の呼び出しには ReduceContext クラスのオブジェクトを新たに用意する必要がある．ReduceContext クラスのコンストラクタは Hadoop の処理系内部でのみ取得することができる引数 (JobConf クラスや，TaskAttemptID クラスのオブジェクト) を与える必要があるため，新たなオブジェクトを作成するには Hadoop 処理系に手を加える必要がある．この問題を回避するため，ReduceContext を継承した CombineContext を新たに作成する．CombineContext 内ではほぼ全ての処理を MapContext の処理に委譲することで，ReduceContext オブジェクトの作成を回避し，Combiner の起動を行うことができる．以上に述べた実装方法を用いることで，Hadoop に手を入れることなく，単なるライブラリとして Combiner 呼び出しを行う Record Reduce を実装することができる．

一方で，上記の Combiner を利用した実装は，Java の Refrection API 経由で Combine 処理を何度も呼び出すため，そのオーバヘッドで処理が重くなる可能性がある．Reflection API によるオーバヘッドを測定するため，Combine 処理を定義した関数を直接ユーザに定義させ，それを呼び出すことで Record Reduce を実現する実装も行った．

4.1.2 2 値集約関数による Record Reduce の実現

Hadoop では，Combiner の呼び出しオーバヘッドが大きいため，一定のファイルサイズ以上にならないと Combiner を呼び出さない実装になっている．オーバヘッドの種類には，以下のものがあると考えられる．

(1) Combine 処理内での，イテレーション処理．
(2) Combine 関数をリフレクション経由で呼び出すことによるオーバヘッド．

これらのオーバヘッドが発生により，Combiner は in-mapper combining と同様の性能を達成できない．そこで，2 値集約による Combine 処理を実現するためのインタフェースとして BinaryCombinable を定義し，それをフレームワーク側から呼び出すように実装した．図 7 に， インタフェースの定義を示す．BinaryCombinable インタフェースは，Mapper に実装することで利用する．第一引数は key の値，第二引数はこれまで集計した中間結果であり，第三引数は新たに emit された値である．第四引数は MapContext であり，context.write() メソッド を呼ぶことで，Hadoop フレームワークに対して値を emit できる．binaryCombine 内で emit が発行されると，フレームワーク内部で保持されている HashMap に key, value が保存される．

Java の Reflection API を用いることにより，Mapper が binaryCombinable インタフェースを実装しているかどうかを見分けることができる．もし，binaryCombinable インタフェースを実装しているのであれば，Mapper は binaryCombine 専用の OutputCollector (Hadoop 内のバッファ管理用クラス) を作成し，その中で emit される度に binaryCombine 関数を呼ぶ．もし bianaryCombinable インタフェースを実装していないのであれば，通常の OutputCollector を作成し，通常の処理を実行する．

以上に述べた変更は，今回の実装では Hadoop 上に 214 行の変更を行うことで実現できた．

4.2 Local Reduce の実装

複数の MapTask の結果を集約するには，MapTask 間でプロセス通信を行い，その結果を共有する必要がある．

Hadoop では，MapTask のプロセス間でデータを共有する仕組みはサポートされていないため，新たに実装を行う必要がある．そこで我々は，MapTask 間で通信を行う仕組みとして，ローカルファイルによる共有を行う．また，複数の MapTask 出力を対象とした Combine 処理を動作させる MapTask の選出に，ファイルロックを用いる．ファイルロックの取得に成功した MapTask は，Combine 処理を行うリーダとして振る舞う．ファイルロックの取得に失敗した MapTask は，空の Local Reduce 結果ファイルを出力して，処理を終了する．以下に，今回のプロトタイプで利用している MapTask 間の通信プロトコルを示す:

(1) Map 処理を完了したら，Map 処理の中間出力ファイルを指定のディレクトリに下に移動する．ファイル名は，一意になるように TaskID に関連した名前とする．
(2) 指定されたディレクトリの下にあるファイルロックを取得しようと試みる．
(3) ファイルロックの取得に成功したら，予め指定された秒数待ち，その後 Combine 処理を起動する．

表 1 Record Reduce の実験に利用した計算機構成

CPU	2.13 GHz x 8
メモリ	8 GB
ネットワーク	1 GbE
OS	Linux カーネル 3.0.11
Apache Hadoop	6 月 17 日の trunk 版

(4) ファイルロックの取得に失敗したら，空の Local Reduce 結果ファイルを出力して，処理を終了する．

(5) 全ての MapTask の中間ファイルの結果を集約したら，ロックを解除し，その出力である Local Reduce 結果ファイルを MapTask の出力ファイルにリネームして処理を終了する．

