MapReduceを利用した決定木生成処理の負荷分散

福本　佳史† 　藤岡　健吾† 　鬼塚　真†

† 日本電信電話株式会社　NTT サイバースペース研究所　〒 239–0847 神奈川県横須賀市光の丘 1–1
E-mail: †{fukumoto.yoshifumi,fujioka.kengo,onizuka.makoto}@lab.ntt.co.jp

あらまし　本論文では，大規模なデータセットを教師データとした分散処理による決定木生成を高速に行う問題を扱う．既存研究においては決定木生成の分散処理は複数回の MapReduce によって実現されているが，提案手法では既存研究をベースに，MapReduce に特化した負荷分散を行う．提案手法は，MapReduce によって決定木の各ノードを展開するとき，（1）生成中の決定木の状態や教師データの偏り方に応じた分散粒度の動的な変更と，（2）ノード展開処理の負荷が平均的になるようなクラスタ上の各マシンへの処理の割り振りにより構成される．結果として教師データの極端な偏りにより発生するノード間の負荷の差を抑え，MapReduce 中に発生する同期待ち時間を削減を試みる．

キーワード　分散処理, クラウドコンピューティング, MapReduce, データマイニング, 機械学習, 決定木

Load sharing of MapReduce for decision tree learning

Yoshifumi FUKUMOTO†, Kengo FUJIOKA†, and Makoto ONIZUKA†

† NTT Cyber Space Laboratories, Nippon Telegraph and Telephone corporation　Hikarinooka 1–1, Yokosuka-shi, Kanagawa, 239–0847 Japan
E-mail: †{fukumoto.yoshifumi,fujioka.kengo,onizuka.makoto}@lab.ntt.co.jp

Abstract　The goal of this work is to learn decision trees efficiently from massive dataset by distributed processing. An existing approach learns decision trees by distributed processing based on multiple iterations of MapReduce. Our approach aims to improve the response time of the decision tree leaning by balancing the loads among machines. This approach is based on the following two ideas in expanding nodes of decision trees with MapReduce: (1) changing the granularity of process distribution dynamically according to the load of those nodes, and (2) assigning expanding node processes to machines so as to balance the loads among those machines. We can reduce difference between loads of expanding nodes to reduce latency time in MapReduce processing.

Key words　distributed processing, cloud computing, MapReduce, data mining, machine learning, decision tree

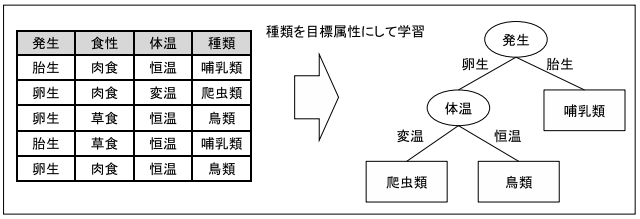
1. はじめに

近年の記憶装置の発達やインターネットの普及などによって，企業が取り扱う情報の量が爆発的に増加しつつある．そしてそれらの膨大な情報はただ蓄積されるだけではなく，データマイニングによって有益な知識が抽出され，企業のマーケティングや Web サービスなど様々な分野において有効活用されるようになった．必然的にそのような大規模な情報を扱っている企業では，情報をより高速に処理したいといった要求が高まっている．

特に Google や Yahoo!などの検索サービスを提供する企業は，膨大な Web データやログデータを日々処理し続けているが，それを 1 台のコンピュータで行うことは難しいため，複数台のコンピュータを用いて分散処理をしている．特に Google では，大量のコンピュータからなるクラスタを構築し，その上で独自のファイルシステムである GFS（GoogleFileSystem）[1] を運用し，巨大なデータを保管している．さらに GFS を前提とした分散処理フレームワークである MapReduce [2] によって大規模な分散処理を行っている．

本論文では，機械学習の一つである決定木の生成に着目し，決定木の性質に応じた工夫を加えることによって，MapReduce を利用した分散処理による決定木生成の高速化を試みる．そのために，まず最もよく知られている決定木生成アルゴリズムの一つである ID3 を，既存研究 [3] を参考にしながら多段の MapReduce の形に適用させた．その上で以下の問題点に対応する手法を考案し，その一部を実装して実験による評価を行った．

