平成 24 年度

学士学位論文

木構造データの縮約アルゴリズムの Hadoop 上での実現とその評価

Implementation and Evaluation of a Tree Reduction Algorithm on Hadoop

1130320　　尾形　勇磨

指導教員　　松崎　公紀

2013 年 3 月 1 日

高知工科大学 情報学群

要 旨

木構造データの縮約アルゴリズムの Hadoop 上での実現とその評価

尾形　勇磨

木構造は，XML データベースやファイルシステム等で用いられているデータ構造である．近年取り扱われるデータの容量は肥大化しており，それは木構造を持った XML データベース等でも例外ではない．そのため，より高速にデータを処理することがである．しかし，現状の CPU やメモリでは取り扱えるデータ量や処理速度に限界があり、1 台のコンピュータ機器を用いての処理では高速化は困難である．そこで複数のコンピュータ機器を用いて並列処理を行う技術が求められる．本論文では，大規模クラスタ向け並列計算フレームワークである MapReduce フレームワークの実装である Hadoop を用いて並列な木構造データの縮約アルゴリズムを実現した．そして，高速化可能であるか，どの程度効果があるのかを検証するした．実験は 16 台のコンピュータ機器により構成されたクラスタ上で並列プログラムを実行し処理時間を計測した．その結果，使用するコンピュータ機器を増やす事によって，処理を高速化することができた．また，並列化するために入力データを分割する際には，分割数を減らし過ぎると map による処理が大きくなり，処理時間が遅くなるという結果を得た．

キーワード　MapReduce，木構造データ，縮約アルゴリズム

Abstract

Implementation and Evaluation of a Tree Reduction Algorithm on Hadoop

Ogata Yuma

Tree-Structure is a data structure that is used in the XML document or file system. In recent years the amount of data to be handled has been enlarged,It is no exception in the XML document, such as using a tree structure processing. Therefore, it is required to process data faster. However, there is a limit to the amount of data that can be handled processing speed in the CPU and memory of the current situation. Therefore, in the high-speed process of using a single computer equipment is difficult. Technique is required to perform parallel processing using a plurality of computer equipment there. In this paper, it has achieved a Contraction algorithm of tree-structured data in parallel using Hadoop MapReduce framework is its implementation in parallel computing framework for large-scale cluster. And I have to verify whether it is possible to speed up or whether how effective they are. Experiments were run to measure the processing time on the cluster that is configured by the computer device 16. As a result, by increasing the computer equipment to be used, it was possible to speed up the process. Also, when dividing the input data to parallel, the results we have obtained map is increasing in processing by reducing the number of divisions, the processing time becomes slow.

key words MapReduce, Tree-Structured Data, Contraction Algorithms

目次

図目次

表目次

第 1 章

はじめに

1.1 背景と目的

木構造は，XML データベース [1] やファイルシステム [8] 等で用いられているデータ構造である．近年取り扱われるデータの容量は肥大化しており，それは木構造を持った XML データベース等でも例外ではない．そのため，より高速にデータを処理することが求められている．

しかし，現状の CPU やメモリでは取り扱えるデータ量や処理速度に限界があり、1 台のコンピュータ機器を用いての処理では高速化は困難である．そこで複数のコンピュータ機器を用いて並列処理を行う技術が求められる．

並列処理な処理を行うためには，MapReduce [2] を用いる方法がある．MapReduce とは，複数のコンピュータ機器によって構築されたクラスタ上で，大規模なデータを並列に処理するためのフレームワークである．また，大規模なデータをクラスタ上で扱うためには，MapReduce の実装の一つである Hadoop[10] を用いる方法が挙げられる．Hadoop とはクラスタ上の各コンピュータ機器を用いる分散ファイルシステムである HDFS (Hadoop Distributed File System) の実装と，並列に動作するジョブのスケジューリングとを行う，MapReduce フレームワークの実装である．しかし，MapReduce は木構造処理のようなデータ同士の依存性を利用するような処理には向いていないという問題がある．よって MapReduce を用いて並列処理を行うためには，データ同士の依存性のある木構造データを利用できるようにする必要がある．

本研究の目的は，木構造を分割しやすい形にすることで，大規模クラスタ向け並列計算

のフレームワークである MapReduce フレームワークによる並列処理をその実装である Hadoop を用いて木構造処理を実現することである．そして，高速化可能であるか，どの程度効果があるのかを検証する事である．

1.2　関連研究

Hadoop は Google によって提案された並列プログラミングモデルとそのフレームワークである MapReduce の実装である．MapReduce は並列計算技術の中でも，複数のコンピュータ機器をひとまとまりにすることで全体を一つのシステムとして並列計算を行うコンピュータ・クラスタと呼ばれる技術である．MapReduce ではネットワーク上のファイルシステムもしくはデータベースを用いることによって並列計算を行っている．

並列処理の実装としては Hadoop 以外にも MPICH[9] などがある．MPICH は MPI [4] (Message Passing Interface) と呼ばれる並列プログラミングの規格のライブラリ実装である．MPI ではメッセージパッシング方式と呼ばれる，プロセス間でメッセージを交信しながら処理することで並列計算を実現する方式を取っている．

1.3　論文の構成

本論文の構成は次の通りである．第 2 章では，MapReduce についての説明を行う．第 3 章では，本論文で用いるリスト表現された木構造データについて説明を行う．第 4 章では，MapReduce に対してどのように木構造処理を適用するのかの説明を行う．第 5 章では，木構造処理を並列化の効果を検証し，その評価実験の内容と結果について述べる．第 6 章で本論文のまとめと今後の課題を述べる．

