平成 23 年度

学士学位論文

MapReduce フレームワークを用いた木構造処理の実現に関する研究

Realizing Tree-Structure Processing on MapReduce Framework

1120232　　川村　高之

指導教員　　松崎　公紀

2012 年 3 月 1 日

高知工科大学 情報システム工学科

要 旨

MapReduce フレームワークを用いた
木構造処理の実現に関する研究

川村　高之

木構造は，探索アルゴリズムやファイルシステム等でよく用いられている．近年では，XML 文書や HTML 文書といった木構造を利用したデータの大規模化が進んでいる．これにより，一台の PC では処理が終わらない，データが扱えないといった問題が発生し，処理の高速化が課題となっている．そこで，並列化することが重要となる．しかし，並列計算プログラムの作成は，逐次プログラムに比べて難しい．特に，木構造に対しては，ノードの親子・兄弟関係にも気を配らなければならない．よって，容易に木構造に対する並列プログラミングを行う方法が必要である．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 フレームワークを用いて木構造処理を実現する．そして，高速化が可能であるのか，どの程度効果があるのかを検証した．実験は，実際に実装したプログラムを 8 台構成のクラスタを用いて実行して処理時間を計測した．結果は，PC 台数を増やすことで，処理を高速化する事ができた．しかし，6 台以降は並列化の効果が薄くなっていた．原因は，ネットワークのオーバヘッドの増加と Hadoop のパラメータ設定のチューニング不足である．

キーワード　MapReduce 木構造

Abstract

Realizing Tree-Structure Processing on MapReduce Framework

Kawamura Takayuki

Tree structure are often used in large-scale data management and search algorithm etc. In recent years, has gotten bigger is data using tree structure such as XML and HTML documents. Problem occurs such as single PC can not end the process and handle data. Processing speed has question. Therefore, Parallelization is important. However, the creation of parallel programs is more difficult compared to the sequential program. In particular, the tree structure, have to care brother and parent-child relationship of node. Therefore, need an easy way to parallel programming for the tree structure. In this paper, realization of processing tree structure using the MapReduce framework, which is a parallel computing framework for large-scale clusters. And to verify it is possible to speed up, and how effective. Experiment, processing time is performed using cluster of 8 nodes implemented program. The result is by increasing the number of PC, able to speed up the process. But, 6nodes later is thinner of parallel effect. Cause is the lack of tuning of Hadoop configuration parameters and an increase in network overhead.

key words MapReduce Tree structure

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，探索アルゴリズム [1] やファイルシステム [2] 等でよく用いられている．また，近年技術が進歩するにつれて PC で扱われるデータの容量が肥大化し，XML 文書や HTML 文書といった木構造を利用したデータでも大規模化が進んでいる．これにより，1 台の PC では処理が終わらない，データが扱えないといった問題が発生する．よって処理の高速化が課題となっている．しかし，現状の CPU の処理速度が頭打ちになりつつあり，CPU を変更した程度では処理の高速化が難しい．そこで，複数 PC を用いての並列処理が重要となる．しかし，並列計算は一般的に，ネットワークの知識や，処理の方法，負荷分散等に気をつけなければならないためプログラム作成が難しい [3]．それらの高度な知識が無くても並列計算を行える方法が必要である．特に，木構造に対しては，逐次処理では気をつけなくても良い木構造の持つ兄弟・親子の依存関係にも気を配らなければ正しい処理が行えない．

高度な知識が無くても並列計算プログラムを構築することができる方法として，MapReduce フレームワーク [4] を利用する方法が挙げられる．MapReduce フレームワークでは，ネットワークを利用した複数台の PC 管理や，負荷分散についてフレームワークが自動で行ってくれる．そのため，プログラマは各 PC で動かす処理の方法を MapReduce プログラミングモデルに合わせて記述するだけで並列処理ができるプログラムを構築することができる．しかし，MapReduce プログラミングモデルはアクセスログの解析や検索エンジンのインデックスデータ作成等のリアルタイムな処理やデータの依存性が無い処理を得意としており，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．よって，MapReduce プ

ログラミングモデルを用いて依存性のある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，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，高速化が可能であるのか，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

1.2.1 MapReduce

データが分割されており，処理が MapReduce プログラミングモデルに適用可能なら MapReduce フレームワークを用いて並列化する事が可能である．近年，CPU の処理速度が頭打ちになりつつある現状で，CPU よりも浮動小数点演算の効率がよい GPU で汎用計算を行う GPGPU[10] が研究されている．GPU は条件分岐を行うとオーバーヘッドが発生し，処理効率が落ちる等問題があるためプログラミングが難しい．また，GPU での MapReduce フレームワークの実装を行う研究もされている [8]．そこで GPU を用いた MapReduce を適用することで GPU プログラミングの難しさを隠蔽し，容易に GPU を用いた並列計算を可能としている．論文 [8] では，フレームワークの実装に CUDA を用いて，結果として CPU のみの場合より 32 倍から 150 倍のスピードアップを達成している．