一つ目の問題点として，既存研究である PLANET [3] では，木の深さ 1 につき一回の MapReduce を行うことで決定木の展開を行っており，木の深さ毎の MapReduce 処理を同期することになる．しかしこの手法では，極端な偏りを持つデータセッ



発生	食性	体温	種類
胎生	肉食	恒温	哺乳類
卵生	肉食	変温	爬虫類
卵生	草食	恒温	鳥類
胎生	草食	恒温	哺乳類
卵生	肉食	恒温	鳥類

図 1　教師データの例と決定木の例

トを教師データとして学習させた場合に，Map 処理から出力される大量の中間データが，分散処理クラスタに所属する複数台のサーバマシンのうちの一部分だけに集中してしまい，一部のマシンの Shuffle 処理および Reduce 処理の負荷が高くなって，結果としてその他のマシンのリソースが無駄になることが予想された．

二つ目の問題点として，PLANET では木のノードと教師データの属性を Map 結果の Key として出力するが，決定木の成長に伴って一度の MapReduce で処理すべきノードの数が増えると，特に多くの値を取りうる属性が教師データに含まれていた場合には Key の種類が急激に増加するため，Map 処理によって大量に中間データが出力され，それをやり取りするための通信コストが増加する可能性がある．

一つ目の問題に対しては，処理の負荷が大きくなることが予測されるノードを検出し，より多くのマシンで処理出来るように Map 処理時に出力する Key の粒度を動的に細かくする．さらに，各クラスタ上の各マシンへ担当する Key を割り振る Partitioner を，単純に数が均等になるようにする従来のものではなく，各ノードの予測される負荷に応じた割り振りを行うものに変更する．二つ目の問題に対しては，Key の種類が極端に多くなることを検出し，逆に Key の粒度を意図的に粗くすることで，中間データの数を削減する．

今回は多段の MapReduce の形に対応させた多分岐 ID3 によって決定木を生成するプログラムを対象に，Map 出力結果における Key 粒度の動的な変更の効果を確認するため，Key 粒度を処理対象のノード ID のみのものと，ノード ID と教師データの属性のインデックスの組の 2 パターンを実装した．

Key の粒度の動的な変更と，負荷に応じて割り振りを行う Partitioner の実装は本論文の段階では完了していないため，提案手法の効果を評価することが出来ていないため，今後の課題とする．

2. 前 提 知 識

2.1 決　定　木

決定木はデータマイニング手法の中でも教師あり機械学習の一つとして知られており，マーケティングや医療などの様々な分野に応用されている．事物の特徴を幾つかの属性に分けて説明するレコードの集合を教師データとし、その中の特定の属性を目標属性として，機械学習によって図 1 のようなツリー構造のルールを導き出すものである．目標属性が欠落したレコードがあったとき，決定木のルートノードから順にノードを辿ってゆくと最終的にリーフノードに到達する．その葉ノードに書

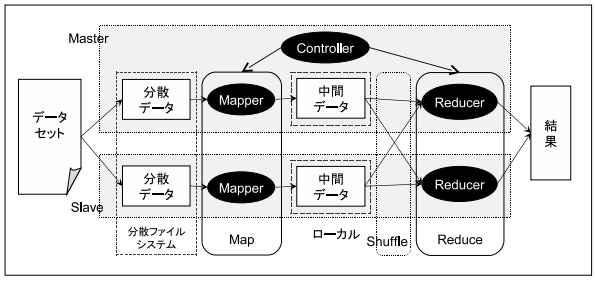


図 2　MapReduce 処理の流れ

かれている値が目標属性に入るべき値を予測したものとなっている．

2.1.1 決定木の生成

目標属性以外の属性において，同じ値を持つレコードが集まるように，もしくはある数値を基準として大小の値を持つレコードが集まるように教師データを分割し，その分割がどれだけうまく目標属性を分割できたか表すスコア（例えば情報利得や GINI 係数）を計算する．これを全属性において試行し，最も高いスコアで分割できた属性を，ルートノードの分割条件として採用し，木を成長させる．そして分割後の教師データを対象に同様の処理を繰り返し，目標属性の値が一種類だけになるまで，または予め設定した深さになるまで木を成長させ，最後に過学習となっている枝を剪定した上で，ツリー状のルールを出力する．

