複数機械学習処理における MapReduce 最適化

福本　佳史[†]　　鬼塚　真[†]

Optimization for Multiple Machine Learning on MapReduce

Yoshifumi FUKUMOTO[†] and Makoto ONIZUKA[†]

あらまし　MapReduce は大規模データの分散バッチ処理を実現する実践的なフレームワークであり，その Java 実装である Hadoop は多くの企業において導入され実際に活用されている．機械学習アルゴリズムを Hadoop 上で動作させることによって，単純な統計解析よりも有益な知識を得る分析処理が注目されつつある．しかし MapReduce を利用した機械学習は，処理結果の最適化のためにアルゴリズムに事前に与えるハイパパラメータ値が異なる複数処理を要するので，コストが大きい．そこで我々は機械学習のハイパパラメータ調整に伴う複数処理を透過的に共有化し，総処理時間を削減するための新しいフレームワークを考案した．本フレームワークは (1) 複数の MapReduce ジョブから自動的に共有可能な部分を見つけ出すことで共有可能な範囲を最大化し，(2) 見つけた共有範囲から共有実行プランを生成し，それに従って MapReduce ジョブを実行することで総処理量を削減する，という二つの特徴をもつ．3 種類の機械学習アルゴリズムにおいて，我々のフレームワークが透過的に共有化を実現し，処理量削減の効果を確認した．

キーワード　MapReduce，マルチクエリ最適化，機械学習

1. まえがき

Web サービスなどを通して，企業がテラバイト級の大規模なデータを収集することは容易になりつつあり，大規模データ分析によって成功を収めている Google や Facebook の技術が注目されている．そのような大規模データ分析には分散処理フレームワーク MapReduce [1] が有効であり，それを Java 実装した Hadoop(注1) がよく活用されている．MapReduce による処理（ジョブ）は map と reduce の 2 フェーズに分かれ，map・reduce 関数を定義することで様々なアルゴリズムを分散処理化することができ，機械学習ライブラリ Mahout(注2) を利用することで機械学習アルゴリズムによる高度な大規模データ分析を容易に実践する環境が整いつつある．しかし，MapReduce を利用した機械学習処理はコストが大きくなる傾向がある．なぜなら，機械学習アルゴリズムは統計解析に比べ計算量が多く，また事前に与えるパラメータ値を調節することで処理結果を最適化しなければならないからである．例えば，K-Means [21] アルゴリズムは，反復的な MapReduce ジョブ（多くは 5〜20 回）が実行されるため計算量が大きい．更に生成されるクラスタの粒度を調節するために，パラメータ K の値が異なるパターンを複数回実行する必要があるため，総合的な計算量は更に大きくなる．Support Vector Machine(SVM) [18], Naive Bayes [20] などの代表的な機械学習アルゴリズムの多くも，処理結果の品質に影響するパラメータ値を要求する．そのようなパラメータはハイパパラメータと呼ばれており，最適化が非常に難しい．以上のように，大規模データを対象とする機械学習は，計算量が多く，更に複数回の試行を伴うため，多くの計算機資源を消費するという問題がある．また，Mahout を活用することで大規模データに対する機械学習自体は容易であるが，開発者やユーザの大規模環境に適応するための学習負荷が高く，既存のアルゴリズムに手を加えて最適化することはいまだに敷居が高い．以上の問題点を解決するため，Hadoop を用いた大規模機械学習処理のハイパパラメータ調節に伴う総処理量の削減を，極力透過的に

[†] 日本電信電話株式会社　NTT ソフトウェアイノベーションセンタ，武蔵野市
NTT Software Innovation Center, NTT Corporation, Musashino-shi, 180–8585 Japan

(注1)：http://wiki.apache.org/hadoop
(注2)：http://mahout.apache.org/

実現する手段が求められている．

我々はこの課題に対して，Mahout に含まれるアルゴリズムを MapReduce を用いた機械学習の典型例とし，Mahout における Hadoop API の利用・記述方法を想定した MapReduce ジョブに対し，透過的に適用な可能な最適化手法を提案することを目的とする．最適化にあたって，利用した機械学習ハイパパラメータ値調節のために実行される複数 MapReduce ジョブには同じアルゴリズム・同じ入力データを利用するため，重複処理（例えば入力データの読込部分や map 関数）があることに着目した．重複処理を含む複数の MapReduce ジョブの総合的な処理量を削減する既存研究は複数存在する．例えば HaLoop [10] や Twister [11]，Incoop [25] は，MapReduce ジョブの中間データ・処理結果をメモ化して再利用することによって，処理の高速化が実現されている．しかし Haloop と Twister の適用には独自の API に従ってアルゴリズムを記述して再利用する処理を定義する必要があり，Incoop は想定する MapReduce ジョブに適用可能であるが，パラメータ値が異なる場合のメモ化データ再利用可否判断について言及されていないため，誤った処理をする懸念がある．これら三つの既存技術はハイパパラメータ調節のケースに透過的に適用可能とは言いがたく，メモ化したデータのメンテナンスなど考慮すべき要素も増加してしまう．MRShare [3] は複数 MapReduce ジョブの重複処理共有化をメモ化を利用せずに実現するフレームワークである．MRShare は重複処理を含む複数 MapReduce ジョブが同時に実行される場合，自動的にそれらを一つのジョブにマージし，そのジョブの入力データの読込部分を 1 度だけ実行し，更に冗長な map 関数の出力（中間データ）も共有化した上で，以降の部分（reduce 関数など）についてはマージ前のジョブのものを個別に実行することで，複数の機械学習処理の総処理量削減を実現している．しかし，MRShare も共有する処理（特に map 関数）の自動判定が成されていないため，機械学習アルゴリズムの内容に関する知識のないユーザの場合は最大限に共有化することができない．具体的には，入力データ読込部分と中間データ以外の共有化の有無はユーザが静的に指定しなければならず，共有可能な処理を見逃してしまう，若しくは共有不可の処理を無理に共有化してしまう可能性がある．アルゴリズムのソースコードを参照し，ハイパパラメータが MapReduce ジョブのどの処理に影響するか調査することで共有可能な処理の指定が可能となるが，それを全てのユーザに強いるのは難しい．以上より，既存技術はハイパパラメータ最適化における総処理量を削減することが可能だが，特に削減量を最大化するための透過性の観点で不足しているといえる．

そのため，我々は Hadoop 上で動作する新しい共有化フレームワークを提案する．本フレームワークは Mahout のものを想定した機械学習アルゴリズムをブラックボックスとして，表層的に若しくは実行時に得られる情報のみを頼りに自動的な共有可能処理の判定を実現し，更に反復的な MapReduce ジョブを実行するタイプの機械学習アルゴリズムに対しても共有化を実現する．まずハイパパラメータが影響する部分を自動的に判定するため，事前にアルゴリズムを実行する過程の MapReduce ジョブを監視することでジョブ中にどのパラメータがどの処理部分で利用されたかの対応関係を検知・記録する．（この処理を監視ステップと呼び，監視ステップはユーザが意識的に主導で実施する必要がある）その後，ハイパパラメータ値調節のために複数の MapReduce ジョブが与えられた際，監視ステップで得られたパラメータと処理部分の対応関係を参照しながら，各ジョブに与えられたパラメータセットを比較し，利用するパラメータの値に差異がない，つまり共有可能な処理部分を検出する．そして，それを反映した共有実行プランを生成し，共有可能部分をもつ複数の MapReduce ジョブを一つのジョブにマージ・実行する．また，MapReduce ジョブの監視によってそのジョブが反復タイプの機械学習アルゴリズムの一部であることも検知・記録することが可能である．反復タイプのアルゴリズムにおいては，可能な場合は入力データ（それをもつ Java VM）をメモリキャッシュし，それ以降の反復に再利用することでより多くの処理量を削減する．

提案する共有化フレームワークを Hadoop の拡張によって実装し，Mahout に含まれる代表的な機械学習アルゴリズム (SVM，Naive Bayes，K-Means) の既存実装を用いた評価実験を既存実装に一切手を加えずに行って，共有化により処理時間が短縮されることを確認した．