1.2.2 木に対する並列計算

本論文では，既存の MapReduce フレームワークを用いた木構造処理の実現について述べているが，木構造に対して並列処理を行う研究はこれまでにもいくつか行われている．そのうちの一つに，木構造上の処理のための MapReduce 型フレームワークを C++と MPI で実装する方法がある [9]．この方法では，結果として最大 16 倍の高速化を達成している．そして，他の方法にスケルトン並列プログラミングの木スケルトンを用いる方法がある [7]．スケルトン並列プログラミングは，並列スケルトンと呼ばれるパッケージ化された並列計算を基本単位として組み合わせる事で並列プログラミングを構成する方法である．これにより

プログラマは同期処理等を意識せずプログラムを考える事ができる．木スケルトンはスケルトン並列プログラミングの中に含まれる並列スケルトンの一つである．

1.3　本論文の構成

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，本論文で用いる木構造の分割方法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 MapReduce に適用するのかを説明を行う．第 5 章では，木構造を並列化することによってどの程度効果があるのかを検証する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 フレームワークと MapReduce プログラミングモデルについて述べる．

2.1 プログラミングモデルとしての MapReduce

MapReduce は Google によって提案されたプログラミングモデルである [4]．処理としては，キー/値のペアのセットを入力として受け取り，出力のキー/値のペアのセットを生成するというものである．この処理をユーザは map と reduce という二つの関数で表現する．

map は，入力のペアを取得し，入力されたデータになんらかの処理を行い reduce に渡すための中間データのキー/値のペアのセットを生成する．

reduce は，反復子を用いて，中間データのキー/値のペアのセットを順番に受け入れ，呼び出し毎に，一般的に 0 か一つの出力を生成する．

全体の処理の流れの例を図 2.1 に示す．プログラム例としては，WordCount がある．map と reduce の擬似コードを図 2.2 に示す．入力データは複数行に渡り英文がかかれたテキストデータを一行毎に分割した物です．図 2.2 を例に挙げると，map には Inputdata であるテキストデータを一行毎に分割した物がキーと共に受け渡され，キー/値の reduce で処理を行う形に変換して出力する．reduce には複数の map からの出力が受け渡され，キー毎に値を集約する．そして，最終的な出力は各 reduce から出力される．