2.1.2 大規模な教師データ

従来の決定木生成アルゴリズムは，教師データ全体をメモリに読み込んで，実際に教師データを分割ないし分割後の状態になるような参照を保持しながら各属性の分割スコアを計算するが，教師データがメモリに収まりきらない規模の場合は，教師データを先頭から一行ずつシーケンシャルに読み込みながら処理を進める必要がある．そのため，高速化するには分散処理を行うことは有効な手段である．

2.2 MapReduce

GFS は大規模なデータを安定的に保管することができる分散ファイルシステムである．その上で動く MapReduce は GFS の特長を活かした分散処理フレームワークであり，大規模なファイルを高速に分散処理するための技術である．ユーザは分散処理に関わる複雑な処理をあまり意識することなく，Map 関数と Reduce 関数を定義するだけで分散処理を実現できるため，極めて生産性が高い．

2.2.1 MapRedcue 処理の流れ

図 2 のように，まず処理対象のデータを分散ファイルシステムに格納する．そして Controller がクラスタ上の各マシンの Mapper タスクを起動する．Mapper は各々ローカルに保持しているブロックをシーケンシャルに読み込み，1 レコードずつ任意の処理加工（Map）をした上で Key と Value のセットを中間データとして出力する．その間 Controller によって各マシンで Reducer が起動され，Reducer は同じ Key を持ったものを 1 台のマシンに集め（Shuffle），さらに任意の処理加工（Reduce）をしてそれを結果として出力する．

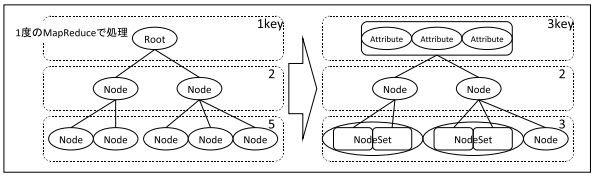


図 3　Key の粒度の変更

3. 従 来 手 法

3.1 MapReduce を利用した決定木生成処理

MapReduce の決定木生成アルゴリズムへの適用は，PLANET によって既に実現されている．PLANET は生成中の決定木（中間木）と展開可能なノードを格納したキューをクラスタに所属するマシン全体で共有しながら，複数のノードを一度の MapReduce によって展開し，それを複数回繰り返すことで決定木を成長させている．MapReduce 処理は，毎回教師データ全体をスキャンし，処理対象のノードの各属性の分割スコアの計算に必要な統計情報を収集するために利用されている．

MapReduce を用いて決定木を生成する場合，ルートノードから順にトップダウンで木の深さ 1 に所属する全ノードを，1 回の MapReduce によって処理することを繰り返す．これは分散ファイルシステムの性質上教師データへのランダムアクセスが難しく，また木の成長に伴って実際に教師データそのものを分割し，サブセットを出力するにも大きなコストがかかるためである．つまり各深さにおける MapReduce 処理毎に同期が必要となり，決定木のノード毎に独立して処理を進めることができない．指摘した問題点は全てこの同期に起因するものである．

決定木生成の過程で教師データを何度も分割するが，分割後のレコード数が一定以下になり，メモリ内にデータセットが全て収まるようになった場合には，元の教師データから MapReduce を利用してノード別にレコードを抽出し，それ以降の処理をオンメモリに切り替えて，該当するノード以下の木を 1 台のマシンで独立して成長させている．オンメモリでの処理の方が当然高速なため，この動的な処理の切り替えは非常に効果が大きく，決定木の深さ 10 の段階で，切り替えなしのものよりも処理速度が約 8 倍程度となっている．