2. 背　　景

2.1 機械学習アルゴリズムによるデータ分析

大規模データを対象とする機械学習は処理結果の性能（精度）改善の可能性が高くなる点で有益だが，処

理時間も長くなる．機械学習のコストを高める主要因として，まず一つ目は機械学習アルゴリズムの計算量が大きいことである．多くのアルゴリズムは平均・分散・頻度などの統計計算や，ベクトルの内積・距離計算などの演算の組合せから構成され，それらが反復されるタイプのアルゴリズムも存在するため，大規模データを対象とする場合に比較的に計算量が大きくなる傾向がある．

二つ目は，パラメータ値調節の必要性である．多くの機械学習アルゴリズムは処理結果の性能に影響するパラメータ値を事前に与える必要があり，そのようなパラメータはハイパパラメータと呼ばれる．適切ハイパパラメータ値は入力データや出力結果の用途などに依存するため，その都度の調整が必須である．

図 1 は SVM アルゴリズムのハイパパラメータ値を変化させながら，出力されたモデルの分類精度をプロットした予備実験の結果である．初期値は 0.01 であるのに対し，値が 1000 のとき最大の精度が記録されており，それより大きい値では精度が下がっている．このようにハイパパラメータ値の最適化は難しく，値を変えながらの試行錯誤が必要となる．

グリッドサーチ [14] はハイパパラメータ値調整に使われる手法で，複数種類のハイパパラメータ値を用いた処理を同時に試行し，処理結果の中で最も良い性能を示した値の周辺値を再度選択して試行を繰り返すことで最適化を進めるというものである．

このような，計算量が大きく，複数回の試行を伴う機械学習は，多くの計算機資源を消費してしまうという問題がある．

2.2 MapReduce

MapReduce は分散バッチ処理に適したフレームワークであり，MapReduce と分散ファイルシステム (DFS) の組合せにより大規模データの効率的な処理を実現している．MapReduce ジョブを実行するサーバクラスタは 1 台のマスタノード（マスタ）と複数台のスレーブノード（スレーブ）から構成され，DFS のデータノードも兼ねるスレーブノードには大規模データが分割格納される．MapReduce ジョブは map・reduce の 2 フェーズから構成される．前者は scan・map の 2 処理単位に分割でき，後者は shuffle・reduce の 2 処理単位に分割可能で，処理の流れを以下のように表すことができる．

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

map フェーズではマスタがスレーブに対して map タスクを割り当てる．map タスクでは，まず分割された入力データ D が scan され，key-value のペア $\{K_1, V_1\}$ が生成される．次に，その key-value ペアに対し，ユーザが定義した map 関数を，map 用パラメータ P_{map} を用いて実行し，新たな key-value ペア $\{K_2, V_2\}$ を出力する．reduce フェーズではマスタがスレーブに対して reduce タスクを割り当てる．reduce タスクでは，map フェーズの出力である key-value ペアを key に従って shuffle（グルーピング）し，同じ key をもつ全ての key-value ペアが一つのスレーブにネットワークを介して集約 $\{K_2, \{V_2\}\}$ する．そして収集されたグループに対しユーザが定義した reduce 関数を，reduce 用パラメータ P_{reduce} を用いて実行し，処理結果として新たな key-value ペア $\{K_3, V_3\}$ を出力する．

機械学習アルゴリズムは一つないし複数の MapReduce ジョブから構成されるため，処理単位の列を図 2 のような論理構造として表すことができ，我々はこれを処理列と呼んでいる．

2.2.1 MapReduce の実装

Hadoop は MapReduce の Java 実装である．Hadoop でのスレーブは，map・reduce タスク割当のたびに一つの Java VM を起動し，たいていはタスクが終了するとその VM も終了する．

全ての Java VM から同じパラメータセットを参照するため，同じコンフィグオブジェクトが与えられており，それには map 用・reduce 用・Hadoop のシス

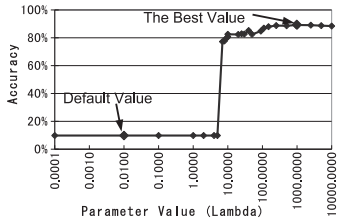

図 1 ハイパパラメータ値による影響
Fig. 1 Inpact of Hyperparameter Values.

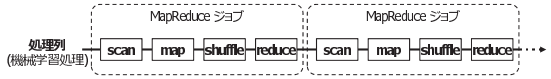

図 2 処 理 列
Fig. 2 Working Unit Sequence.

テム用・入力出力先など全てのパラメータが区別なくハッシュマップとして格納されていて，各パラメータ値は MapReduce ジョブ実行時にプログラム内でハードコーディングされたパラメータ名を用いて参照される．そのため，ソースコードを見ない限り，どのパラメータがどの処理単位で使われるかは MapReduce ジョブが実行されるまで不明である．

3. 共有化フレームワークの概説

大規模データを代表的な機械学習アルゴリズムを利用して分析する例として，SVM，Naive Bayes，K-Means を用い，我々のフレームワークがどのように自動的な共有化を実現するかを概説する．

3.1 SVM アルゴリズム

まず，SVM アルゴリズムがどのように我々のフレームワークによって共有化されるかを述べる．データ分析の例として，手書き数字画像の認識を用いる．0 から 9 の数字ラベルのどれかが付与された手書き数字画像ベクトルのセットを入力データとして用意し，それに SVM を適用すると分類モデルが出力される．この分類モデルを用いることで，数字ラベルのない手書き数字画像にどの数字ラベルが付与されるべきかを推定することができる．

MapReduce を利用する SVM として機械学習ライブラリ Mahout に含まれる実装(注3)を利用する．この実装は，MapReduce ジョブ 1 回で構成されており，map フェーズでは入力ベクトルのフィルタ・サンプリングを行い，reduce フェーズでは数字ラベルごとに集約されたベクトルセットから一対他の分類モデル（0vs.1-9 など）を出力する．そして，MapReduce ジョブの後に 10 個（0-9）の分類モデルを統合し，一つの分類モデルを出力する．

ユーザははじめに，実際にハイパパラメータ値の調節を行う前に，同じ処理列をサンプリングされた入力データを利用して実行し，その過程の MapReduce ジョブを我々のフレームワークに監視させる．それによって，ハイパパラメータが reduce 処理内で使われることを含む，全パラメータとそれぞれが参照される処理単位の対応関係が検知され，共有化のための情報として記録される．

次に，SVM アルゴリズムに事前に与えるハイパパラメータの値を適切に決定するために，グリッドサーチ法による最適化の過程で複数の SVM を同時に実行する．図 3 はその複数処理列の論理構造を示している．この例では，n パターンのハイパパラメータ値の試行に伴い，同時に n 個の MapReduce ジョブを開始し，n 個の分類モデルを出力して，最終的に最も良い精度を示したものを採用する．ハイパパラメータ値の異なる値を与えた複数処理列が与えられたとき，n 個の MapReduce ジョブが Hadoop のジョブキューに投入される．我々のフレームワークはそれらのジョブのもつコンフィグオブジェクトに格納されたパラメータセットを分析し，異なる値のパラメータが使われるのは reduce 処理のみであることを，先に得ていたパラメータと処理単位の対応関係を利用して検出して，図 4 のような論理構造の共有実行プランを生成する．そして，その共有実行プランに従って n 個の MapReduce ジョブを一つの MapReduce ジョブにマージした上で，scan・map・shuffle の処理単位を 1 度だけ実行し，n 個の reduce 処理をハイパパラメータ値ごとに個別に実行する（厳密には，ハイパパラメータ 1 パターン分の中間データが shuffle され，n パターンの reduce 処理をまとめて実行する reduce タスクが，中間データのグループ数だけ実行される）．

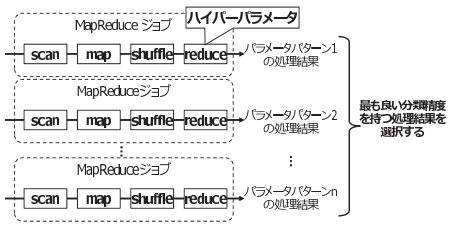

図 3 SVM の複数処理列
Fig. 3 Multiple Working Unit Sequence of SVM.

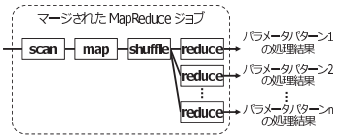

図 4 SVM の共有実行プラン
Fig. 4 Sharing Execution Plan of SVM.