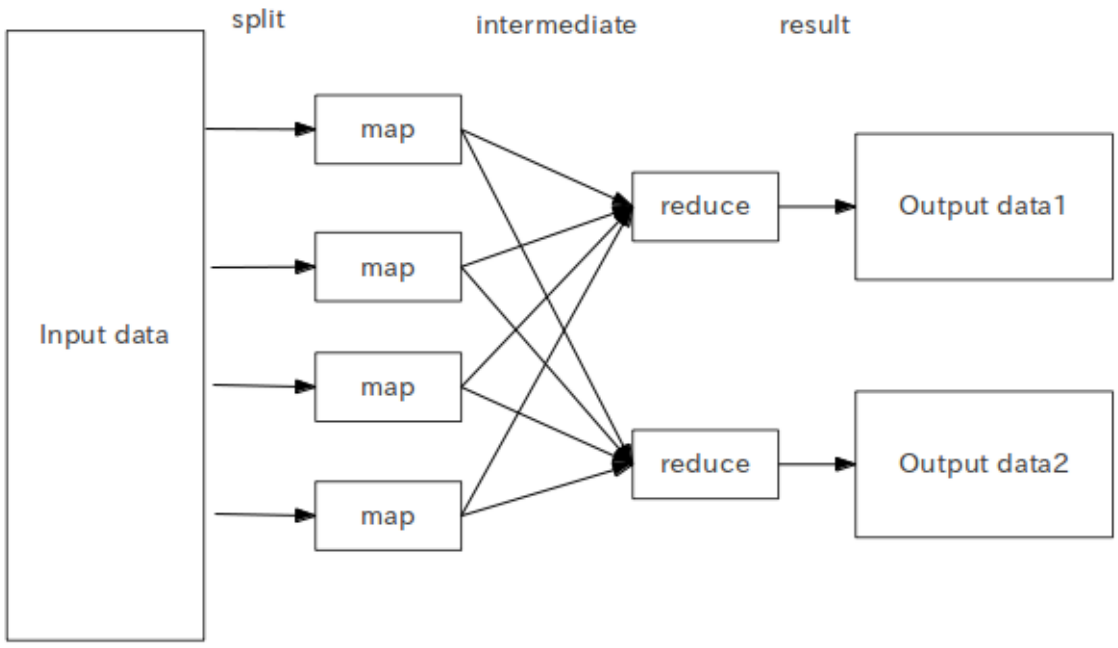

図 2.1　MapReduce Programming Model

2.2　フレームワークとしての MapReduce

Google の MapReduce は，大規模なデータに対して効率的に並列処理を行うための分散処理フレームワークの事である．また，MapReduce フレームワーク実装の一つである Hadoop[6] は，Google の MapReduce フレームワークのオープンソース実装である．本節では，MapReduce フレームワークを構成する MapReduce と分散ファイルシステムについて述べる．

2.2.1　MapReduce

MapReduce は，分散データ処理モデルおよびコモディティマシンで構成される大規模クラスタ上の実行環境である．

MapReduce は節 2.1 で述べた事と同様に，map と reduce という 2 つのフェーズに分割して処理を行なわれる．各フェーズの入出力はキーと値のペアの集合である．一般的な処理の流れとしては，map で渡されたデータに対してなんらかの処理を行い，reduce で map

```
map(long key, String line){
   String[] word=lineを空白で分割;
   int i=0;
   while(!(word[i]==null)){
            key = word[i];
      reduceへの出力(key,1);
      i++;
   }
}
reduce(long key, Iterable<String> values){
   int sum=0;
   for(value : values){
      sum +=value;
   }
   keyごとの出力(key, sum);
}
```

図 2.2　プログラム例:WordCount

から渡されたデータを集約して出力するという形になる．その際，map フェーズと reduce フェーズで行う処理に関してはプログラマ自ら記述する．各フェーズの入出力データの受け渡しは，フレームワークが自動で行なうので，プログラマは各フェーズの入力されてから出力されるまでのみを記述するだけでクラスタ上で行われる並列処理を行うことができる．

2.2.2　分散ファイルシステム

分散ファイルシステムは，コモディティマシンで構成される大規模クラスタ上の分散ファイルシステムである．

並列処理を行わなければならないようなデータは，データの容量が数百メガバイトから数ギガバイト，あるいは数テラバイトの非常に大規模なデータなることがある．1TB を越えるようなデータを処理する場合，データ容量が一台のマシンのストレージ容量を越えてしまう事がある．この場合，処理するデータを複数のマシン上に分散して配置する必要がある．この時用いられるのが分散ファイルシステムである．分散ファイルシステムはネットワークを介して複数のマシンを管理する必要があるため，通常のファイルシステムに比べて処理が煩

雑になる．しかし，Google の MapReduce フレームワークには GFS という分散ファイルシステムがある他，前述した Hadoop には，HDFS という分散ファイルシステムがあるため，それらの用意された分散ファイルシステムを用いる事で大規模データを扱う事ができる．

第 3 章

木構造の分割

並列計算は複数マシンで処理を分散する事で高速に処理を行う手法である．しかし，並列計算を行うには，用意するデータが分割された形でなければならない．本章では，本論文で扱うデータ構造である木構造を分割する方法である m-bridge について述べる．

m-bridge[5] とは，グラフ理論における根付き木を分割する方法の事である．木構造を用いる処理は木の内部ノードの依存関係や木の形を利用するものがある．木を分割した場合，内部ノードの依存関係の崩壊，木の形が不均衡になることで処理が行えなくなる場合がある．しかし，m-bridge は木を分割した状態から元の木を復元する事ができるため依存関係や木の形を維持したまま木を分割する事ができる．

m-bridge を施した後の木の性質としては，以下の 3 つが挙げられる．

- 分割した状態から分割する前の木を復元可能
- 分割した木が含むノード数は m 個以下
- ノード数を N とした場合に，分割木数=$(\frac{2N}{m} - 1) * 2 + 1$ 以下

Algorithm1 (m-bridge による木の分割)

1. すべてのノードで自分自身と自分の子孫の数を足した数を数える
2. 葉でない内部ノードの事を v，v の左右の子をそれぞれ l，r，ノード x に対して $x.size$ を x 以下のノード数，m は分割時に与えるとする．この場合に，深さ優先探索で以下の二つの数式が両方成立したノードを探し，マークをつける
3. マークを付けたノードで分割する