3.2 問　題　点

また，PLANET では，Key の中身がノード ID と属性の情報の 2 つに固定されているため，生成中の決定木の深さによって Map から出力される Key の数が増加してゆく．木が浅い段階では，ルートノードの展開時に Map から出力される Key の数が少ないため，複数台あるマシンの一部でのみで Reduce が動作することになる．逆に生成中の決定木が深くなった時には，処理対象のノード数増加に伴って Key の種類が爆発的に増加し，中間データを出力するための Map 処理と，大量に出力された中間データを各 Reducer に収集するための Shuffle 処理のコストが高くなる可能性がある．途中からオンメモリ処理に切り替わるためその影響は緩和されるが，教師データが超大規模になった場合や偏りが大きい場合には対応出来ない．

4. 提 案 手 法

今回は，決定木生成アルゴリズムの中でも広く知られている多分岐 ID3 を MapReduce に適用させて実装をした．

4.1 Key の粒度変更による負荷分散

1 章で説明した 1 つ目の負荷が分散できないという問題と，2 つ目の中間データが増え過ぎる問題に対して，クラスタに所属しているマシンの台数と，決定木の生成途中で知り得る次の MapReduce で処理対象となるノードの数や，各ノードに分類されるレコード数などの情報を利用し，MapReduce 時に出力される Key を例えば図 3 のように決定木の各深度での粒度を動的に変更することで，生成される Key の数が最適になるよう調節する．具体的には，Map 処理において，各レコードがどのノードに分類されるかを判定（Algorithm1 line 1）し，そのノードに分類されるレコード数が処理対象ノードの平均値を大きく上回っている場合や，Key の数が多くなる場合を検出（Algorithm1 line 3,7,9）し，それに応じた粒度の Key を持つ中間データを出力（Algorithm1 line 5,8,10）する．Reduce 処理では中間データの Key の粒度を判定（Algorithm2 line 2,6,15）し，それぞれ集約した上で分割スコアを計算（Algorithm2 line 3,9,17）し，部分木を生成（Algorithm2 line 4,12,20）して結果を出力（Algorithm2 line 5,13,21）する．Key の粒度が細かく，Reduce の段階で一つのノードに関する全ての統計情報が集まらない場合（Algorithm2 line 2-5）には，MapReduce が完了して処理が Controller に戻った後にさらに Reduce 結果をマージ（Algorithm3 line 7,8）し，最適な分割をする属性を決定した上で木を更新（Algorithm3 line 17-19）する必要がある．

Algorithm 1 Map

Require: 処理対象ノード群 N, 中間木 T, レコード R(属性群 X, 目標属性 y) $\in$ 教師データ D

1: n = traverseTree(T, R)
2: **if** $n \in N$ **then**
3:　**if** load(n) > average(N) * 閾値 || マシン台数 > $|N|$ **then**
4:　　**for all** $x \in X$ **do**
5:　　output($k(n, x), v(\{\text{value}(x) : \{y : 1\}\})$)
6:　　**end for**
7:　**else if** load(n)| < average(N) / 閾値 || マシン台数 * 閾値 < $|N|$ **then**
8:　　output(k(NodeSet N'), $v(\{n : \{\text{value}(x) : \{x : \{y : 1\}\}\}\})$)
9:　**else**
10:　　output($k(n), v(\{x : \{\text{value}(x) : \{y : 1\}\}\})$)
11:　**end if**
12: **end if**

4.2 負荷の平均化

Mapper が処理を終えた後，同じ Key を持つ KeyValue の組をクラスタ上の各マシンで起動された Reducer に割り振る際，PLANET では単純に担当する Key の数が均等になるようにマシンを決定している．これは Key 毎の Reducer への割り振りを担っている Partitioner クラスが，初期状態ではハッシュ値を利用したものとなっているからである．Map 処理によって出力された中間データの Key の数がクラスタを構成するマシン