以上に述べた変更は，今回の実装では Hadoop 上に 330 行の変更を行うことで実現できた．

提案手法では，Combine 処理を行っている最中に，MapTask が停止すると，データが失われる可能性がある．Local Reduce に対して耐故障性を担保するには，タスクが現在どの状態かを MapReduce のスケジューラが把握し，上記の処理を行っている最中に Mapper が失敗したら，データが失われないように MapTask をリトライするようにするという処理が必要となる．この処理については，今後の課題とする．

5. 評　価

提案手法の効果を確認するため，標準の Hadoop と Map Multi-Reduce を実装した Hadoop 間で比較を行った．

Record Reduce の評価実験では，ユーザプログラムで最適化を行った in-mapper combining と，前章で述べた複数の設計で実装した Record Reduce で性能比較を行い，フレームワーク化された Record Reduce が実用的なオーバヘッドで動作することを示す．

Local Reduce の評価実験では，in-mapper combining のプログラムを標準の Hadoop と Local Reduce が有効になっている Hadoop の上で動作させ，処理速度の差と ReduceTask の入力レコード数および Shuffle 量を測定することで，提案手法が有効であることを示す．

5.1 Record Reduce のフレームワーク化によるオーバヘッドの測定

Record Reduce の評価は，表 1 に示すマシン 1 台を用いて，Hadoop の疑似分散モードを動作させて行った．分散モードでなく，疑似分散モードで動作確認を行ったのは，in-mapper combining の処理の効果は疑似分散モードでも観測できるためである．

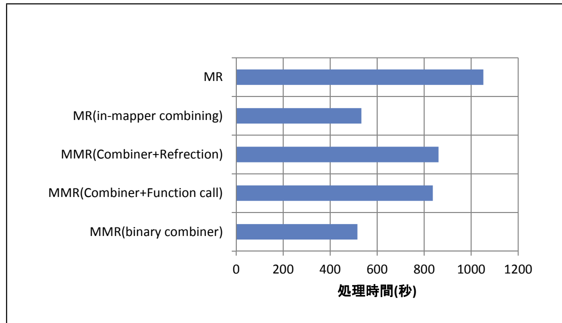

図 8 10GB のデータに対して，疑似分散モードで WordCount を行ったときの処理時間の結果．

本実験で用いた入力データは，Apache Hadoop に標準に含まれる RandomTextWriter を用いて，ランダム生成したテキストデータ 10GB である．

本実験で比較対象となる処理は以下の通りある：

(1) MR : WordCount のプログラム．Map 関数内で key として単語を，value として 1 を emit し，Reduce 関数内で key 毎に集約を行うことで，単語数を数え上げる．

(2) MR(in-mapper combining) : in-mapper combining を用いて最適化を行った WordCount．

(3) MMR(Combiner + Function) : ユーザが定義した Combiner を関数呼び出しとして呼び出す方法．

(4) MMR(Combiner + Reflection) : Combiner をリフレクション経由で呼び出す方法．

(5) MMR(binary combiner) : ユーザが定義した二値集約関数を関数呼び出しとして呼び出す方法．

以上を比較した実験結果を，図 8 に示す．MR と比較して，MR(in-mapper combining) の実行時間はほぼ 50%程度に抑えられている．MMR(Combiner+Reflection) と，MMR(Combiner + Function) は，通常の WordCount と比べ，約 25% 高速であるが，MR(in-mapper combining) の速度に及んでいない．また，MMR(Combiner+Reflection) と，MMR(Combiner + Function) は，3%程度しか実行速度に差がない．つまり，Refrection API のオーバヘッドはそれほど大きくない．一方で，MMR(binary combiner) は比較対象の中では最速で，MR(in-mapper combining) と同程度の速度が出ている．逐次集約処理を行うことができる binary combiner と in-mapper combining は，Combine 呼び出し時に必要なオブジェクト生成コストおよび繰り返し処理コストを省略できるため，呼び出しコストが他の手法に比べて小さい．よって，この

ⓒ 2012 Information Processing Society of Japan

表 2 Local Reduce の実験に利用した計算機構成

CPU	Core(TM)2 Duo CPU E7400 2.80GHz x 8
メモリ	8 GB
ネットワーク	1 GbE
OS	Linux カーネル 2.6.18
Apache Hadoop	6 月 17 日の trunk 版

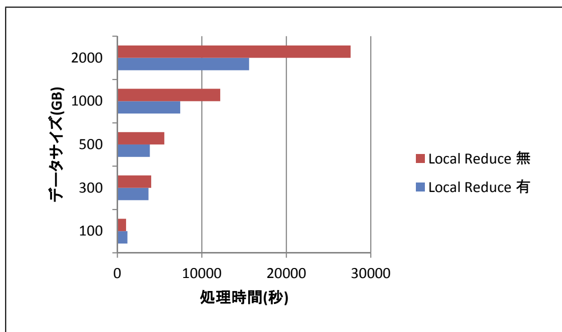

図 9 Local Reduce を有効/無効にしたときに，完全分散モード 40 台で，データサイズを変えながら in-mapper combining 版 WordCount を動作させたときの処理時間の比較.