第 2 章

MapReduce

本章では，MapReduce プログラミングモデルと MapReduce フレームワークについて述べる．

2.1 プログラミングモデルとしての MapReduce

MapReduce プログラミングモデルでは，プログラマは map と reduce と呼ばれる二つの処理を記述することによって，並列プログラムを実現できる．プログラム中でのデータのやりとりはキーと値のペアによって行われ，そのキーと値の型はプログラマが自由に指定することが可能である．

全体の処理の流れは図 2.2 に示す．map では入力のキーと値のペアを受け取り，中間的なキーと値のペアを生成する．map が終了した後には、同じキーを持つ中間出力のペア同士をひとまとめにして reduce に渡される．reduce は受け取った中間出力のペアから，最終的なキーと値のペアを生成し出力する．キーの型はユーザが自由に指定することができるが，map が生成する中間出力と reduce が受け取る入力の型は一致しなければならない．

プログラムの例として，英文を入力データとして受け取り，その単語毎の出現回数をカウントする WordCount を図 2.1 に挙げる．map は入力ファイルである英文から一行毎に分割されたものを入力データとして受け取り，その単語をキーとする定数とのペアを生成して reduce に渡す．reduce は複数の map から出力されたデータを，キー毎に受け取りその数をカウントすることで値を集約する．そしてその結果を各 reduce が出力する．

```
map(key,line){
   i = 0, word[] = line を単語毎の分割し，配列に格納;
   while(!(word[i]==null))
      key = word[i], i++, output(key,1);
}
reduce(key,values){
   sum = 0;
   for(value に values を要素がなくなるまで代入)
      sum++;
   output(key,sum);
}
```

図 2.1　WordCount の擬似コード

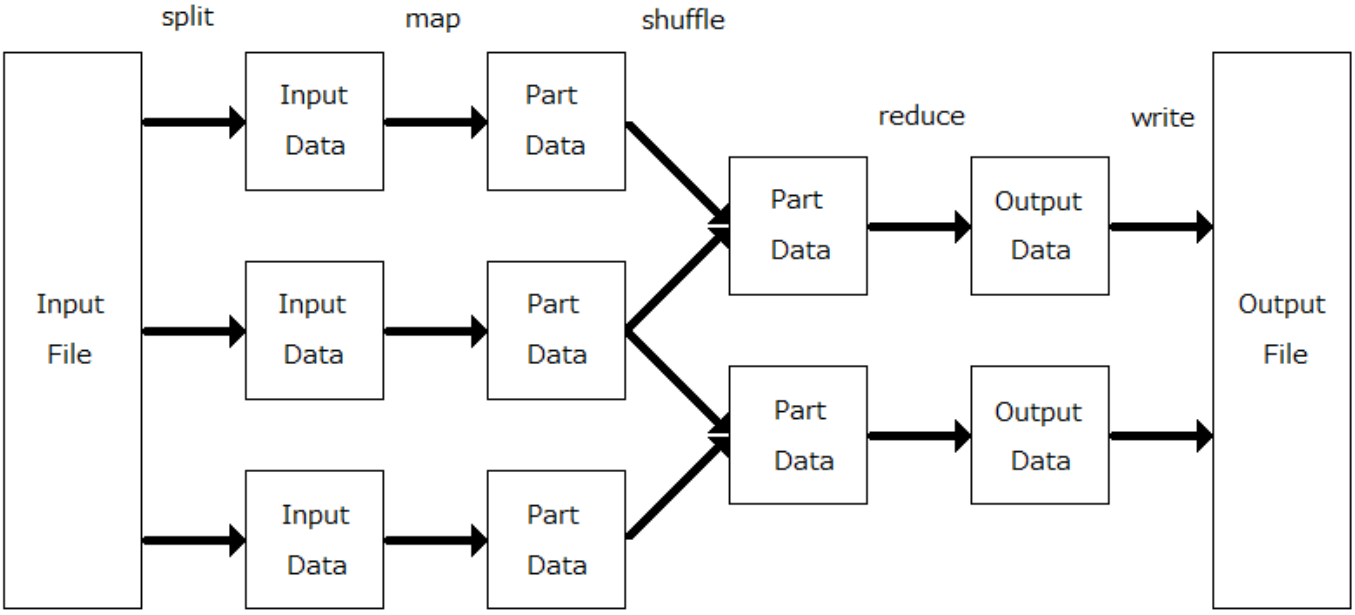

図 2.2　MapReduce の処理の流れ

2.2　フレームワークとしての MapReduce

MapReduce は，大規模なデータをクラスタ上で効率的に並列処理するための分散処理フレームワークである．また，Hadoop は分散ファイルシステムの実装とジョブの管理を行う，MapReduce フレームワークのオープンソース実装である．

2.2.1　MapReduce

MapReduce は，大規模クラスタ上の実行環境である．MapReduce フレームワークは 2.1 節で述べた map と reduce の 2 つのステップを並列に処理する．各ステップの入出力はキーと値のペアで行われ，その入出力データの受け渡しはフレームワークが行う．そのため，ユーザは各ステップでの処理と入出力の型を記述することによってクラスタ上で行われる並列処理を実装できる．

2.2.2　分散ファイルシステム

並列処理を行うデータは，データの容量が数テラバイトに及ぶような大規模になものとなる．このような大規模なデータを処理する場合，1 台のコンピュータ機器では容量が足りなくなる．Hadoop では HDFS とい分散ファイルシステムが実装されており，これを用いることによって 1 台のコンピュータ機器では扱い切れない大規模なデータをクラスタ上で扱うことができる．