このようにして，scan・map・shuffle 処理が共有可能であることが自動的に判定され，共有化によってディスク I/O・CPU・ネットワークコストが削減される．

3.2 Naive Bayes アルゴリズム

次に，Naive Bayes アルゴリズムの共有化について述べる．Naive Bayes は実践的な分類アルゴリズムの一つで，スパムフィルタ生成等に活用されるが，説明

(注3)：https://issues.apache.org/jira/browse/MAHOUT-232

を単純化するために手書き数字画像認識の例を用いる．Naive Bayes は，手書き数字画像ベクトルを入力すると，SVM と同様に数字ラベルのない手書き数字画像ベクトル付与されるべきラベルを推定する分類モデルを出力する．

Mahout の Naive Bayes は 3 回の MapReduce ジョブから構成され，1・2 回目の MapReduce ジョブで入力データを tf/idf データに変換し，3 回目でそれを集約して分類モデルを出力する．

Naive Bayes もハイパパラメータ値の調節を要するアルゴリズムであり，図 5 はそれに伴う n 個の処理列を示している．ユーザは SVM と同様に，Naive Bayes アルゴリズムの MapReduce ジョブ中に全てのパラメータがどの処理単位において参照されるかの対応を我々のフレームワークに事前に取得させた上で，複数の処理列を与える．我々のフレームワークは最初の n 個の MapReduce ジョブ間に差異が全くないことを自動的に検知し，図 6 のように，そのうちの一つだけを実行する．同様に 2 番目のジョブも一つだけを実行し，最後のジョブではハイパパラメータが使われることを検知して，そのパラメータパターンごとに個別に実行する．

このケースでは複数の MapReduce ジョブの全体を共有化するため，処理量が大きく削減される．

3.3 K-Means アルゴリズム

最後に K-Means アルゴリズムの共有化について述べる．K-Means は典型的なクラスタリングアルゴリズムで，類似するベクトルを集めたクラスタを生成する

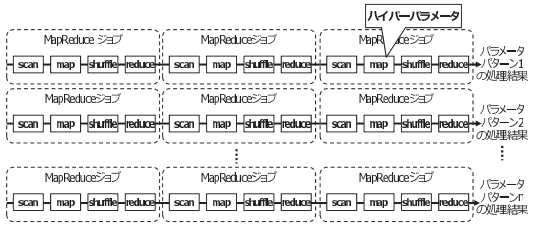

図 5 Naive Bayes の複数処理列
Fig. 5 Multiple Working Unit Sequence of Naive Bayes.

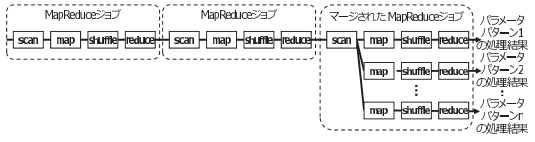

図 6 Naive Bayes の共有実行プラン
Fig. 6 Sharing Execution Plan of Naive Bayes.

ることができる．手書き数字画像ベクトルから 10 個のクラスタを生成した場合，同じクラスタには同じ数字ラベルが付与されたベクトルが所属する傾向が現れると考えられる．

K-Means は反復的な MapReduce ジョブで構成される．まず K 個の初期セントロイド（クラスタの中心点ベクトル）が与えられる．その後収束するまで反復される各 MapReduce ジョブでは，毎回同じ入力ベクトルセットを読み込み，それぞれを最も距離の近いセントロイドをもつクラスタに所属させ，その後各クラスタのセントロイドを所属する全ベクトルの重心位置に移動させる．全てのセントロイドの位置が一定距離以上移動しなくなった時点で反復を終了し，最終的に K 個のセントロイドをモデルとして出力する．

ユーザは出力されるクラスタの粒度を決定するために，適切なハイパパラメータ K の値を与える必要がある．図 7 はそのための n パターンのハイパパラメータに伴う処理列を示している．ハイパパラメータ K は直接 MapReduce ジョブ内では利用されないが，それぞれ異なるセントロイドが与えられるため，セントロイドの格納ディレクトリを示すパラメータ値が異なる．セントロイドの格納ディレクトリは map 関数内で参照されるため，scan 処理までを共有可能である．

ユーザははこれまで説明したアルゴリズムと同様に，事前に K-Means を実行し，MapReduce ジョブを監視させることで，セントロイド格納ディレクトリが map 関数で参照されることを含む，各パラメータと使用される処理単位の対応関係を我々のフレームワークに検知させる．我々のフレームワークは反復的に実行される MapReduce ジョブの scan 処理を毎回共有化する図 8 の共有実行プランを生成し，それを適用することで，入力データの読み込み I/O コスト削減に寄与する．

我々のフレームワークでは，K-Means アルゴリズムに対しては追加的な共有化を行う．K-Means は反復タイプのアルゴリズムであり，反復される MapReduce ジョブは毎回同じ入力ベクトルを読み込むため，2 回

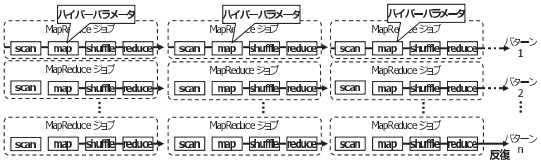

図 7 K-Means の複数処理列
Fig. 7 Multiple Working Unit Sequence of K-Means.

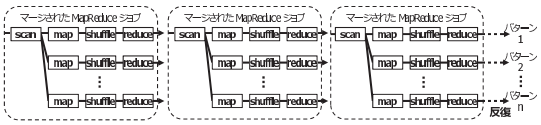

図 8　K-Means の共有実行プラン
Fig. 8　Sharing Execution Plan of K-Means.

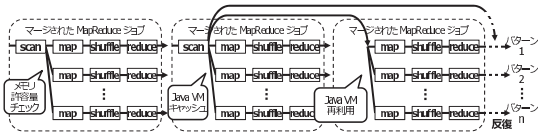

図 9　K-Means の共有実行プラン（全 scan 処理共有）
Fig. 9　Sharing Execution Plan of K-Means. (share all scans)

目以降の MapReduce ジョブの scan 処理及び，入力ベクトルの 2 乗和計算が冗長となっている．そこで，我々のフレームワークはパラメータと処理単位の対応関係のほかに，その MapReduce ジョブが反復タイプのアルゴリズムの一部かどうかも併せて事前に検知・記録しておく．そして，図 9 のように，反復の 1 回目において各スレーブノードの map 関数に入力入力ベクトルの総量がメモリに収まるかどうかをチェックする．入力ベクトルがメモリに収まる量であった場合は，反復の 2 回目において，map タスクのために起動されて入力ベクトルをメモリ内に保持したままの Java VM を，タスクが完了しても終了せずに保持するようにする．反復 3 回目以降は Java VM がキャッシュを再利用することで入力ベクトルの scan 処理をスキップする．以上のように，我々のフレームワークは入力データが十分に小さい場合は追加的に読み込み I/O コストの削減を実現する．

3.3.1　入力ベクトルの演算結果再利用

Mahout はベクトルを入力データとする際に，VectorWritable クラスを用いて入力データをデシリアライズする．VectorWritable は基本的なベクトル演算をメンバメソッドを用いて実行可能となっており，そのベクトルのみで完結する計算結果はメンバ変数としてキャッシュする機構をもっている．我々のフレームワークによって，VectorWritable をメモリ内に保持した Java VM が再利用された場合，副次的にキャッシュされたベクトル演算の結果も再利用されることになる．