Algorithm 2 Reduce

Require: Key k, Value V

```
1: HashMap M = Aggregate(V)
2: if k == n, x then
3:    s = calcurateScore(M)
4:    t = createPartTree(M)
5:    output(k, v(s, t))
6: else if k == N' then
7:    for all n ∈ A do
8:       for all x ∈ X do
9:          s = calcurateScore(getAttributeStatistics(M, n, x))
10:         updateBestSplit(s, x)
11:      end for
12:      t = createPartTree(M)
13:      output(k, v(best(x, t))
14:   end for
15: else
16:   for all x ∈ X do
17:      s = calcurateScore(getAttributeStatistics(M, x))
18:      updateBestSplit(s, x)
19:   end for
20:   t = createPartTree(M)
21:   output(k, v(best(x, t))
22: end if
```

Algorithm 3 Controller

Require: 目標属性 y, 教師データ D

```
1: 中間木 T = createPartTree()
2: ノードキュー Q = createNodeQueue(getRootNode(T))
3: while !isEmpty(Q) do
4:    NodeList N
5:    (K, V) = mapreduce(y, D, T, Q)
6:    for all (k, v) ∈ (K, V) do
7:       if k == n, x then
8:          (k', v') = updateBestSplit(N, n, x, v)
9:       else if k == N' then
10:         for all n ∈ N' do
11:            updateTree(T, n, v)
12:         end for
13:      else
14:         updateTree(T, n, v)
15:      end if
16:   end for
17:   for all n ∈ N do
18:      updateTree(T, n, v)
19:   end for
20:   updateNodeQueue(Q, T)
21: end while
22: output(T)
```

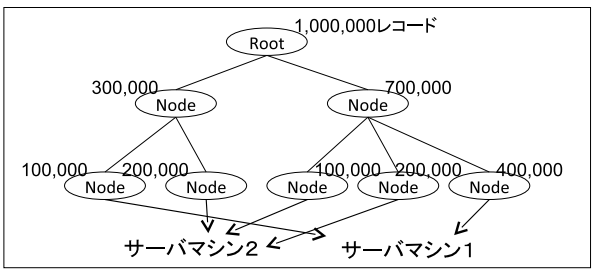


図 4　ノード負荷の平均化

の数を上回った場合は，1 台のマシンが複数の Reduce 処理を行うことになるが，特定のノードに関するレコードが極端に多い時や，ある属性だけ突出して値の種類が多い場合などは，特定のマシンに処理の負荷が大きい Key が集まる可能性がある．

Algorithm 4 Partitioner

Require: ノードキュー Q, Reduce タスク数 r

```
1: sortByLoad(Q)
2: TaskID t = 0
3: for i = |Q|%r;   i > 0;   i − − do
4:    n = unshift(Q)
5:    setTaskID(n, t + +)
6: end for
7: SetSize z = |Q|/r
8: while !isEmpty(Q) do
9:    n = unshift(Q)
10:   setTaskID(n, t)
11:   for i = z − 1;   i > 0;   i − − do
12:      n = pop(Q)
13:      setTaskID(n, t)
14:   end for
15:   t + +
16: end while
```

これを防ぐために，単純なハッシュを利用した Partitioner の代わりに，図 4 のように予測された Key 毎の負荷に応じてクラスタの各マシンに Key を割り振ることが出来るよう設定した Partitioner を使用する．具体的には，例えば，処理対象ノードが格納されているキューを予測される負荷でソート（Algorithm4 line 1）し，負荷の大きいノードは除外（Algorithm4 line 3-6）した上で，キュー中の負荷の高いノードと（Algorithm4 line 8），低いノードを幾つか取って組にする (Algorithm4 line 10-13）．

5. 予 備 実 験

決定木生成の中でも離散値のみを対象とした，広く知られているアルゴリズムである多分岐の ID3 を，PLANET を参考に，Hadoop を利用して実装した．予備実験では Map 処理の時に出力される Key の粒度を変えた 2 種類の実装を使って，偏りのある人工データと偏りのない人工データ，実データの 3 種類の教師データを学習させた．これにより Key の粒度別に教師データの偏りと木の深さによる影響を調べる．