第 3 章

木構造データのリスト表現と縮約

3.1　木のリスト表現

通常木構造データは，ノード間のポインタを利用した構造となっている．これは容易に挿入・削除を行うことが可能であり，木を変形させる場合においては有用である．しかしポインタを多様したデータ構造は木の変形を行わない場合は無駄となる．そのため木の深さを表す整数とそのノードの値 (open 要素) もしくは値がないことを示すタグ (close 要素) のペアによるリスト構造 [3] で木構造データを表現する．

この表現方法では，図 3.1 のように木を深さ優先で探索を行い，あるノードに入る時にその値を open 要素として取り出し，そのノード以下の子ノードの探索を終えそのノードから出るときに close 要素としてタグを追加することによって木構造データからリストを生成することができる．close 要素を”/” として図 3.2 の木構造データの例をリスト表現で示した場合は

$$[(0,72),(1,27),(2,12),(3,48),(3,/),(3,21),(4,28),(4,/),(4,6),(4,/),$$

$$(3,/),(2,/),(2,80),(2,/),(1,/),(1,21),(2,3),(2,/),(1,/),(0,/)]$$

となる．また図 3.2 のリスト表現を図示したものを図 3.3 に示す．値が記されている白い丸が open 要素であり、灰色の丸が close 要素である．

木のリスト表現では一つのノードに対して open 要素と close 要素が対になるように生成されることから，ノード数 N の木構造データのリスト表現の長さは 2N となる．また，部分木が親の区間に含まれているという特徴がある．つまり，深さ d の open 要素から始まり，

```
ChangeList(tree){
   output(tree の高さ,tree の値);
   if(tree の左の子が存在するか？)
      ChangeList(tree の左の子);
   if(tree の右の子が存在するか？)
      ChangeList(tree の右の子);
   output(tree の高さ,"/");
}
```

図 3.1　木構造データを表現リストへ変換するプログラムの擬似コード

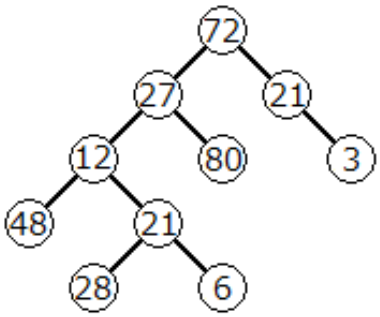

図 3.2　木構造データ

深さ d 以下の深さの要素を含まずに深さ d の close 要素で終わるような部分リストが与えられた場合，両端の open 要素と close 要素は同じノードに対応するペアであり，リストの内部はそのノードを根ノードとする部分木のリストとなる．

3.2　木の縮約

木の縮約 [7] は，葉ノードを親ノードにマージしていき (この操作を rake と呼ぶ)，根ノードが残るまで繰り返すことである．この実験では木の縮約の一つである maxPathSum[5] と呼ばれる任意の葉ノードから根ノードまでのパスに現れる値をすべて足し合わせた値のうち最も大きな値を返す計算を並列化する．図 3.2 の maxPathSum を求める工程を，図 3.4 で示す．葉ノードは同じ親ノードを持つ葉ノードとその値を比較して，より大きな値が親ノードの値と足し合わせられる．葉ノードは親ノードにマージされたことにより消失し，親ノードは葉ノードとなる．この行程を根ノードだけが残るまで繰り返す操作が maxPathSum の

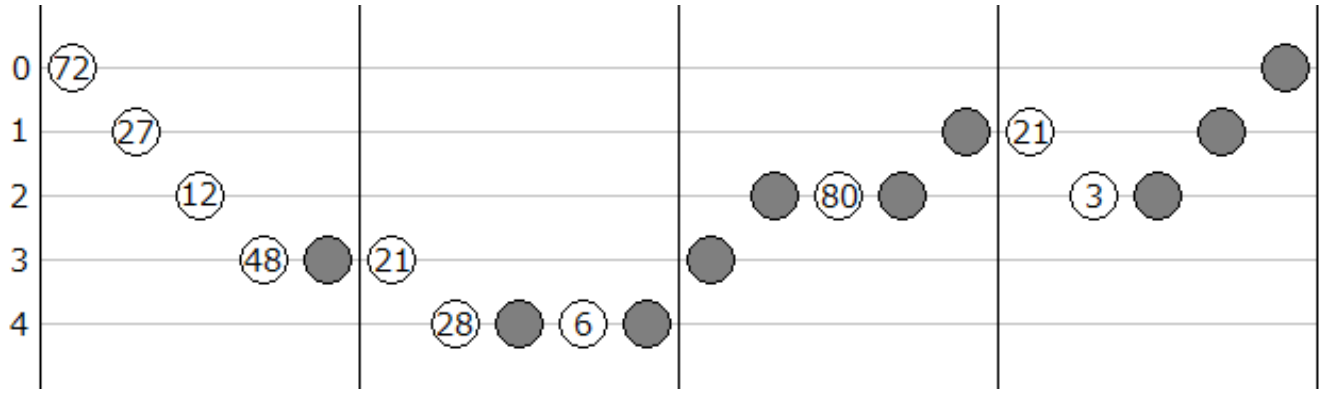

図 3.3　図 3.2 のリスト表現の解釈

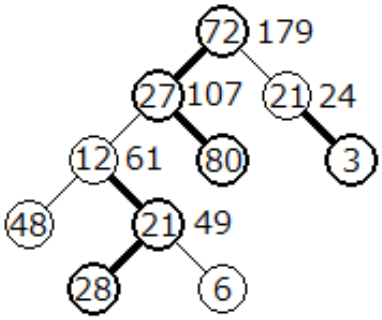

図 3.4　木の縮約

計算であり，図 3.4 の例では 179 の値が返される．

木のリスト表現には 3.1 節で示したように，両端が対応する open 要素と close 要素である部分リストの内部は部分木となるという性質がある．この事から対応する open 要素と close 要素を持った部分リストが与えられた場合，対応する部分木の縮約が行えることが分かる．