4.　共有化アルゴリズム

我々の提案する共有化フレームワークについて説明する．我々のフレームワークは，機械学習アルゴリズムのハイパパラメータ調節に伴う複数の MapReduce ジョブから共有可能部分を自動的に検出することで，最大限の共有化を透過的に実現するフレームワークである．そのために，まず **2.2** にて言及した処理単位を共有化のための最小単位として扱い，共有可能な処理単位ペアを定義する（**4.1**）．それに従って処理単位の共有可否を検証するためにどのパラメータがどの処理単位で利用されるかの対応関係情報が必要となるが，我々のフレームワークではこの情報を，事前にハイパパラメータ値調節の対象となる機械学習を別途実行し，それを監視することで取得する（**4.2**）．そして，複数の処理列が与えられたとき，取得された情報をもとに各処理列に与えられたパラメータを比較し，共有可能な処理単位を自動的に判定して共有実行プランを生成する（**4.3**）．その実行プランに従って複数の MapReduce ジョブがマージし実行する．また，もしマージされた MapRedcue ジョブが反復タイプのアルゴリズムの一部である場合，入力ベクトルデータそのものをキャッシュして再利用する（**4.2**，**4.3.1**）．マージされたジョブは専用の mapper・reducer を用いて実行し，総処理量が削減される（**4.4**）．

4.1　共有可能な処理部分の定義

2.2 より，機械学習処理は処理列として表すことができ，我々のフレームワークはそれを構成する 4 種類の処理単位 (scan, map, shuffle, reduce) を共有化の最小単位として扱う．ある処理単位の共有可否は入力データ D，その処理単位内で実行されるユーザ定義関数 F，その処理単位内で使われるパラメータ P と直前の処理単位の共有可否により決定することが可能で，以下のように定義する．

定義 1 (共有可能な処理単位)　U_i，U_j を処理単位とする．U_i，U_j は，それぞれの D，F，そして P が同じであり，それぞれの直前の処理単位が存在しないか共有可能であるとき，共有可能である．

処理単位の共有化のために，MapReduce のジョブをマージする必要がある．マージ可能な MapReduce ジョブを以下のように定義する．

定義 2 (マージ可能な MapReduce ジョブ)　J_i，J_j を *MapReduce* ジョブとする．J_i，J_j はそれぞれの処理単位のうち少なくとも *scan* 処理が共有可能であるとき，マージ可能である．

ハイパパラメータ値の調節をするためには，**2.1** で述べたとおり，グリッドサーチ法等の過程でハイパパ

ラメータ値の異なる複数の処理を実行する．この複数処理は複数の処理列として表現が可能で，それらの処理列に与えられる入力データ D とユーザ定義関数 F は共通で，パラメータ P に差異がある．

4.2 監視ステップ

複数の処理列の間に存在する冗長な処理単位を自動的に共有化するために，ハイパパラメータが処理列内のどの処理単位で使われるかを検知する必要がある．また，反復タイプの機械学習アルゴリズムの共有化のために，処理列が反復タイプのアルゴリズムかどうかも検知する必要がある．

前者の課題は **2.2** で述べた Hadoop のデザインに起因する．Hadoop 上の MapReduce ジョブは，事前に与えられる全てのパラメータがそれぞれプログラムのどこで利用されるかの情報をもたないため，ハイパパラメータがどの処理単位で利用されるかを知ることが容易ではなく，冗長な処理単位の検知を阻害している．このような課題はプログラムから任意の変数が影響する範囲を切り出すプログラムスライシング [12] と，プログラムの部分的な実行を実現する部分評価 [13] によって解決可能であると考えられるが，原則としてソースコードが必要であり，我々は透過性を考慮してユーザがソースコードを用意することを想定しないため，これらの技術を利用することはできない．

この課題を効率的に解決するために，我々は Hadoop のパラメータを管理しているコンフィグオブジェクトを，拡張し，MapReduce ジョブ実行中に各処理単位からのパラメータ参照を検知・記録する機能を追加する．このオブジェクトはパラメータの参照を検知すると，Java のスタックトレース機能を用いて，参照されたパラメータ名と参照元の Java クラス名（Mapper・Reducer のクラス名）・メソッド名を記録する．このようなパラメータと処理単位の対応関係を得るために，我々のフレームワークでは，機械学習アルゴリズムのハイパパラメータ値調節を行う前に，ユーザがそのアルゴリズム（処理列）を手動で別途動作させることが必要となり，我々はこのステップを監視ステップと呼んでいる．監視ステップは情報収集が目的であるため，分析対象のデータ全体を用いる必要はなく，サンプリングされたサブセットで十分であるため，計算量を小さく抑えることが可能である．監視ステップによって，特に map と reduce 処理で使われるパラメータ (P_{map}, P_{reduce}) が判明する．

反復タイプの検知も監視ステップによって解決可能である．Hadoop のコンフィグオブジェクトは入力データのディレクトリと出力先のディレクトリ情報も管理しており，反復タイプの処理列ではたいていの場合，入力データに同じディレクトリが（更に出力先は反復回数が接尾辞として付与されたディレクトリが）利用されるため，それを検知して，そのジョブの Mapper クラス名・Reducer クラス名を反復タイプのアルゴリズムで利用されるクラスとして記録しておくことで，以降その Mapper や Reducer が含まれる MapReduce ジョブを反復タイプとして扱うことができる．

(a) 監視ステップの制約

本フレームワークは，監視ステップにおいてハイパパラメータが必ず 1 度は利用されなければ，それを検知することができないが，この制約はそれほど厳しいものではない．何故ならば，ほとんどのハイパパラメータは変数に格納するために，mapper か reducer の初期化関数 (setup) 内で少なくとも 1 度はアクセスされるからである．

ハイパパラメータごとにパラメータ名・値の形式でコンフィグオブジェクトに格納されていることも制約の一つである．機械学習アルゴリズムの設計によってハイパパラメータが，シリアライズされたハッシュマップの形式でコンフィグオブジェクトに格納された場合や，値が記述されたファイルパスが格納された場合などでは，我々のフレームワークは監視ステップでハイパパラメータと処理単位の対応関係を正常に検知できない．

(b) パラメータ・処理単位対応表のメンテナンス

監視ステップによって得られたパラメータとそれを参照する処理単位の対応表は，小規模なメタデータであるため例えば HDFS に恒久的に保持しても問題はない．機械学習アルゴリズムに変更がある場合は対応表をメンテナンスすることが必要であるが，例えば MapReduce ジョブの jar ファイルのハッシュ値を記憶しておいて，相違がある場合は改めて対応表を作成し直すことでメンテナンス自動化が実現可能であると考えられる．

4.3 共有実行プランの生成

グリッドサーチ法などによるハイパパラメータ最適化の過程で，パラメータのみ異なる複数処理列が開始されたとき，Hadoop のジョブキューには複数の MapReduce ジョブが投入される．我々のフレームワークはそれらのジョブをいったん別のキューに横取りしてジョブの開始を指定時間遅延させ，その中のマージ

可能なジョブをマージする．マージされたジョブは，遅延時間内であれば新たに投入されたジョブも追加的にマージされ，遅延時間が経過すると共有実行プランが生成され，通常の Hadoop ジョブキューに戻されて実行される．この共有実行プランは MapReduce ジョブをノードとするトリー構造で表現できる．各ノードには，処理単位をノードとするサブトリーが格納される．まず，複数の処理列から MapReduce ジョブレベルの共有実行プラン生成にフォーカスして説明し，その後 MapReduce ジョブ内の処理単位レベルでのサブトリー生成方法について述べる．

Algorithm 1 共有実行プランの生成

Input: $\{S_1, S_2, ..., S_y, ..., S_n\}$
　//処理列: $S_y = \{J_{y.1}, J_{y.2}, ..., J_{y.x}, ..., J_{y.m}\}$
Output: root
1: currentNode = new Node()
2: root = currentNode
3: mergeQueue = new Queue()
　//x: 処理列の x 番目の MR ジョブ, y: 処理列 ID
4: **for** $x = 1$ to m **do**
5:　**for** $y = 1$ to n **do**
6:　currentNode = root.getLeafNodeBy(y) //x が 1 なら root.
7:　mergeQueue.enqueue({currentNode, $J_{y.x}$})
8:　**while** ! mergeQueue.isEmpty() **do**
9:　{parentNode,J_i} = mergeQueue.dequeue()
10:　J = null
11:　**for all** childNode ∈ parentNode.getChildren() **do**
12:　J_j = childNode.getJob()
13:　J = merge(J_i, J_j) //Algorithm 2
14:　**if** $J \neq$ null **then**
15:　childNode.setJob(J)
16:　break
17:　**if** J = null **then**
18:　parentNode.addChild(new Node(J_i))
19:　root.launchMRJobsOfLeaves()