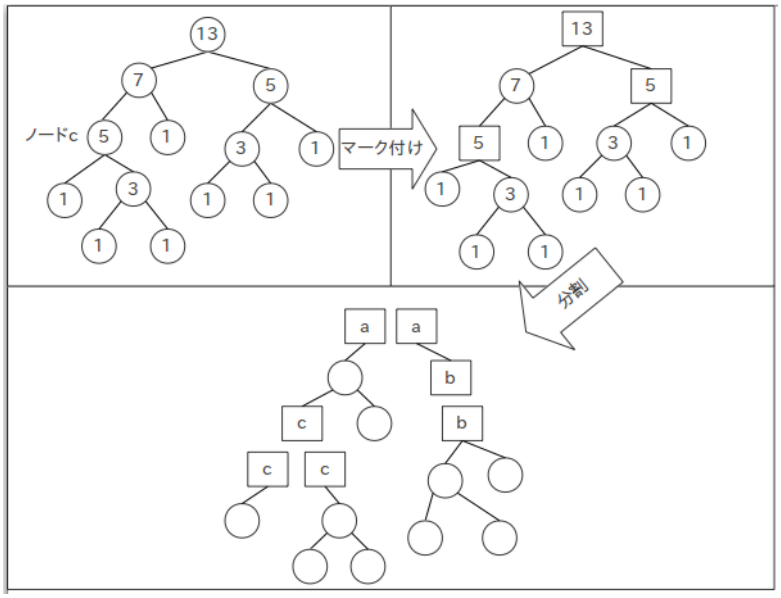

図 3.1　m-bridge の例

$$\left\lceil \frac{v.size}{m} \right\rceil > \left\lceil \frac{l.size}{m} \right\rceil$$
$$\left\lceil \frac{v.size}{m} \right\rceil > \left\lceil \frac{r.size}{m} \right\rceil$$

分割されている状態の木からの復元は，マークがついているノードをつなぎ合わせる事で元の木を得る事ができる．

図 3.1 に全二分木に m=4 とした場合の m-bridge の処理の例を挙げる．図中の◯はノード，□はマーク付きノードを示している．ノード内の数字はそのノード以下のノード数，アルファベットは同じノードということを示している．例えば，ノード c を数式に当てはめると m が 4，$v.size$ が 5，$l.size$ が 1，$r.size$ が 3 となり，数式を両方満たすためマークを付け，分割される．

第 4 章

木構造処理への MapReduce の適用

本章では，どのように木構造データと MapReduce を組み合わせるのかを記述する．MapReduce には MapReduce のオープンソース実装である Hadoop を用いた．

木構造処理へ MapReduce を適用する場合，map では依存性に関係無い部分までの計算を行い，reduce に依存性がある部分の処理を任せる形になる．木構造データと Hadoop を組み合わせる方法を，評価実験に用いる木の高さを求める計算を例に挙げて述べる．木構造データと Hadoop を組み合わせる方法は，以下の様になる．

1. 木構造データを m-bridge を用いて分割
2. 分割したデータを Hadoop に入力
3. map フェーズで各マシンで Hadoop から自動で割り当てられた分割木毎の木の高さを求め，reduce に結果を受け渡す
4. reduce フェーズで map から受け渡された結果から分割する前の元の木の高さを求め出力

1. を前処理，3. を map，4. を reduce としてプログラムを実装した．

4.1 前処理

木構造データに m-bridge をかけて分割して Hadoop に投入するデータを出力するまでを前処理とした．前処理は MapReduce フレームワークを用いずに実装し，並列処理ではなく逐次処理で行った．

4.1.1 m-bridge

3 章でも m-bridge について述べたが，そのままでは実装する際にマーク付きノードの扱いで困るため，実装を容易にするために木を分割する場所をマーク付きノードからマーク付きノードの直下に変更した．そして，マーク付きノードの扱いの変更に伴い前述の方法では元の木を得る事ができなくなるため，木の復元方法も変更した．木を復元出きるようにするために，分割木に識別子を割り振り，その識別子を用いて分割木を繋いだ全体の構成を表す構成木を生成するように変更した．ここまで説明してきた m-bridge の変更点を図 4.1 に示す．そして，変更した事を踏まえて m-bridge を実装した．

4.1.2 データの出力

前処理の段階で出力するファイルは木構造データに m-bridge をかけて得た分割木と全体の構成を表す構成木の二つである．出力の形は Hadoop 側で読みやすいようにテキストファイルに一行に一つの分割木を記述する形とした．この時出力した構成木のファイルは Hadoop にジョブを実行させる際に，処理を行うすべてのノードに一度だけ配布され，map と reduce から利用できるようにした．出力ファイルの例を図 4.2 に示す．行頭に付いている番号は各分割木を識別するための番号である．

4.2 map

評価実験に用いる木の高さを求める計算を例に上げて map で行う処理を解説する．map では複数のノードで分散して処理を行う部分を記述した．処理の流れを以下に示す．入力は