第 4 章

縮約アルゴリズムの MapReduce 上での実装

本章ではリスト表現された木構造データをどのように MapReduce を用いて縮約するかを記述する．MapReduce にはその実装である Hadoop を用いた．

木構造データの縮約アルゴリズムを MapReduce 実装する際に，map 処理では依存性の無い部分での計算を行い，reduce 処理では依存性のある部分での計算を行う．具体的には，map 処理で部分リストの縮約を行い，reduce 処理で縮約された部分リストをマージすることによって木全体の縮約を行う．

4.1 Hill Normal Form

リスト表現された木構造データの部分リストには，部分木として成立しており縮約が可能な谷の部分と，それ以外の丘の部分が存在する．この部分リストの谷を縮約した形を hill normal form (以降 h-NF) として定式化する [6]．h-NF は四つの要素からなる．

- 与えられた部分リストに対応する close 要素のない open 要素が格納されるリスト as.
- as に対応する同じ長さを持つリスト bs．as の要素の子を縮約した値が格納される．
- 対応する open 要素と close 要素が存在し、縮約された値が格納されるリスト cs.
- 丘の部分の高さを格納する d.

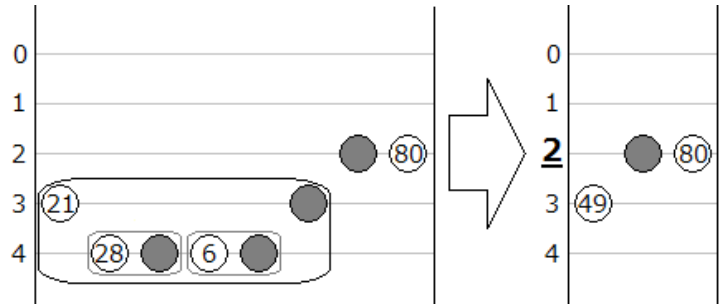

図 4.1　図 3.3 の部分リストと部分的に縮約した hill normal form

例えば図 3.3 の部分リスト

$$[(3,21),(4,28),(4,/),(4,6),(4,/),(3,/),(2,/),(2,80)]$$

に対応する h-NF は

$$d = 2$$

$$cs = [49,\ \iota\ ]$$

$$as = [80]$$

$$bs = [\ \iota\ ]$$

となる．ι は値を持たない要素であり，値を比較し大きな方の値を取得する演算に対する単位元である．図 4.1 は図 3.3 と同様にリストを図示したものである．図中の枠線で囲まれた要素の組は対応する open 要素と close 要素を持つ谷である．

対応する opne 要素と close 要素を持つ 1 番目から 6 番目の要素は 21 の値を持つノードを根ノードとする部分木となっていることから，縮約が可能である．その縮約結果と対応する open 要素を持たない close 要素である 7 番目の要素は cs に格納される．また，対応する close 要素を持たない open 要素である 8 番目の要素は as に格納され，bs は as と同じ長さである必要があるため，空の木が存在していると想定して ι が格納される．そして対応する要素を持たない要素で最大の高さである 2 が丘の高さとなり，d に格納される．

```
d = list[0] の高さ;
if(list[0] の値 == "/") cs.push( ι ), cs.push( ι );
else cs.push( ι ), as.push(list[0] の値), bs.push( ι );
for(int i=1;i<部分リストの長さ;i++)
   if(list[i] の値 != "/") as.push(list[i] の値), bs.push( ι );
   else
      if(as が空か？) cs.push( ι ), d = list[i] の高さ);
      else a = as,pop(), b = bs.pop();
         if(b == ι ) t = a;
         else t = a + b;
      if(bs が空か？) c = cs.pop();
         if(c < t) c = t, cs.push(c);
      else b = bs.pop();
         if(b < t) b = t, bs.push(b);
```

図 4.2 部分リストを h-NF 形式に変換するプログラムの擬似コード

4.2 map

map 処理ではリスト表現された木構造データの部分リストを受け取り，それを縮約し h-NF の形に変換し出力する．例として部分リスト $list$ を h-NF 形式に変換する処理を図 4.2 に示す．map は最初の要素の高さを d とし，その後丘が現れるとその時の高さを d とする．open 要素を見つけた時はその値を as に格納し，bs にその縮約された子ノードを格納するために単位元を格納する．close 要素を見つけた場合は as と bs の要素を取り出し，縮約を行う．ただし open 要素と対応を持たない close 要素の場合は丘となっているため，その要素の高さを d として取得する．