Algorithm 1 に共有実行プランの生成アルゴリズムを示す．共有実行プランはトリー状のデータ構造として表現され，幅優先的に構築される．ハイパパラメータ値調節のために n 個のパラメータパターンによる n 個の処理列 $\{S_1, S_2, ..., S_y, ..., S_n\}$ が与えられる．各処理列は m 個の MapReduce ジョブ $\{J_{y.1}, J_{y.2}, ..., J_{y.x}, ..., J_{y.m}\}$ から構成される各処理列先頭のジョブ $x = 1$ がジョブキューに投入されるとき，我々のフレームワークはそれを横取りし，currentNode とそのジョブのペア {currentNode, $J_{y.1}$} を mergeQueue に格納する．(4〜7 行目) currentNode は getLeafNodeBy(y) 関数によってそれぞれの処理列の直前の MapReduce ジョブ ($J_{y,x-1}$) かそれがマージされたジョブが格納されているノードであり，$x = 1$ の場合は root ノードとなる．mergeQueue に格納されたそれぞれのノードとジョブのペア {parentNode, J_i} において，J_i は parentNode の子ノードに格納された全てのジョブと比較され，マージ可能なジョブ J_j が格納されていた場合はそのジョブとマージされる．(8〜13 行目) ジョブのマージに関しては後述する．もし，parentNode が子ノードをもたない場合やマージ可能なジョブがない場合は J_i を格納する新たなノードが生成され，parentNode の子ノードとなる．(14〜16 行目) このジョブのマージを行う一連の処理を mergeQueue が空になるまで繰り返し，空になった時点で共有実行プランの全ての葉ノードに格納されている未実行 MapReduce ジョブを実行し，$x = 2$ へ進む．以上を処理列の末端まで繰り返し実行し，共有実行プランを構築しながら複数処理列が実行される．

Algorithm 2 MapReduce ジョブのマージ

Input: $J_i, J_j, D_i, D_j, P_i, P_j, \{P_{map}, P_{reduce}\}$
1: J = new MergedJob()
2: **if** $D_i \neq D_j$ **then**
3:　**return** null //このジョブペアはマージできない
4: J.addNode($scan_i$)
5: $P^{\triangle}_{map} = (P_i \triangle P_j) \cap P_{map}$
6: **if** $P^{\triangle}_{map} \neq$ null **then**
7:　J.addBranchs($\{map_i, shuffle_i, reduce_i\}$, $\{map_j, shuffle_j, reduce_j\}$)
8:　**return** J //scan 処理まで共有可能
9: J.addNode(shareMapUnits(map_i, map_j))
10: J.addNode(shareShuffleUnits($shuffle_i, shuffle_j$)
11: $P^{\triangle}_{reduce} = (P_i \triangle P_j) \cap P_{reduce}$
12: **if** $P^{\triangle}_{reduce} \neq$ null **then**
13:　J.addBranchs($reduce_i, reduce_j$)
14:　**return** J //map まで可なら，shuffle も可
15: $J = J_i$
16: **return** J //このジョブペアは全体を共有可能

Algorithm 2 は異なる処理列の MapReduce ジョブ同士をマージし，処理単位をノードとしたサブトリーを生成するアルゴリズムを示している．二つの MapReduce ジョブ $J_i(J_j)$ が与えられたとき，$D_i(D_j)$ をそれぞれの入力データ，$P_i(P_j)$ をそれぞれのパラメータセット（コンフィグオブジェクト）とする．そして，P_{map}，P_{reduce} を，パラメータセットのうち監視ステップで検知・記録された map・reduce 処理内で使われるものとする．MapReduce ジョブは，scan，map，shuffle，そして reduce の四つの処理単位から構成されており，二つのジョブの先頭から順に共有可能かどうかを判定していく．そして，共有できない処理単位がある場合，判定をその時点で終了し，それ以降の処理単位は全て共有不可とする．最初の判定では D_i と D_j が同じであるかどうかを確認する．もし同じでない場合は，対象となっている二つのジョブはマージ不可能と判断する．同じである場合は，scan 処理を共有可能と判断し，次の処理単位（map 処理）の共有可否判定に進む．(2〜4 行目) 次に，二つのジョブ

のパラメータセットの対称差 $P_i \triangle P_j$ となっているパラメータと map 処理で利用されるパラメータとの積集合を計算し，もしそれが空集合でないならば二つのジョブの map 処理で利用されるパラメータ値は異なると判断できるため，map 処理からは共有化せず分岐するサブトリーとする．そうでない場合，二つのジョブの map 処理は共有可能と判断する．(5〜9 行目）もし map 処理が共有可能であるならば，map 処理から出力されるデータは同じであり，次の処理単位である shuffle 処理はその結果をグルーピングするだけの処理であるため，必然的に shuffle 処理も共有可能と判断できる．(10 行目）しかしながら，あえて shuffle 処理を共有しない方がよいケースがあり，それついては後述する．次に map 処理と同様の方法で reduce 処理に関しても共有可否判定を行う．(11〜14 行目）もし reduce 処理までもが共有可能であるなら，二つのジョブは全体が共有され，どちらか片方のジョブのみが実行される（15〜16 行目）．

このようにして，監視ステップで得られたハイパパラメータが利用される処理単位の情報を元に，MapReduce ジョブ同士の共有可否判定を自動的に実行して共有実行プランが生成され，それをもとに MapReduce ジョブをマージして実行する．

(c) 共有実行プラン生成アルゴリズムの制約

本アルゴリズムは，scan・shuffle 処理内でハイパパラメータが使われないことを前提としている．実際に，Mahout に含まれるほとんどの機械学習アルゴリズムではそれらの処理単位でハイパパラメータが使われることはない．仮に，パラメータを利用するように拡張されたユーザ定義クラスが scan・shuffle 処理内で使われたとしても，map・reduce 処理と同様に利用されるパラメータを比較するよう修正することは容易である．

本アルゴリズムでは，combiner は動作する回数を制御することが難しくハイパパラメータが利用されることもほとんどないと考えられるため，shuffle 処理の一部として扱っている．仮に combiner 内でパラメータを利用するアルゴリズムがある場合は，五つ目の処理単位として，map・reduce 処理と同様に扱う必要がある．

4.3.1 反復タイプに特化した共有化

我々のフレームワークは，マージされた MapReduce ジョブが反復タイプのアルゴリズムの一部であった場合に追加的に共有化を行う．反復 1 回目のマージされたジョブ J の map 処理において，入力データが十分に小さくスレーブノードごとにメモリに収まる量であるかを判定し，収まる量である場合は 2 回目のマージされたジョブおいて map タスクのために起動された Java VM を終了せずに保持させることで，メモリ内の入力データを維持する．そして反復 3 回目以降に Java VM が保持されている場合は，それを再利用して入力データの scan 処理をスキップさせることが可能である．副次的な効果として，Mahout でベクトルを入力データとして扱う場合はベクトルの 2 乗和やノルム演算結果のキャッシュを再利用することができる．Mahout はベクトルを入力データとする場合に VectorWritable クラスを用いてデシリアライズを行うが，VectorWritable は基本的なベクトル演算を行うためのメンバメソッドをもっており，そのベクトルに閉じて計算可能なベクトル演算メソッドの処理結果をメンバ変数にキャッシュする機能を有している．デシリアライズ済みの入力データを維持している Java VM が再利用された場合，これらのベクトル演算結果のキャッシュも再利用することが可能である．

4.4 共有化されたジョブの実行パターン

共有実行プランをどのように実行するかについて述べる．共有実行プランにはマージされた MapReduce ジョブの実行が含まれており，我々のフレームワークは MRShare と同様に mapper・reducer クラスを専用の拡張したものに置き換え，ユーザが与えた mapper や reducer をその中で呼び出す方法で実行する．共有化した処理単位は 1 度だけ実行され，それ以降の処理単位はパラメータパターンごとに個別に実行される．共有化のパターンは，(a) scan 処理まで，(b) map 処理まで，(c) shuffle 処理まで，(d) reduce 処理まで（MapReduce ジョブ全体）の 4 パターンである．