5.1 実 験 条 件

実験環境として Celeron 430(1.8GHz), 2GB RAM, 500GB HDD のサーバ 5 台によるクラスタを利用した．教師データは偏りの無い例として，7 属性，各属性が取りうる値が 0~ 9 の乱数となるレコード 1 千万行分のものと，極端な偏りがある例として 7 属性，各属性値が取りうる値は 0~ 99 の乱数だが，99 %の確率で 0 の値を取るレコードを 1 千万行分，どちらも約 150MB の人工データを与えた．実データとしては UCI Machine Learning Repository にある 1990 年度の米国国勢調査のサンプリング（約 350MB）を利用した．実験は，Key を展開対象となっているノードの ID のみとしたものと，ノード ID に各属性のインデックスを加えたものの 2 パターンのプログラムを用い，決定木を出力する際に各段階の MapReduce に

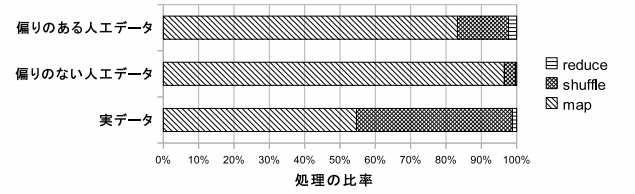


図 5　MapReduce 中の各処理の比率

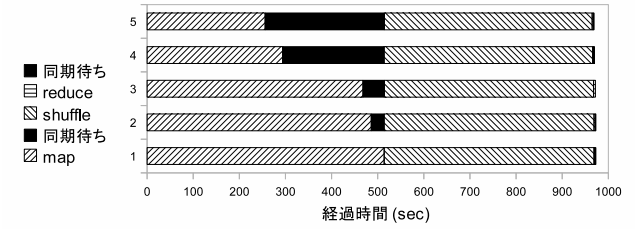


図 6　各マシンにおける MapReduce 処理時間の内訳

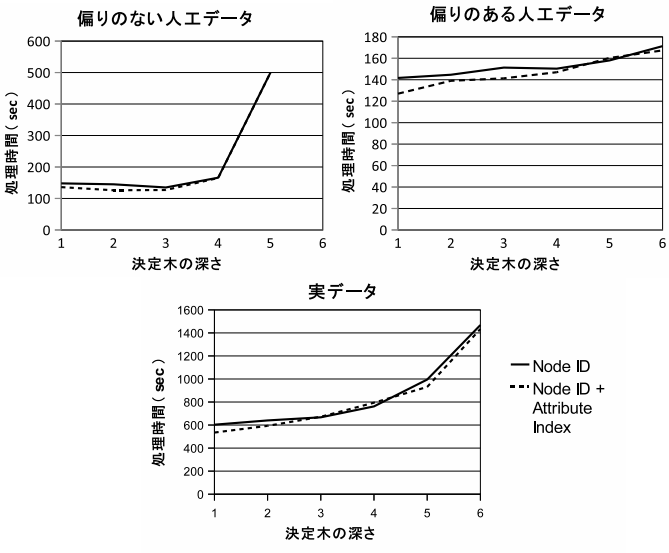


図 7　決定木の深さと MapReduce 処理時間

要した処理時間を計測した．

5.2　考　　察

結果は図 5,6,7 のようになった．図 5 の処理の比率のグラフはそれぞれ決定木の深さが 5 の段階のもので，図 6 の処理時間の内訳は粒度の細かい方の実装を使って実データを教師データとし，深さ 5 の時のものである． 図 7 より、どの教師データにおいても木の深さに伴って処理時間が増加する傾向にあることが分かる．これは木の深さに伴って処理すべきノードが増加したことと，Map 処理の際に各レコードが決定木のどのノードに分類されるかを木を辿って調べる処理の負荷が大きくなったことが原因として考えられる．特に偏りがない人工データに関しては木の深さ 4 を超えると一挙に処理時間が増加しているが，偏りの全く無い教師データを学習する場合が最も処理対象ノードの増加が速いからであると考えられる．

木が浅い段階では，Key をノード ID と属性のインデックスとした粒度の細かいほうが，僅かに速い傾向がある．これに関しては，Key の種類の増加に伴なって Shuffle 処理と Reduce 処理のコストが大きくなることはほとんどなく，Map と Combine 処理部分の性能差によるものであると考えられる．より深くまで木を成長させる場合やより多くのマシンから構成されるクラスタでの実験結果も欲しいところだが，今回の結果からは Key の粒度は処理速度にそれほど大きな影響を及ぼさないことが分かる．