4.3 reduce

reduce 処理では縮約され h-NF に変換された部分リストを受け取り，それをマージし木構造データ全体を縮約した結果を出力する．map から受け取った h-NF を順番にマージして行き，最終的には単一の h-NF が生成され，その cs 要素が縮約結果となる．その処理を図 4.3 に示す．擬似コードでは配列 $array$ の長さを $|array|$ のように表し，$hnf_x = (d_x, cs_x, as_x, bs_x)$ に $hnf_y = (d_y, cs_y, as_y, bs_y)$ をマージした結果を $hnf_z = (d_z, cs_z, as_z, bs_z)$ に格納するとしている．h-NF 同士の縮約は，二つの h-NF を横に慣

```
if(dx < dy) dz = dx, n = |csy|, m = |asy|
   for(int i=0;i<|asx|-n+1;i++) asz.push(asx[i]);
   for(int i=0;i<m;i++) asz.push(asy[i]);
   for(int i=0;i<|asx|-n;i++) bsz.push(bsx[i]);
   v = Merge(hnfx,hnfy,n-1);
   if(bsx.get[|bsx|-n] < v)
      if(csy[n-1] > v) bsz.push(csy[n-1]);
      else bsz.push(v);
   else
      if(bsx[|bsx|-n] < csy[n-1]) bsz.push(csy[n-1]);
      else bsz.push(bsx[|bsx|-n]);
   for(int i=0;i<m;i++) bsz.push(bsy.[i]);
   for(int i=0;i<|csx|;i++) csz.push(csx[i]);
else
   dz = dy, n = |asx|+1, m = |csx|;
   for(int i=0;i<|asy|;i++) asz.push(asy[i]);
   for(int i=0;i<|bsy|;i++) bsz.push(bsy[i]);
   for(int i=0;i<m-1;i++) csz.push(csx[i]);
   v = Merge(hnfx,hnfy,n-1);
   if(csx[m-1] < v)
       if(csy[|csy|-n] > v) csz.push(csy[|csy|-n]);
       else csz.push(v);
   else
       if(csx[m-1] < csy[|csy|-n]) csz.push(csy[|csy|-n]);
       else csz.push(csx[m-1]);
   for(int i=0;i<|csy|-n;i++) csz.push(csy[n+i]);


Merge(hnf1,hnf2,int n){
   if(n==0) return ι;
   x = Merge(hnf1,hnf2,n-1);
   if(bs1[|bs1|-n] < x)
      if(cs2[n-1] > x) x = cs2[n-1];
   else
      if(bs1[|bs1|-n] < cs2[n-1]) x = cs2[n-1];
      else x = bs1[|bs1|-n];
   if(x != ι) x += as1[|as1|-n];
   else x = as1[|as1|-n];
   return x;
}
```

図 4.3 h-NF をマージするプログラムの擬似コード

れべた時に新しく生成される谷の部分を縮約することで行われる．open 要素と close 要素が対応していない丘の部分はそのまま新しく生成される h-NF に格納され，新たに谷として生成された部分は縮約されて格納される．

第 5 章

評価実験

本章では，実際にリスト表現された木構造データの縮約アルゴリズムを MapReduce の適用による並列化によって高速化することができたのか，どの程度の効果があるのかを検証する．実験に用いた機材を表 5.1 に示す．また実験に用いプログラムは付録 A に示す．

5.1 入力データの作成

実験ではリスト表現された木構造データを入力データとして扱う．この節では入力データの作成に使用する木構造データの生成アルゴリズムについて説明する．また，木構造データをリスト表現に変換する方法は 3.1 節で解説したものを使用する．

実験にはランダムに生成された二分木を用いた．ノード数 n の木を生成する処理を図 5.1 に示す．根ノードからランダムに左右の子ノードへの探索を繰り返し，探索する子ノードが見つからなければそこに新しいノードを生成する．ノードを生成したら根ノードから再び探索をする．これを生成したノード数が n に達するまで繰り返すことでランダム二分木を生成

表 5.1 実験機材 (16 台)

OS	Ubuntu11.10 64bit
CPU	Intel(R)Core(TM)i5 CPU3.30GHz
memory	8GB
Java	Version6
Ethernet	1GB/s
Hadoop	Version 1.0.4

```
root = 根ノードを生成;
for(i=1;i<n;i++)
   current = root, hight = 0;
   while(無限ループ)
      if(左の子を探索する場合)
         if(左の子ノードが存在しない) 左の子ノードを生成, break;
         else current = 左の子ノード, hight++;
      else
         if(右の子ノードが存在しない) 右の子ノードを生成, break;
         else current = 右の子ノード, hight++;
```

図 5.1　二分木を生成するプログラムの擬似コード

する．

5.2　台数による実行速度の変化

5.2.1　実験内容

実験は用意したリスト表現された木構造データを分割した入力データを並列プログラムで処理を実行し，その処理時間を計測することで台数の変化による処理速度の変化を調べた．処理時間は Hadoop へジョブを投入してから終了するまでの時間である．実験に使用したデータは，ランダムに生成された 10^8 ノードの木構造データをリスト表現に変換し，10^3 個に分割したデータである．入力データはすべての実験で同じものを使用した．

5.2.2　結果と考察

結果は表 5.2 と図 5.2 に示す．1 台から 16 台に台数を増やし並列処理を行うことでおよそ 3.7 倍に高速化することができた．台数増加により処理の高速化が実現した要因として，1 台のコンピュータ機器が同時に処理可能なタスク数の制限が挙げられる．実行した際の map タスクの数は 18 個，reduce タスクの数が 1 個であった．それに対して Hadoop のパラメータがデフォルトであったことから，1 台が同時に動作可能な map タスクの数は 2 個と制限されていた．そのため 16 台構成の場合を除いて実行中の map タスクが終了しなければ次のタスクが実行できない状態にあった．このことから実行時間に差がついたと考えられる．ま