(a) scan 処理までの共有化

二つのジョブ J_i, J_j がマージされたジョブを想定する．このジョブが scan 処理まで共有可能である場合の共有実行プランは以下のように表すことができる．

$$scan(D) \rightarrow \{K_1, V_1\}$$

$$\frac{map(K_1,V_1,P^i_{map}) \rightarrow \{tag_i(K^i_2), V^i_2\}}{map(K_1,V_1,P^j_{map}) \rightarrow \{tag_j(K^j_2), V^j_2\}}$$

$$\frac{shuffle(\{tag_i(K^i_2), V^i_2\}) \rightarrow \{tag_i(K^i_2), \{V^i_2\}\}}{shuffle(\{tag_j(K^j_2), V^j_2\}) \rightarrow \{tag_j(K^j_2), \{V^j_2\}\}}$$

$$\frac{reduce_i(tag_i(K^i_2), \{V^i_2\}, P^i_{reduce}) \rightarrow \{K^i_3, V^i_3\}}{reduce_j(tag_j(K^j_2), \{V^j_2\}, P^j_{reduce}) \rightarrow \{K^j_3, V^j_3\}}$$

2.2 との違いは二つのジョブが実行されており最初の scan 処理のみが 1 度だけ実行されている点と，map 処理の結果として出力される中間データにパラメータ

パターンごとに区別するためのタグが付与される点である．MRShare [3] の scan 共有化と同様なので説明を割愛する．

(b) map 処理までの共有化

この共有パターンは scan 処理と map 処理の両方を共有化し，shuffle 処理と reduce 処理は個別に実行するパターンであり，以下のように表すことができる．

$$scan(D) \rightarrow \{K_1, V_1\}$$
$$map(K_1, V_1, P^i_{map}) \rightarrow \frac{\{tag_i(K^i_2), V^i_2\}}{\{tag_j(K^j_2), V^j_2\}}$$
$$\frac{shuffle(\{tag_i(K^i_2), V^i_2\}) \rightarrow \{tag_i(K^i_2), \{V^i_2\}\}}{shuffle(\{tag_j(K^j_2), V^j_2\}) \rightarrow \{tag_j(K^j_2), \{V^j_2\}\}}$$
$$\frac{reduce(tag_i(K^i_2), \{V^i_2\}, P^i_{reduce}) \rightarrow \{K^i_3, V^i_3\}}{reduce(tag_j(K^j_2), \{V^j_2\}, P^j_{reduce}) \rightarrow \{K^j_3, V^j_3\}}$$

このパターンでは map 処理は共有化されるが，あえて map 処理の出力は共有化しないということになる．そのため，map 処理に係る CPU コストは削減されるが，出力のための I/O コストや shuffle のためのネットワークコストは削減されない．

(c) shuffle 処理まで共有化

この共有パターンでは，scan，map，shuffle 処理を共有し，reduce 処理のみパラメータパターンごとに個別に実行される．具体的には一つの reduce タスク内で複数パターン分の reduce 処理が実行され，パラメータごとに処理結果が出力される．本パターンは以下のように表すことができる．

$$scan(D) \rightarrow \{K_1, V_1\}$$
$$map(K_1, V_1, P^i_{map}) \rightarrow \{tag_{ij}(K_2), V_2\}$$
$$shuffle(\{tag_{ij}(K_2), V_2\}) \rightarrow \{tag_{ij}(K_2), \{V_2\}\}$$
$$reduce(tag_{ij}(K_2), \{V_2\}, P^{ij}_{reduce}) \rightarrow \frac{\{K^i_3, V^i_3\}}{\{K^j_3, V^j_3\}}$$

このパターンでは中間データに，複数のタグを付与することで中間データの shuffle 処理を共有化するため，map 処理の出力に係る I/O コストや shuffle 処理に係るネットワークコストを削減することが可能である．

(d) reduce 処理まで共有化

この共有パターンでは全ての処理単位を共有して実行するパターンであるため，マージされたジョブのうちどれか一つのみを実行し，出力結果のみを必要に応じてコピーする．ジョブを丸ごと共有化するため，最も共有化による各種処理コスト削減の効果が大きくなる．

(e) 共有化ジョブのタスクスケジューリング

共有化のためにマージされたジョブのタスクスケジューリングには特に手を加えておらず，通常の MapReduce ジョブと同様に扱われるが，マージされた影響によって処理の粒度が大きくなることの弊害が発生する．例えば，scan が共有化され map 以降は非共有のジョブを実行する際に，もしクラスタの map タスクの許容量（map スロット数）がタスク数よりも多い場合でも，粒度の大きい map タスクが map スロットの一部のみを使用して実行されることになる．また，中間データのキーのグループ数 $|K_2|$ が reduce スロット数よりも小さい場合，shuffle 処理までを共有化すなわち中間データを共有化してしまうと，reduce フェーズにおいて動作しないスレーブノードが発生する可能性があり非効率である．この問題を解決するには，同じ中間データをパラメータパターンの数だけ複製し，パラメータパターンのタグを付与することであえてキーのグループ数を増やして，異なる reduce スロットにおいて reduce 処理が行われるようにすることで負荷を分散するべきである．

本フレームワークは，以上のような共有化による処理粒度増大による並列度の低下を抑制するため，自動的に平均化することが将来的な課題となっている．

5. 評価実験

我々は Hadoop のバリエーションの一つである CDH3 (Cloudera's Distribution for Hadoop)(注4) を拡張する方法で提案するフレームワークを実装した．全ての評価実験は 10 台ないし 20 台の物理サーバマシンを利用して行った．各サーバのスペックは次のとおりである，Intel Core2Duo 1.86 GHz CPU，8 GByte RAM，7200 rpm 1 TByte HDD，1 Gbit/s NIC．実験に利用する機械学習アルゴリズムは，Hadoop 上で動作する機械学習ライブラリ Mahout に含まれる，SVM(注5)，Naive Bayes(注6)，K-Means(注7) を用いた．Hadoop のチューニングパラメータは HadoopDB [22] のものを参考に，io.sort.mb (256 MByte), io.sort.factor (100), mapred.child.java.opts(-Xmx2048 m), dfs.block.size (256 MByte), and dfs.replication (1) を除いて全て初期値を利用している．

機械学習の対象となる入力データとして，LIBSVM(注8) のサンプルデータセットとして用意されてい

(注4)：https://ccp.cloudera.com/display/SUPPORT/Overview
(注5)：https://issues.apache.org/jira/browse/MAHOUT-232
(注6)：https://cwiki.apache.org/MAHOUT/naivebayes.html
(注7)：https://cwiki.apache.org/MAHOUT/K-Means-clustering.html

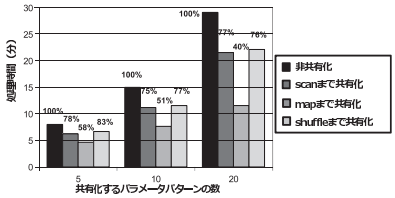

図 10　各処理単位の共有化効果（サーバ 20 台）
Fig. 10　Effectiveness of Sharing Working Units. (20 nodes)

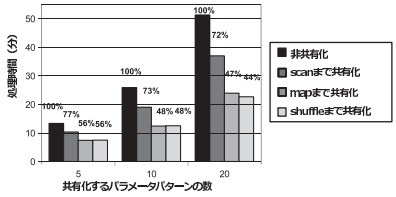

図 11　各処理単位の共有化効果（サーバ 10 台）
Fig. 11　Effectiveness of Sharing Working Units. (10 nodes)

るデータの中で最もサイズの大きいものを利用した．このデータはベクトル化された手書き数字 800 万個のデータ(注9)で，ベクトルの次元数は 784，平均の濃度（非ゼロ説明変数個数）は 199 で，ファイルサイズは 16 GByte である．

5.1　処理単位の共有化効果

MapReduce ジョブ内の処理単位を共有化することによる効果を SVM アルゴリズムを利用して実験を行うことで確認した．Mahout の SVM は Pegasos [19] をベースとしたアルゴリズムであり，ハイパパラメータ値が処理結果に及ぼす影響が特に大きいアルゴリズムである．この SVM アルゴリズムは MapReduce ジョブ 1 回で構成されており，ハイパパラメータは reduce 処理において利用されているため，scan，map，shuffle 処理は共有可能である．ハイパパラメータを 5，10，20 パターン用意し，10 台ないし 20 台のサーバクラスタ上で同時に処理を開始した場合の総処理時間を測定した．