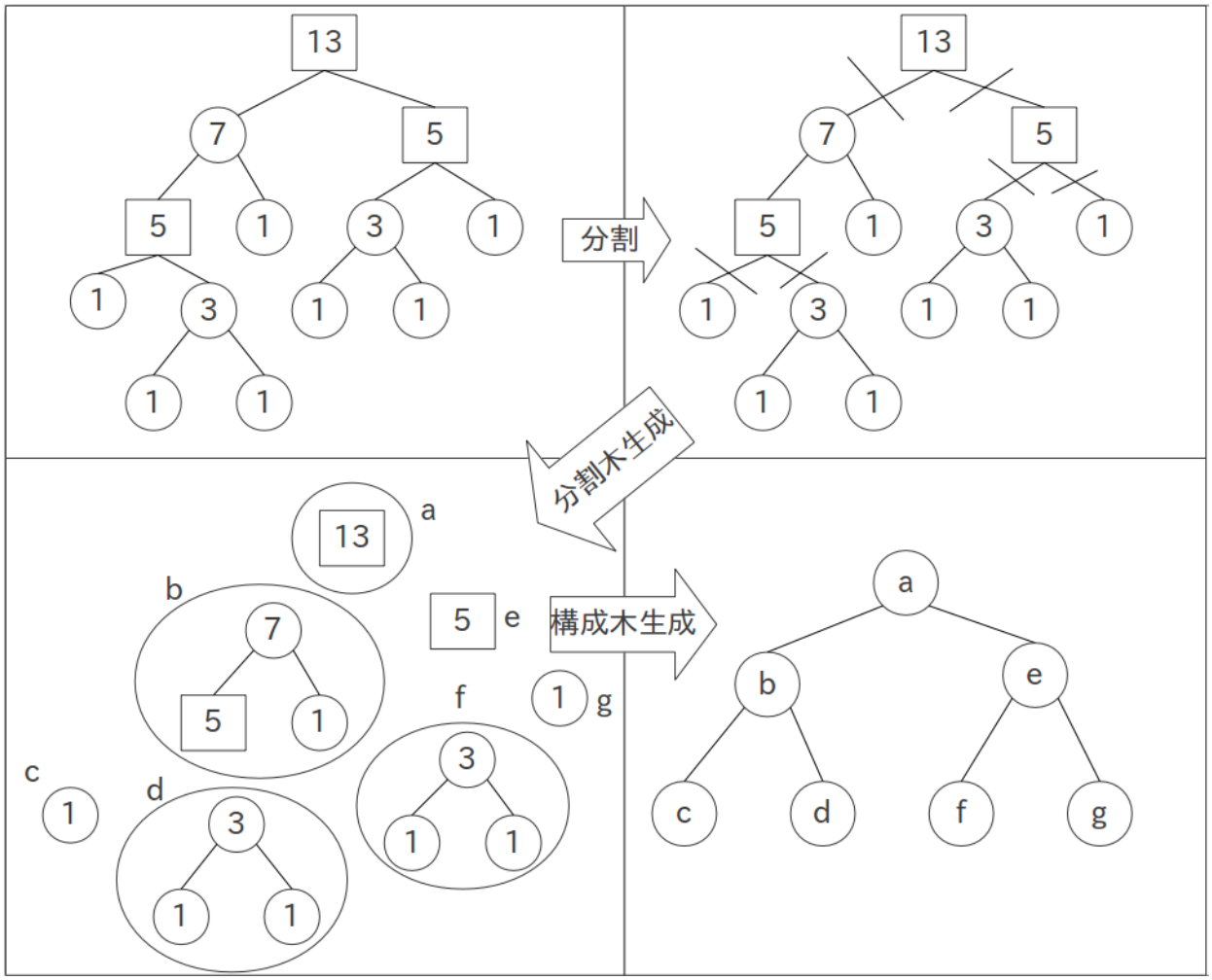

図 4.1　変更後の m-bridge の例

分割木がかかれたファイルから Hadoop が一行づつ抜き出したものを取る．

1. 入力から分割木を再構成
2. 分割木の高さを計算
3. 求めた高さを Reduce に送る

2. の分割木の高さの計算を行う際に二通りの高さを計算する必要がある．それは，分割木の最大の高さと分割木の根からマーク付きノードまでの深さの二つである．木の高さを計算する場合，根から一番深い葉までの高さを調べれば良い．しかし，分割木の場合，図 4.3 に示す根がノード d の分割木はマーク付きノードまでの深さが 2，分割木の高さが 3 だがマー