表 5.2　ノード数 10^8 分割数 10^3 の入力データに対する台数毎の実行時間

台数	1	2	4	8	16
実行時間 (秒)	134	82	59	46	36

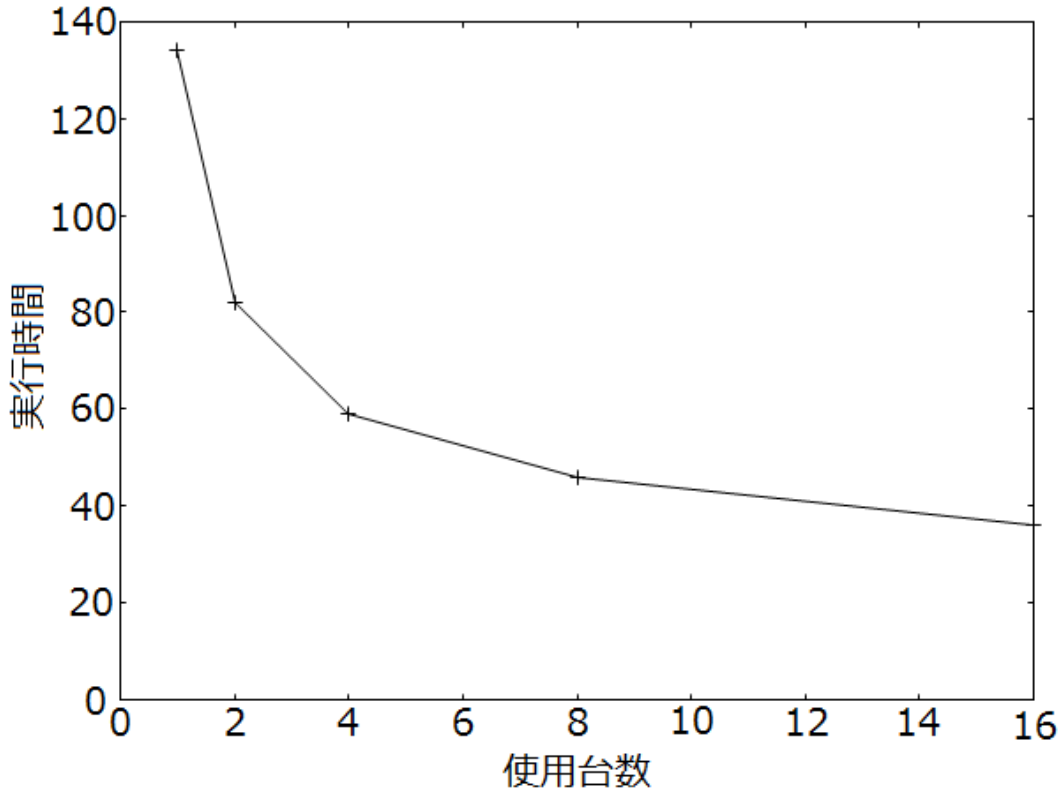

図 5.2　ノード数 10^8 分割数 10^3 の入力データに対する台数毎の実行時間

た，これ以上台数を増やしても処理できるタスクの数に変化はないため，高速化は望めないと推測できる．

5.3　分割数による実行速度の変化

5.3.1　実験内容

実験は用意したリスト表現された木構造データを分割した入力データを並列プログラムで処理を実行し，その処理時間を計測することで分割数の変化による処理速度の変化を調べた．処理時間は Hadoop へジョブを投入してから終了するまでの時間である．実験に使

表 5.3　8 台で処理した時の分割数毎の実行時間

分割数	8×10^7 ノード (秒)	4×10^7 ノード (秒)	2×10^7 ノード (秒)	10^7 ノード (秒)
25	240	65	46	41
50	95	47	38	38
10^2	48	40	37	38
10^3	38	39	37	38
10^4	35	34	33	35
10^5	34	38	37	38

用したデータは，ランダムに生成された木構造データをリスト表現に変換したデータであり，8 台で構成されたクラスターで実行した．入力データはノード数が 8×10^7，4×10^7，2×10^7，10^7 の 4 つの木構造データのものを使用し，それぞれのデータの分割数のみを変えて実験を行った．

5.3.2　結果と考察

結果は表 5.3 と図 5.3 に示す．実行時間から分割数が少なくなると処理にかかる時間が増える傾向にあることがわかる．この傾向はデータサイズが大きくなるほど顕著に出ている．このことから map タスク一つ当たりのデータが大きくなるほど処理が遅くなることがわかる．また，ノード数が少ない 2×10^7 や 10^7 などの入力データの場合は分割数が 10^5 くらい多くなりすぎた場合も処理時間が増える傾向がある．これはデータの受け渡しと reduce が処理するデータの大きさが，map 処理一つ当たりの処理時間減少量よりも多いためだと推測できる．これらの結果から分割されたリストのノード数が 10^3 前後となるように分割することが効率的であることが分かる．