図 10 は共有する処理単位を変えながらサーバ 20 台のクラスタ上で測定したものである．マージされる MapReduce ジョブの数が多いほど，共有部分の処理時間が占める割合が小さくなるため改善率が向上している．しかし shuffle 処理までを共有化したパターンは，ネットワークコスト (shuffle size) がそれぞれ 5 分の 1，10 分の 1，20 分の 1 になっているのにもかかわらず，map 処理までを共有化したパターンよりも処理時間が長くなった．これは Mahout の SVM アルゴリズムの実装では，手書き数字認識のケースにおいて目的変数がとり得る値が 0〜9 の 10 種類であるために中間データのグループ数が 10 個となり，shuffle 処理までを共有（中間データを共有）してしまうとマージされたジョブにおける中間データのグループ数も 10 個となってしまったため，20 台中 10 台のみに reduce タスクが割り当てられ，処理の並列度が不足したことが原因である．map 処理までで共有化を止めた場合は，中間データのグループ数は 10 個×パラメータパターン数となり，20 台全ての reduce スロットが活用されるため，並列度が適切となっている．つまり，中間データ共有による NW コスト削減よりも reduce 処理の並列度向上の方が処理時間短縮に寄与していると考えられる．

図 11 はサーバを 10 台に減らして測定した場合のものである．この場合は shuffle 処理まで共有化したときアイドル状態となるサーバが発生しなくなるため，map 処理よりも処理時間が短縮されることが確認できた．このことから，中間データのグループ数と reduce スロット数に応じて，map 処理まで若しくは shuffle 処理まで共有化のどちらか適する方を選択すべきであることが判明した．自動的にどちらかを選択させることは今後の課題である．

5.2　ジョブの共有化効果

MapReduce ジョブを丸ごと共有化した場合の改善効果を，Mahout の Naive Bayes アルゴリズムを用いて確認した．このアルゴリズムは 3 回の MapReduce ジョブから構成されており，ハイパパラメータは 3 回目のジョブにおいて使われるため，1・2 回目のジョブは完全に共有可能である．ハイパパラメータを 10 パターン用意し，20 台のサーバクラスタ上で同時に処理を開始した場合の処理時間を測定した．

図 12 は MapReduce ジョブを丸ごと共有化した場合の効果を示している．図の中の帯はそれぞれ一つの MapReduce ジョブの処理時間を表している．上側には 10 パラメータパターン×ジョブ 3 回の合計 30 個の帯がある．共有化しない場合の 2 回目・3 回目のジョブに関しては，それぞれの前のジョブが終了次第 Hadoop のジョブキューに投入され，リソースが空き次第 FIFO スケジューリングに従って処理が開始されるが，1 回目のジョブよりも入力データが小さくなる

(注8)：http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/multiclass.html
(注9)：http://yann.lecun.com/exdb/mnist/

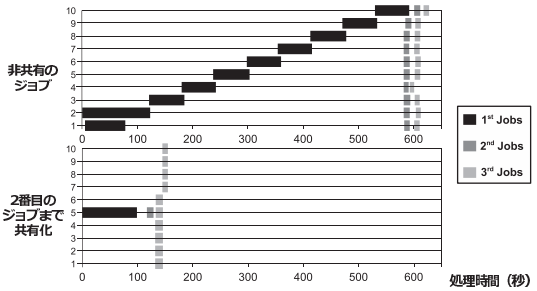

図 12　MapReduce ジョブの共有化効果
Fig. 12　Effectiveness of Sharing MapReduce Jobs.

ため，各ジョブが消費する map スロット・reduce スロットが少なくなり，スレーブ 20 台でほぼ並列して処理が行われている．全処理時間のうちの大半を最初のジョブが占めており，2 回目のジョブまでで 90 %以上を占めることが分かる．この 1・2 回目のジョブを共有化し，3 回目のジョブのみハイパパラメータごとに個別に実行したものが下側に示されている．結果として総処理時間は共有化によって 4 倍高速化されたことを確認できた．このように処理列（機械学習アルゴリズム）が複数の MapReduce ジョブからなり，ハイパパラメータが処理列の末尾に近い場所で利用されるパターンにおいて，本提案手法は特に効果的である．

5.3　反復タイプアルゴリズムの共有化効果

反復タイプのアルゴリズムを共有化することによる効果を Mahout の K-Means アルゴリズムを利用して実験を行うことで確認した．Mahout の K-Means は反復的な MapReduce ジョブで構成されており，ハイパパラメータはそれぞれのジョブの map 処理において利用されているため，それぞれの scan 処理は共有可能である．ハイパパラメータを 5，10，20 パターン用意し，20 台のサーバクラスタ上で同時に処理を開始した場合の総処理時間を測定した．K-Means の反復回数は 3 回までに制限している．

図 13 は反復される MapReduce の scan 処理のみを共有化した場合の改善効果を示している．scan 処理の共有化により約 15 %の高速化が確認できた．

図 14 は入力ベクトルデータ再利用による効果を評価するため，入力データを半分に削減し，シミュレーションとして，最初の MapReduce ジョブにおいて利用された map タスク用の Java VM を以降の反復で再利用するようにして測定したものである．入力データのキャッシュによる処理時間の改善効果は約 6 %となった．これは反復 2 回目以降のマージされた MapReduce ジョブの scan 処理を省略したことと，

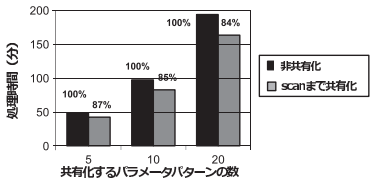

図 13　K-Means の scan 処理の共有化効果
Fig. 13　Effectiveness of Sharing Iterative Jobs.

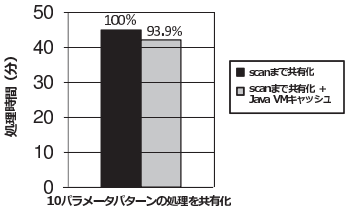

図 14　Java VM のキャッシュによる効果
Fig. 14　Effectiveness of Caches.

Mahout の VectorWritable としてデシリアライズされた入力データにおいて，ベクトルの 2 乗和演算結果キャッシュを再利用したことが寄与していると考えられる．それほど大きい効果ではないが，2 回目以降の MapReduce ジョブでは入力データ読込み時の I/O コストやデシリアライズに係る CPU コストがなくなっており，メモリの容量が許すのならばトレードオフとなる要素もないものと考えられる．

6.　関 連 研 究

MapReduce [1] は大規模なデータを処理するための最も一般的なフレームワークの一つである．我々の手法はデータベース分野における複数クエリ最適化 [2] を MapReduce に適用したものであり，同様の研究が存在する．

我々の目的は大規模データ分析の高速化であり，その中でも MapReduce を利用した機械学習処理を効率的に実行できるようにすることは重要である．MapReduce 上での機械学習に関する研究は多数行われており，アプリケーション側とエンジン側のアプローチに大別できる．

6.1　MapReduce フレームワーク上での共有化

先に述べたとおり，MRShare [3] は MapReduce ジョブの自動的な共有化を実現している．共有化前後の I/O 及びネットワークコストを算出するコストモデルを確率し，それに基づいてジョブをグルーピングする点が特徴であるが，共有する処理部分をユーザ静的に決定する必要があるため様々なアルゴリズムに柔軟に対応することができない．また，特に機械学習アルゴ

リズムでは中間データが相互比較を前提としていない専用のデータ構造（例：名前付きベクトル，クラスタ）として出力される場合があり，冗長な中間データの共有化機能が有効でない可能性がある．