ような結果になったのだと考えられる.

5.2 Local Reduce の効果

5.2.1 データサイズの増加に対する効果

Local Reduce の評価は，表 2 に示すマシン 40 台を用いて，Hadoop の分散モードを動作させて行った. 本実験で用いた入力データは，RandomTextWriter を用いてランダム生成したテキストデータ 100GB, 500GB，1TB，2TB であり，Local Reduce の Combine 間隔は，3 分とした．実行したプログラムは，前節で用いた in-mapper combining 版 WordCount である.

実験結果を，図 9 に示す．データサイズが 100GB のとき，Local Reduce がない場合に比べて遅くなってしまっている．これは，Local Reduce を行う間隔を 3 分と決め打ちにしているため，ある計算機に割り当てられた他の MapTask が終了した後も処理をしばらくブロックしてしまうことが原因である．この問題は，MapReduce のスケジューラを改造することにより対処が可能だが，今後の課題とする.

一方，データが 300GB 以上の場合は，徐々に性能が向上していることが分かる．2TB の WordCount では，ジョブ全体で約 1.7 倍の高速化ができるということを確認できた.

データの増加につれて若干性能が向上するのは，Local Reduce により Reducer の負荷の一部を Mapper に分散することによる効果が大きくなるためと考えられる．実験結果から，データ量が増加しても一定以

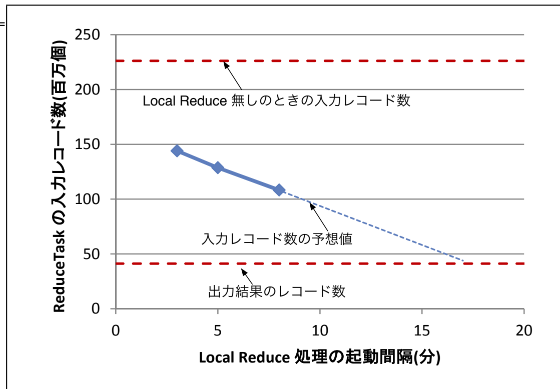

図 10 100GB のテキストに対して Local Reduce を実行する間隔を変更したときの ReduceTask の入力レコード数の推移.

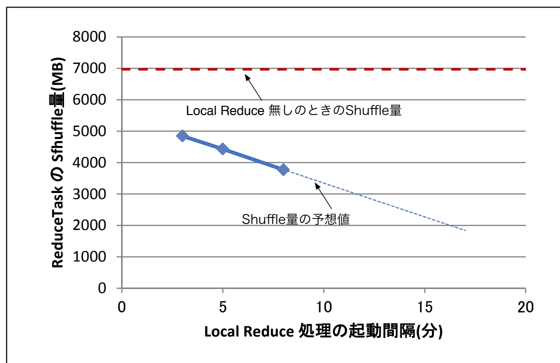

図 11 100GB のテキストに対して Local Reduce を実行する間隔を変更したときの Shuffle 量の推移.

上の性能向上が見られるため，Local Reduce は in-mapper combining や Combier に比べ，データ量増加に対するスケーラビリティが高いといえる.

5.2.2 集約間隔の変更による ReduceTask への入力レコード数および Shuffle 量の削減効果

前節の実験では，Local Reduce を実行する間隔を 3 分に固定して実験を行っていた．この間隔を長くすると，Local Reduce により集約できる MapTask の中間出力の数が増加し，ReduceTask の入力レコード数および Shuffle 量をより多く削減することができる．この様子を確かめるため，100GB のデータに対して，集約間隔を変更しながら ReduceTask の入力レコード数および Shuffle 処理量の変化を計測した結果と，実測値を元にシミュレーションを行った結果を図 10, 11 に示す．Local Reduce の実行間隔は，3 分，5 分，8 分と変化させた．8 分で測定が終わってしまっているのは (図の実線の部分)，利用しているバージョンの Hadoop のタイムアウトに引っかかってしまい，測定