5.3　分割数による実行速度の変化

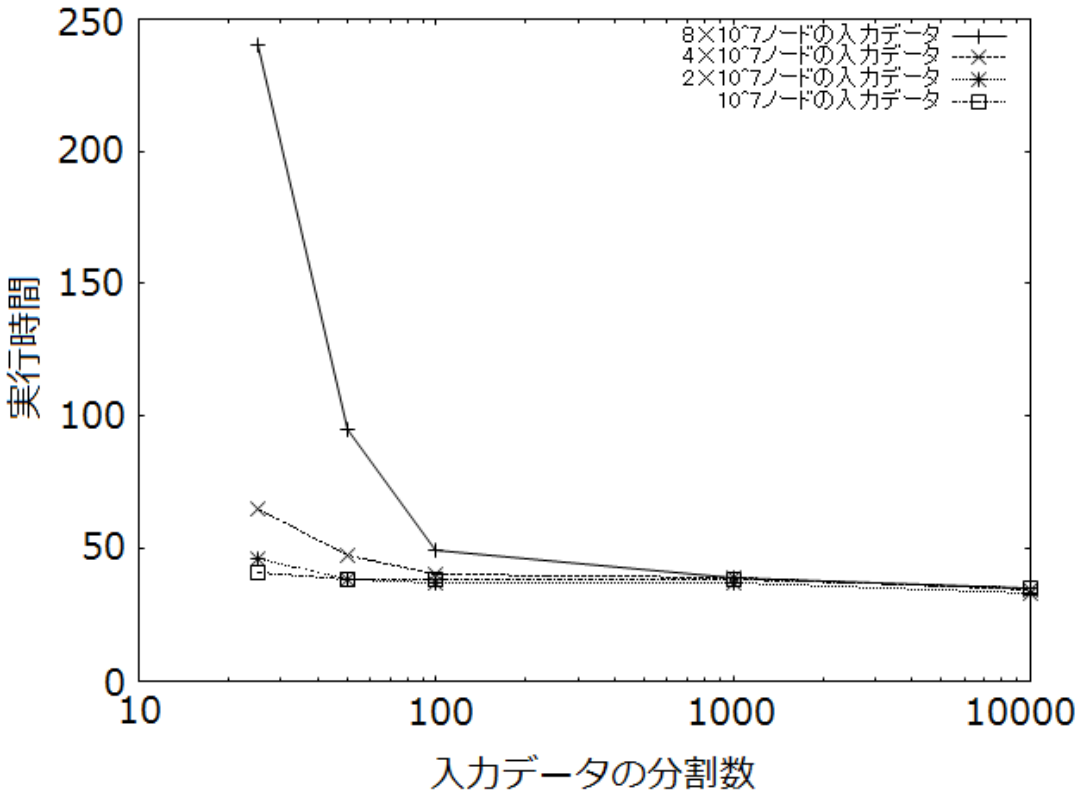

図 5.3　8 台で処理した時の分割数毎の実行時間

第 6 章

まとめと今後の課題

6.1 まとめ

本論文では，Hadoop を用いネットワークを介して複数のコンピュータ機器によるクラスターを作成し，MapReduce フレームワークによる並列な木構造データの縮約アルゴリズムを実現した．そして，並列化による効果を確かめるために 16 台のコンピュータ機器によって構成されたクラスター上で処理時間を計測する評価実験を行った．その結果，使用するコンピュータ機器を増やす事によって，処理を高速化することができた．また，並列化するために入力データを分割する際には，分割数を減らし過ぎると map による処理が大きくなり，処理時間が遅くなるという結果を得た．

6.2 今後の課題

今回の実験では分割された入力データを受けとることによって map 処理を並列に実行したが，reduce 処理では map 処理されたデータを順番に受け取り逐次処理を行うだけで並列化が行われていなかった．今後は reduce 処理も含めて並列化を可能にしたい．また木構造データの縮約として maxPathSum を求めるプログラムを作成したが，これらの計算パターンを応用して同じ木構造データを持つ XML データベースに対するクエリ処理などに活用したい．

謝辞

本研究を行う際に, ご指導いただいた松崎公紀准教授に深く感謝いたします．また，研究中に様々な意見，感想をいただいた松崎研究室のメンバーに感謝致します.

有難うございました.

参考文献

[1] Serge Abiteboul, Peter Buneman, Dan Suciu. 横田 一正 監訳, 國島 丈生 訳. XML データベース入門. 共立出版株式会社, 東京. 2006.

[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Operating Systems Design and Implementation*, pp. 137–150, 2004.

[3] Robert Endre Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium on Foundations of Computer Science, 1984.*, pp. 12–20, October 1984.

[4] ウィリアム・グロップ, ユーイング・ラスク, ラジーブ・タークル 著. 畑崎 隆雄 訳. 実践 MPI-2. 株式会社ピアソン・エデュケーション, 東京, 2009.

[5] 井町 宏人. 準構造化データ処理の効率的 MapReduce 実装に関する研究. 東京大学大学院情報理工学系研究科数理情報学専攻. 2012 年.

[6] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient parallel tree reductions on distributed memory environments. In *Proceedings of the 7th international conference on Computational Science, Part II*, ICCS '07, pp. 601–608, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Gary Lee Miller and John Henry Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pp. 478–489, Portland, Oregon, October 1985. IEEE.

[8] 小高 知宏 監修, 高橋 勇, 小倉 久和 共著. 演習中心 UNIX 入門. 森北出版株式会社. 2004.

[9] Peter S.Pacheco 著. 秋葉 博 訳. MPI 並列プログラミング. 培風館, 東京, 2001.

[10] Tom White 著. 玉川 竜司, 兼田 聖士 訳. *Hadoop*. 株式会社オライリー・ジャパン, 東京, 2010.