複数分析処理の中にある冗長な処理部分を共有化するためには，一つの処理を複数のパーツに分解し，マージや並べ換えを行った上でそれぞれの実行を制御する必要がある．例えばSQLクエリのような宣言的な言語や独自APIに従って処理が記述されている場合は，クエリそのものを共有化のためのヒントとして利用できるため，分解・再構成による共有化が比較的容易である．Comet [5] はDryad [9] での複数クエリ共有化を実現している．この研究では，実データに対する定期的なクエリパターンの中には冗長な部位が多く存在することを明らかにし，scan・shuffle処理を共有化することで総処理量の削減が実現されている．YSmart [4] はHadoop上でSQLライクなクエリ言語Hive QLを提供するHive [6] 用のクエリオプティマイザであり，MapReduceジョブのscan・shuffleの共有化を含んでいる．Cheetah [8] も同様にSQLライクな記述を利用しての複数クエリ最適化を実現している．Pig [7] はMapReduceジョブを記述するための抽象的な言語Pig Latinを提供している．ユーザはMapReduceを使ったプログラムをより簡便に記述することが可能で，スクリプトを解析することによるscan，shuffleの共有化も提案されている．HaLoop [10] とTwister [11] は反復タイプのアルゴリズムに特化した最適化を行っている．例えばPageRankやK-Meansにおいて，中間データ・出力結果をメモ化し，インデックスを生成してアクセスを高速化することが提案されており，我々のフレームワークが対象としているハイパパラメータ調節に伴う複数処理の共通部分のメモ化をすることが可能と考えられる．これらの既存技術はいずれも宣言的言語や独自APIを前提として共有化を実現するものであり，本研究が対象とする通常のHadoopAPIのみに従って記述された様々な機械学習アルゴリズムにおいて，パラメータ値の異なる複数処理の共有化に適用する場合，開発者やユーザに作業的・知識的な負担を要するため，透過性に欠けていると考えられる．

6.2 アルゴリズム改善による効率化

MapReduceフレームワーク上の機械学習処理最適化は多くの研究において取り組まれている．我々は実験のためにMahoutを利用したが，MahoutはMapReduceに様々な種類の機械学習アルゴリズムを適用する研究 [17] をもとにしている．この研究では機械学習アルゴリズムをMapReduce上で動作可能にするために，*summation form* という形式に変換することを提案している．PLANET [16] はMapReduceを利用して決定木を生成する処理を行う場合に，必要なジョブの回数を削減することによって高速化を行い，PEGASUS [15] はMapReduce上でのベクトル演算を最適化して，特にshuffleするデータ量を削減することによってベクトル演算を伴う分析アルゴリズムの高速化を実現している．MapReduce上での機械学習アルゴリズムに手を加えることができれば，MapReduceジョブの回数やshuffleするデータ量を削減することによって処理を効率化できると考えられる．

7. むすび

我々は，MapReduceジョブの中でパラメータの影響する部分を検知することによって，自動的に複数の機械学習処理を共有可能な範囲を最大化することのできる新しいフレームワークを提案した．そのフレームワークをHadoopを拡張することで実装し，実際にハイパパラメータ調節に伴う複数の機械学習処理を共有化した際の共有化による総処理時間単出の効果を確認した．

本研究にはもう一つの課題がある．

特徴選択　特徴選択 [23] も機械学習アルゴリズムは処理結果の質を最適化するために必要な要素であり，機械学習を高コストにする要因である．たいていの機械学習は入力データとしてベクトルデータセットを用いる．ベクトル（行）は複数の説明変数（列）とその値で構成されており，最適化された処理結果を得るためには，ベクトルのどの説明変数を学習処理の対象とするかを選択しなければならない．説明変数には有用なものもあれば雑音的なものもあるためである．

我々は説明変数サブセットが異なるが一部重なりのある複数の機械学習処理を共有化することを考慮している．そのためにはカラムストア [24] の技術が有用となる可能性がある．

文　献

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Commun. ACM, vol.51, no.1, pp.107–113, 2008.
[2] T.K. Sellis, “Multiple-query optimization,” ACM Trans. Database Syst. (TODS), vol.13, no.1, pp.23–52, 1988.
[3] T. Nykiel, M. Potamias, C. Mishra, G. Kollios,

and N. Koudas, “MRShare: Sharing across multiple queries in MapReduce,” Proc. VLDB Endowment, vol.3, no.1-2, pp.494–505, 2010.
[4] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, “YSmart: Yet another SQL-to-MapReduce translator,” ICDCS, vol.3, no.1-2, pp.494–505, 2011.
[5] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, “Comet: Batched stream processing for data intensive distributed computing,” Proc. 1st ACM symposium on Cloud computing, pp.63–74. ACM, 2010.
[6] Facebook Data Infrastructure Team, “Hive - A warehousing solution over a map-recduce framework,” Proc. VLDB Endowment, vol.2, no.2, pp.1626–1629, 2009.
[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: A not-so-foreign language for data processing,” Proc. 2008 ACM SIGMOD International Conference on Management of Data, pp.1099–1110, ACM, 2008.
[8] S. Chen, “Cheetah: A high performance, custom data warehouse on top of MapReduce,” Proc. VLDB Endowment, vol.3, no.1-2, pp.1459–1468, 2010.
[9] M. Isard, M. Isard, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” ACM SIGOPS Operating Systems Review, vol.41, no.3, pp.59–72, 2007.
[10] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst, “HaLoop: Effcient iterative data processing on large clusters,” Proc. VLDB Endowment, vol.3, no.1-2, pp.285–296, 2010.
[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, “Twister: A runtime for iterative MapReduce,” Proc. 19th ACM International Symposium on High Performance Distributed Computing, pp.810–818, ACM, 2010.
[12] M. Weiser, “Program slicing,” Proc. 5th International Conference Software Engineering, pp.439–449. IEEE Press, 1981.
[13] N.D. Jones, C.K. Gomard, and P. Sestoft, Partial evaluation and automatic program generation, Peter Sestoft, 1993.
[14] C.W. Hsu, C.C. Chang, and C.J. Lin, et al., A Practical Guide to Support Vector Classification, 2003.
[15] U. Kang, C.E. Tsourakakis, and C. Faloutsos, “PEGASUS: A peta-scale graph mining system - Implementation, observations,” Data Mining, 2009. ICDM’09, Ninth IEEE International Conference on, pp.229–238, IEEE, 2009.
[16] B. Panda, J.S. Herbach, S. Basu, and R.J. Bayardo, “PLANET: Massively parallel learning of tree ensembles with MapReduce,” Proc. VLDB Endowment, vol.2, no.2, pp.1426–1437, 2009.
[17] C.-T. Chu, S.K. Kim, Y.-A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun, “Map-reduce for ML on multicore,” NIPS, vol.19, p.281, 2006.
[18] C. Cortes and V. Vapnik, “Support-vector networks,” Mach.-Learn., vol.20, no.3, pp.273–297, 1995.
[19] S. Shalev-Shwartz, Y. Singer, and N. Srebro, “Pegasos: Primal estimated sub-GrAdient SOlver for SVM,” Proc. 24th International Conference on Machine learning, pp.807–814, ACM, 2007.
[20] J.D.M. Rennie, L. Shih, J. Teevan, and D.R. Karger, “Tackling the poor assumptions of naive bayes text classifiers,” Machine Learning-International Workshop then Conference, vol.20, p.616, 2003.
[21] J.A. Hartigan and M.A. Wong, “Algorithm AS 136: A K-means clustering algorithm,” Applied Statistics, pp.100–108, 1979.
[22] A. Abouzeid, K. BajdaPawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, “HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads,” Proc. VLDB Endowment, vol.2, no.1, pp.922–933, 2009.
[23] A. Jain and D. Zongker, “Feature selection: Evaluation, application, and small sample performance,” IEEE Trans. Pattern Anal. Mach. Intell., vol.19, no.2, pp.153–158, 1997.
[24] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik, “C-store: A column-oriented DBMS,” VLDB Endowment, pp.553–564, 2005.
[25] P. Bhatotia, A. Wieder, R. Rodrigues, U.A. Acar, and R. Pasquin, “Incoop: MapReduce for incremental computations,” Proc. 2nd ACM Symposium on Cloud Computing, p.7, ACM, 2011.
[26] 福本佳史，鬼塚　真，“複数分析処理における MapReduce 最適化，” DEIM, 2011.

（平成 24 年 7 月 3 日受付，10 月 29 日再受付）

福本　佳史

2009 慶大･環境情報卒．同年，NTT 入社．

鬼塚　　真　（正員）

1991 東工大･工情報工学卒．同年，NTT 入社．2000～2001 年ワシントン州立大学客員研究員．現在，日本電信電話（株）ソフトウェアイノベーションセンタ 特別研究員．博士（工学）