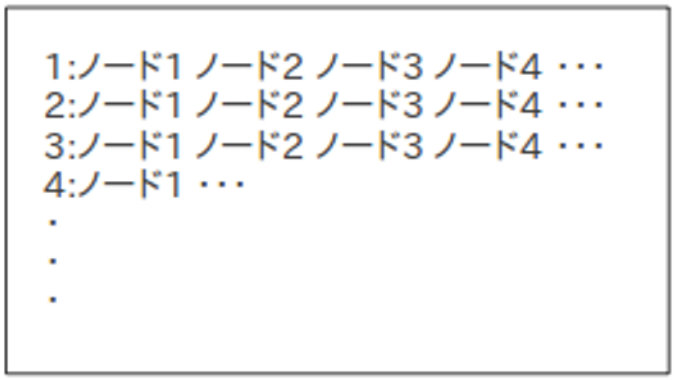

図 4.2　出力ファイルの例

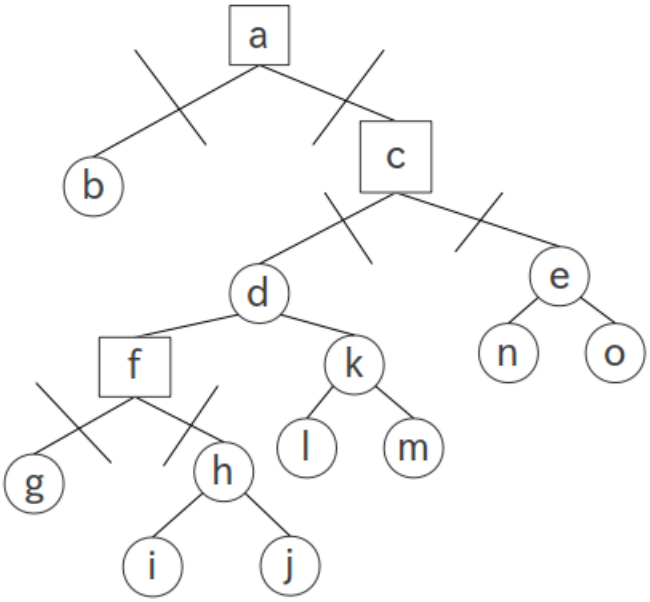

図 4.3　分割木の深さ例 (m=4 で分割した場合，□がマーク付きノード，○が内部ノードを表す)

ク付きノードの下にさらに高さ 2 の分割木がつながっている．この場合，分割木の高さが木全体で見た際の高さとは関係無い．この場合分割木の根からマーク付きノードまでの深さがあれば，reduce でより下位の分割木の深さと合わせて最大の高さを計算する事ができる．根がノード d の分割木に map を実行した結果は，出力の形が“ 分割木の識別番号 : 木の高さ , マーク付きノードまでの深さ ”とすると，4 : 3 , 2 となる．

4.3 reduce

map と同様に reduce で行う処理を解説する．reduce ではすべての map から送られてくる入力から，構成木データを用いて依存性を解決しながら処理を行う部分を記述した．処理の流れを以下に示す．入力はすべて map から送られてきたデータを反復子を用いて順番に取り出すことによって得る．map とは違い reduce は依存性の解決を担うため reduce タスクは一つに限定される．

1. map から送られてきたデータを反復子を用いて一つ受け取り，識別子を key，高さのデータを値として hashmap に格納
2. 1. を map から渡されるデータがなくなるまで繰り返す
3. 構成木データから構成木を構築
4. hashmap と構成木を用いて木全体の高さの最大値を計算
5. 結果を出力

$maxh$ を分割木の最大の高さ，$markh$ を分割木の根からマーク付きノードまでの深さとした場合の 4. の木全体の高さの最大値を計算する計算式を以下に示す．図 4.3 を用いて説明する．図 4.3 の木は高さが 6 である．これは木を a, c, d, f, h, j と通った場合である．処理は以下のようになる．

1. 根ノード a から構成木を葉ノードまで下記の数式の $result$ から L,R を用いて再起的に探索
2. 葉に到達するとその分割木の $maxh$ と $markh$ の大きい方を返す
   ノード h の分割木なら 2
3. 葉以外かつ根ではない内部ノードは左右の下位ノードから返された値の大きい方と $markh$ を足した値と自身の $maxh$ の大きい方を返す
   ノード d の分割木なら g から 1, h から 2 が返るので，$max(d$ の $markh + max(1,2), d$ の $maxh)$ で 4 を返す

4. 根まで戻ったら結果を出力

$$result = \begin{cases} max(L, R) + markh & (otherwise) \\ max(maxh, markh) & (leafnode) \end{cases}$$

$$L = \begin{cases} max(maxh, markh) & (leafnode) \\ max(L, R) + markh & (otherwise) \end{cases}$$

$$R = \begin{cases} max(maxh, markh) & (leafnode) \\ max(L, R) + markh & (otherwise) \end{cases}$$

第 5 章

評価実験

本章では，実際に木構造に対して MapReduce を適用して高速化することができたのか，どの程度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．なお，Hadoop の設定は mapred.child.java.opts=-Xmx1024 以外はデフォルトである．実験に用いたプログラムは付録 A に示す.