ⓒ 2012 Information Processing Society of Japan

ができなかったためである．

Local Reduce の実行間隔を 3 分にしたとき，入力レコード数および Shuffle 量は Local Reduce なしのときと比較して，37%削減できている．そして，5 分，8 分と Local Reduce の実行間隔を大きくするほど，集約する MapTask の中間ファイルの量が大きくなり，ReduceTask の入力レコード数および Shuffle 処理の大きさは小さくなる傾向が観測され，8 分間隔のときに最大で 53%削減できている．

シミュレーションは，実測値をもとに線形近似を行った (図中の波線)．Local Reduce の実行間隔を大きくしていくと，ReduceTask の入力レコード数が，ReduceTask の出力レコード数の値に近づいていくが，WordCount の際の入力レコード数が出力レコード数以上に小さくなることはありえない．このため，ReduceTask の出力結果と交わったところ (17 分地点) で，シミュレーションを終了している．Shuffle 量についても同様の傾向がみられたが，17 分地点より先は入力レコード数の集約効果が変わることに起因して，Shuffle 量の変化の特性が変わることが予想される．よって，ReduceTask の入力レコード同様，17 分地点でシミュレーションを終了している．

シミュレーション結果によると，17 分地点で集約を行うことにより，ReduceTask の入力レコード数・Shuffle 量を最大で 75% ほどまで削減できる可能性がある．この結果は，WordCount 固有の結果であるので，別の処理についても成立する性質ではない．しかしながら，入力データが大きく，key の数が一定数であるような MapReduce 処理において，提案手法は Reducer の負荷を大きく軽減できる可能性があるといえる．ここで示した可能性の検証や，WordCount 以外での Local Reduce の効果検証は，今後の課題とする．

6. 関連研究

Dremel[8] は，カラムナストレージと分散 DB の集約技術を用いて，高速に SQL を実行できる技術である．提案手法は，MapReduce 上で集約処理を行っており，MapReduce の API を変更せずに実現可能な点が異なる．

Dryad[5] は，MapReduce よりも汎用的なプログラミングモデルを提供することで，より柔軟，高速に処理を行うための技術である．Dryad には，自動的に部分集約するような仕組みがあるが，ノード間で処理の受け渡しを行うことが前提となっており，Mapper 側で集約処理を行うことは考慮されていない．また，Dryad の場合は，処理のフローをデータフローとして記述する必要があるが，提案手法では MapReduce 以上の処理を記述する必要がない．

Li ら[6] は，reducer 側で逐次集約を行うことにより Hadoop の性能を高める方法を提案している．Li らは shuffle された複数の結果を reducer マージする処理のコストが高いことを指摘しており，この問題を reducer 側に hashmap を持たせて逐次集約することで解決する．これに対して，提案手法は mapper 側で集約する方法であるため，補完的な方法であるといえる．

7. まとめ

本稿では，集約処理を用いて MapReduce 処理を高速化する Map Multi-Reduce について述べた．今後は，より大規模なデータを用いて実験を行い，提案手法の有効性を検証する予定である．また，耐故障性を担保した Local Reduce の設計，実装を行い，その効果について検証する予定である．

参考文献

1) : Amazon Elastic MapReduce. `http://aws.amazon.com/jp/elasticmapreduce`.
2) : Apache Hadoop. `http://hadoop.apache.org`.
3) : Apache Hadoop Wiki. `http://wiki.apache.org/hadoop/PoweredBy`.
4) Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 10–10 (2004).
5) Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, New York, NY, USA, ACM, pp. 59–72 (2007).
6) Li, B., Mazur, E., Diao, Y., McGregor, A. and Shenoy, P.: A platform for scalable one-pass analytics using MapReduce, *SIGMOD '11: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, ACM, pp. 985–996 (2011).
7) Lin, J. and Dyer, C.: *Data-Intensive Text Processing with MapReduce*, Morgan and Claypool Publishers (2010).
8) Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M. and Vassilakis,

ⓒ 2012 Information Processing Society of Japan

T.: Dremel: interactive analysis of web-scale datasets, *Commun. ACM*, Vol. 54, No. 6, pp. 114–123 (2011).

9) Silberstein, A. E., Sears, R., Zhou, W. and Cooper, B. F.: A batch of PNUTS: experiences connecting cloud batch and serving systems, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, New York, NY, USA, ACM, pp. 1101–1112 (2011).

10) Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., SenSarma, J., Murthy, R. and Liu, H.: Data warehousing and analytics infrastructure at facebook, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, ACM, pp. 1013–1020 (2010).

ⓒ 2012 Information Processing Society of Japan