当初は教師データの偏りが大きい場合に，各ノードに分類されるレコードの数に大きな差が生じるために，同じ Key を持った中間データが大量に出力され，その Key を担当する Reducer に多くの中間ファイルが集中してしまって，そこだけ負荷が高くなるのではないかと予想していたのだが，Map 処理の後のローカル Reduce とも言える Combine 処理の効果が思いのほか大きく，出力される中間データの数が大幅に削減されたため，Shuffle や Reduce のコストが大きくなることがなく，教師データの偏りによる影響はほとんど無かった．

図 5，図 6 より最も重い部分は Map 処理であることが分かった．図 6 のようにクラスタに属する各マシンにおいて Map の処理時間に 2 倍近くの差が発生することがあり，その場合には Map 処理が全て完了するまで Reduce 処理が開始されないため，早く Map 処理が完了したマシンのリソースが無駄になる．これは Mapper が担当する教師データの部分によって処理の負荷に差によるものであると考えられる．今回の提案手法は Shuffle 処理と Reduce 処理に関するものであるため，効果が出たとしてもその貢献度は低くなる．提案手法とは別に，Map 処理を削減・効率化する手法を考える必要がある．

今回は実験環境としてサーバマシン 5 台のクラスタを利用し，教師データも 1GB に満たないサイズのものを使ったが，MapReduce は本来数百台のマシンにおいて数十~ 数百 GB のデータを処理するためのものである．その規模になると，今回のような小規模な実験では発生しない問題が顕在化する可能性がある．今後はより大きなデータセットを用意し，より多くのマシン環境での実験が課題となる．

6.　議　　論

6.1　ボトルネックの解消

今回の実装と実験では，決定木を MapReduce を利用して生成する場合，Map 処理が一番のボトルネックとなることが判明した．マシン間で Mapper の処理時間に 2 倍近い差が生じているため，これを解消することが出来れば高速化に繋がる．そのためには先述したように，Map と Reduce の間の同期が障害となっている．

同期そのものを解消する手段として，Reducer に同じ Key を持った中間データを収集する処理を，プル型ではなくプッシュ型に変え，Mapper から中間データが出力されたタイミングでそのまま Reducer に送るようにすることが考えられるが，その場合は Reducer がそれぞれ担当する Key を持った中間データを全て集め終わったと判断する基準を失ってしまう．そのため，そのようなパイプライン型の処理は，それぞれの Reducer が必要とするレコードが単一もしくは数が決まっている場合のみ有効となり得る．決定木の場合は，ある深さのノードを処理

する際，処理対象のノードの数とそれぞれのノードに分類されるレコードの数を，一段階前の統計情報より得ることができる．予め処理対象のノードの数だけ起動された Reducer は，入力された中間データの数が必要なレコード数に達した時点で統計情報が揃うため，担当するノードの次の分割条件を決定する事が出来る．つまりノード毎に独立して処理を進めることが出来るようになる．次の MapReduce をどのタイミングで開始するのか，複数の MapReduce 処理を同じクラスタ上で開始する弊害はどの程度か，など様々な問題が残るが，パイプライン型の MapReduce が実現すれば決定木生成処理をより高速化出来る可能性がある．

その他の Map 処理の負荷を軽減する手段として，決定木の各ノードにおいて分割条件を決定した際に，別の MapReduce 処理によって教師データそのものも分割してしまって，サブセットを次の MapReduce に入力することが考えられるが，そのためのコストが大きくなる可能性がある．もし分割するとしても，決定木の浅い段階でのみ行うのが良いと考えられる．

6.2 他の機械学習への適用

本論文の提案手法は MapReduce を利用した決定木生成アルゴリズムにはあまり向いていないという結果となったが，その他の機械学習アルゴリズムを MapReduce に適応する際，特に Reduce 処理がボトルネックになるものや，Combiner によって中間データを集約することが出来ないものがあれば，効果を発揮する可能性がある．