5.1 m による処理時間の変化

5.1.1 実験内容

実験内容は，用意したデータを木の高さを求める並列プログラムで処理し，その処理時間を計測することで m による処理速度の変化を調べた．処理時間は 5 回計測を行いその平均を結果とした．処理時間は Hadoop にジョブを投入してから終了するまでの時間であり Hadoop による各 PC へのデータの配布や map や reduce 以外の処理の時間も含んでいる．

実験に用いたデータは，ランダムに生成した 100,000,001 ノードの全二分木を m=100 から 1,000,000 までで分割したデータである．ファイルの大きさはそれぞれ約 1.5GB で，全二分木のランダム生成は，図 5.1 に示すように，次の 1. から 5. までをノードの数が設定した数になるまで繰り返すことで行った．全二分木のランダム生成の擬似コードを図 5.2 に示す．擬似コードの for 文で 2. から 5. を while 文で 4. から 5. をループしており，最初の if 文で 2. と 3. と 5. を二つめの if 文が 4. を表している．

1. 根を作成

5.1 m による処理時間の変化

表 5.1 実験機材 (8 台)

OS	Ubuntu11.10 32bit
CPU	Intel(R) Core(TM)i5 CPU2.80GHz
memory	2GB
Java	Version6
Ethernet	1GB/s
Hadoop	Version 0.20.203.0

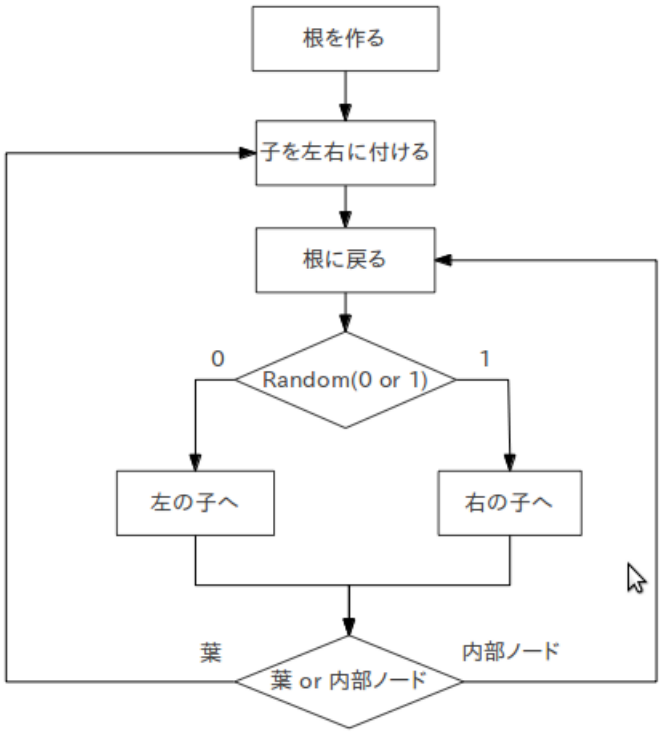

図 5.1 全二分木のランダム生成

2. 左右の子を作成
3. 根に戻る
4. ランダムで左右の子へ
5. 葉なら 2. へ，そうでないなら 3. へ

```
RandomCreateBinaryTree(ノード数n){
  Node root=根ノード作成;
  Node current;
  int hight=0
  int i=0;
  for(i=1;n>i;i=i+2){
    current=root;
    hight=1;
    while(true){
      If(葉かどうか){
        左右の子を作成;
        currentに左右の子を繋ぐ;
        左右の子の親をcurrentに;
        break;
      }
      if(random(0 or 1)==0){
        current=currentの右の子;
        hight++;
      }else{
        current=currentの左の子;
        hight++;
      }
    }
  }
  return root;
}
```