付録 A

実験で用いた Hadoop プログラムの一部

```
import java.util.*;
import java.io.*;
import java.lang.*;
import org.apache.hadoop.util.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.JobStatus;

public class MRSublist{
  public static void main(String args[])throws Exception{
    Configuration conf = new Configuration();
    Job job = new Job(conf,"mapreducesublist");
    job.setJarByClass(MRSublist.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setOutputKeyClass(LongWritable.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.setInputPaths(job,new Path(args[0]));
    FileOutputFormat.setOutputPath(job,new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
class Map extends Mapper<LongWritable,Text,LongWritable,Text>{
  protected void map(LongWritable key,Text value,Context context)throws
IOException,InterruptedException{
    Text hnfvalue = new Text();
    LongWritable finalkey = new LongWritable(1);
    String sublistline = value.toString();
    String subliststring[] = sublistline.split(",");
    int sls = subliststring.length;
    String sublistnum[][] = new String[sls-1][];
```

```
    for(int i = 1;i<sls;i++)
      sublistnum[i-1] = subliststring[i].split(":");
    List sublist[] = new List[sls];
    for(int i = 0;i<sls-1;i++)
      sublist[i] = new List(Integer.parseInt(sublistnum[i][0]),Integer.parseInt
(sublistnum[i][1]));
    hnfvalue = Hnftext(Integer.parseInt(subliststring[0]),
Hillnormalform(sublist,sls-1));
    context.write(finalkey,new Text(hnfvalue));
  }
  public Hnf Hillnormalform(List[] list,int length){
    Hnf hnf = new Hnf(list);
    int a,b,c,t;
    hnf.setd(list[0].gethight());
    if(list[0].getvalue() == -1){
      hnf.cspush(-1);
      hnf.cspush(-1);
    }
    else{
      hnf.cspush(-1);
      hnf.aspush(list[0].getvalue());
      hnf.bspush(-1);
    }
    for(int i=1;i<length;i++){
      if(list[i].getvalue() != -1){
        hnf.aspush(list[i].getvalue());
        hnf.bspush(-1);
      }
      else{
        if(hnf.asempty()){
          hnf.cspush(-1);
          hnf.setd(list[i].gethight());
        }
        else{
          a = hnf.aspop();
          b = hnf.bspop();
          if(b == -1)
            t = a;
          else
            t = a + b;
          if(hnf.bsempty()){
            c = hnf.cspop();
            if(c < t)
              c = t;
              hnf.cspush(c);
            }
            else{
              b = hnf.bspop();
              if(b < t)
              b = t;
              hnf.bspush(b);
            }
          }
        }
      }
    return hnf;
  }
}
```

```
class Reduce extends Reducer<LongWritable,Text,LongWritable,Text>{
  protected void reduce(LongWritable key,Iterable<Text> values,
Context context)throws IOException,InterruptedException{
    LongWritable finalkey = new LongWritable(1);
    String th[];
    Hnf hnf[] = new Hnf[9999999];
    int count = 0;
    Hnf hnfx = new Hnf();
    Hnf hnfy = new Hnf();
    Hnf hnfz = new Hnf();
    int n,m,v;
    for(Text value : values){
      th = value.toString().split(":");
      hnf[Integer.parseInt(th[0])] = Texthnf(th);
      while((hnf[count] != null)){
        if(count==0)
          hnfx = hnf[0];
        else{
          hnfy = hnf[count];
          hnfz.clear();
          if(hnfx.getd() < hnfy.getd()){
            hnfz.setd(hnfx.getd());
            n = hnfy.cspoint();
            m = hnfy.aspoint();
            for(int i=0;i<hnfx.aspoint()-n+1;i++)
              hnfz.aspush(hnfx.asget(i));
            for(int i=0;i<m;i++)
              hnfz.aspush(hnfy.asget(i));
            for(int i=0;i<hnfx.aspoint()-n;i++)
              hnfz.bspush(hnfx.bsget(i));
            v = Merge(hnfx,hnfy,n-1);
            if(hnfx.bsget(hnfx.bspoint()-n) < v)
              if(hnfy.csget(n-1) > v)
                hnfz.bspush(hnfy.csget(n-1));
              else
                hnfz.bspush(v);
            else
              if(hnfx.bsget(hnfx.bspoint()-n) < hnfy.csget(n-1))
                hnfz.bspush(hnfy.csget(n-1));
              else
                hnfz.bspush(hnfx.bsget(hnfx.bspoint()-n));
            for(int i=0;i<m;i++)
              hnfz.bspush(hnfy.bsget(i));
            for(int i=0;i<hnfx.cspoint();i++)
              hnfz.cspush(hnfx.csget(i));
            }
            else{
              hnfz.setd(hnfy.getd());
              n = hnfx.aspoint()+1;
              m = hnfx.cspoint();
              for(int i=0;i<hnfy.aspoint();i++)
                hnfz.aspush(hnfy.asget(i));
              for(int i=0;i<hnfy.bspoint();i++)
                hnfz.bspush(hnfy.bsget(i));
              for(int i=0;i<m-1;i++)
                hnfz.cspush(hnfx.csget(i));
```

```
              v = Merge(hnfx,hnfy,n-1);
              if(hnfx.csget(m-1) < v)
                if(hnfy.csget(hnfy.cspoint()-n) > v)
                  hnfz.cspush(hnfy.csget(hnfy.cspoint()-n));
                else
                  hnfz.cspush(v);
              else
                if(hnfx.csget(m-1) < hnfy.csget(hnfy.cspoint()-n))
                  hnfz.cspush(hnfx.csget(hnfy.cspoint()-n));
                else
                  hnfz.cspush(m-1);
              for(int i=0;i<hnfy.cspoint()-n;i++)
                hnfz.cspush(hnfy.csget(n+i));
              }
            hnfx.clear();
            hnfx.copy(hnfz);
          }
        hnf[count] = null;
        count++;
      }
    }
    context.write(finalkey,new Text(""+hnfx.cspop()));
  }
  public int Merge(Hnf hnf1,Hnf hnf2,int n){
    if(n==0)
      return -1;
    int x = Merge(hnf1,hnf2,n-1);
    if(hnf1.bsget(hnf1.bspoint()-n) < x)
      if(hnf2.csget(n-1) > x)
        x = hnf2.csget(n-1);
    else
      if(hnf1.bsget(hnf1.bspoint()-n) < hnf2.csget(n-1))
        x = hnf2.csget(n-1);
      else
        x = hnf1.bsget(hnf1.bspoint()-n);
    if(x != -1)
        x += hnf1.asget(hnf1.aspoint()-n);
    else
      x = hnf1.asget(hnf1.aspoint()-n);
    return x;
  }
}
```