7. 関 連 研 究

7.1 決 定 木

決定木は，エントロピーを分割のスコアリングに利用し離散値属性のみを対象とした ID3 [4] を代表として，その後登場した GINI 係数をスコアリングに利用する CART [6] や，情報利得を利用する C4.5 [5] などによって，連続値への対応，高精度化，高速化が図られてきた．分散処理を伴わないアルゴリズムの中では，特に大規模なデータを対象とした BOAT [10] が高速で，決定木を 1 段ずつ処理するのではなく，たった 2 回の教師データ全体のスキャンで決定木を生成してしまうというものである．どれもルートノードから順にトップダウンで処理を進める点が共通している．

その後大規模な教師データと並列化を想定した SPLINT [7] が登場した．SPLINT は教師データをレコード単位ではなく属性単位で管理し，前処理として教師データ全体のソートをした上で，決定木の深さ 1 段に所属するノードを一度に処理することで決定木を成長させる．それを参考にして実際に PC クラスタを用いた分散処理によって決定木生成を行い，台数効果が確認されている [8].

今回特に参考にした PLANET [3] は，MapReduce を生み出した Google が自社が持つ膨大なログデータなどから決定木を生成するためのフレームワークである．

7.2 その他の MapReduce を使った機械学習

複数台のマシンを利用する純粋な MapReduce ではないが，単純ベイズや k-means，サポートベクタマシンなど様々な機械学習が，複数の CPU を用いて MapReduce と同様の概念を利用した処理で実現されている [9].

8. おわりに

本論文では，MapReduce を応用して決定木生成処理を行う過程で発生する同期による処理待ち時間を，データセットの偏りと決定木の深さに基づいて削減する手法を提案した．具体的には，マシン台数や各ノードに分類されるレコード数を元にして，Map 処理時に出力される Key の粒度を動的に調節する．さらに各ノードの Reduce 処理の負荷が均等になるようにクラスタの各マシンに割り振る．

提案手法の効果を確認するために，予備実験として，Key の粒度を変えた実装を 2 パターン用意し，それぞれ 3 種類の教師データを用いて決定木を生成させたところ，木が深くなるほど処理時間が増加すること，Map が最も重い処理であること，木が浅い段階では Key の粒度が細かい方が僅かに優位であること，教師データの偏りは Shuffle や Reduce の処理速度にほとんど影響しないことが判明した．

提案手法をそのまま実装したとしても決定木生成を高速化することは難しく，実現するには Map 処理の高速化が不可欠であり，そのための手段が幾つか考えられるが議論の余地は多く残されている．

文　　献

[1] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, "The Google File System", SOSP 2003.

[2] Jeffrey Dean, Sanjay Ghemawat, "MapReduce:Simplified Data Processing on Large Clusters", OSDI 2004.

[3] Biswanath Panda, Joshua S. Herbach, Sugato Basu, Roberto J. Bayardo, "PLANET:Massively Parallel Learning of Tree Ensembles with MapReduce", VLDB 2009.

[4] J. Ross Quinlan, "Induction of Decision Trees", Machine Learnig 1986.

[5] J. Ross Quinlan, "C4.5 : programs for machine learning", The Morgan Kaufmann series in machine learning 1993.

[6] L.Breiman, J.H.Friedman, R.Olshen, C.Stone, "Classification and Regression Trees", Wadsworth and Brooks, 1984.

[7] Jhon Shafer, Rakesh Agrawal, Manish Mehta, "SPLINT: A Scalable Parallel Classifier for Data Mining", VLDB 1996.

[8] 久保田 和人, 中瀬 明彦, 酒井 浩, 小柳 滋, "PC クラスタを用いた決定木生成", 2000.

[9] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, Yuan Yuan Yu, Gary Bradski, Andrew Y. Ng, Kunlie Olukotun, "Map-Reduce for Machine Learning on Multicore", NIPS 2006, pp. 281-288.

[10] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, Wei-Yin Loh, "BOAT-Optimistic Decision Tree Construction", SIGMOD 1999.