図 5.2　全二分木のランダム生成の擬似コード

5.1.2　結果と考察

結果において処理速度の変化が大きかった m の分割木に含まれるノード数のヒストグラムを図 5.4，図 5.5，図 5.6，図 5.7，図 5.8 に示す．各図のノード数 1 の分割木の数は，1,125,067 個，131,071 個，16,383 個，255 個，127 個である．図 5.3 から 100,000,001 ノードの木に対して m=1,000 から 10,000 で分割するのが一番効果が高かった．図 5.4 と図 5.5 から m=100 から m=1,000 までは，ノード数 1 の分割木の数が大きく減ったため高速化したとわかる．図 5.7 と図 5.8 を見ると，ノード数の少ない分割木の数が約半分になっている

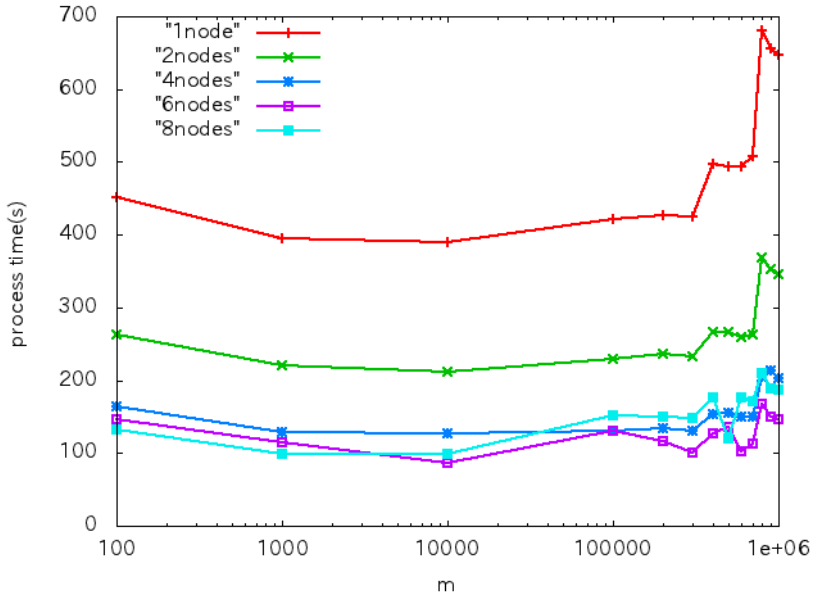

図 5.3　台数効果

のに対して，分割木が含むノードの数は違うが，大量のノードを抱えている分割木が多くなっている．m=700,000 と m=800,000 の間で処理速度が大きく増えたのは大量のノードを抱える分割木が増えたことで map タスクで行う処理が増えたからだとわかる．これらのことから，m-bridge を行う場合，m は分割木の数を多くしすぎず，分割木を大きくしすぎないように設定しなければならないと分かる．

5.2　台数効果

5.2.1　実験内容

実験内容は，用意したデータを PC 台数を変えながら木の高さを求める並列プログラムで処理し，その処理時間を計測することで台数効果を調べた．処理時間は 5 回計測を行いその平均を結果とした．処理時間は Hadoop にジョブを投入してから終了するまでの時間であり Hadoop による各 PC へのデータの配布や map や reduce 以外の処理の時間も含んでいる．

実験に用いたデータは，m による処理時間の変化を調べた物と同様である．

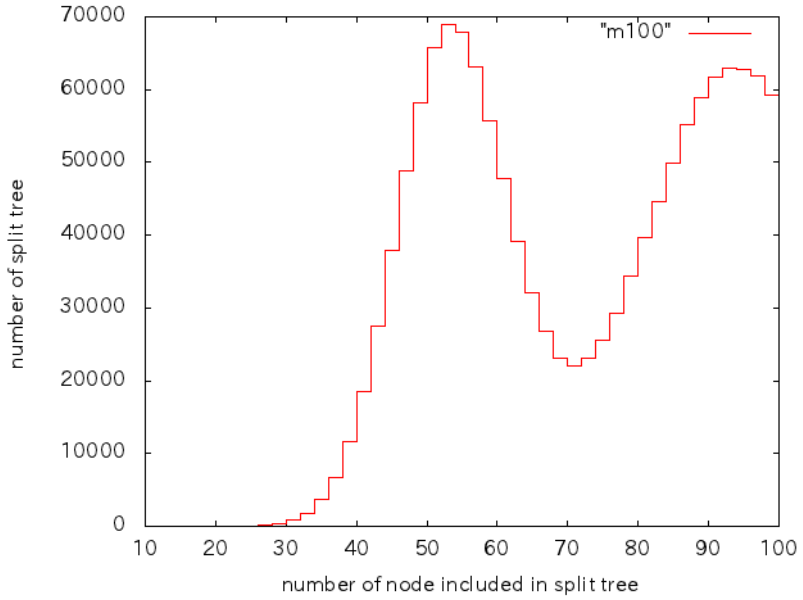

図 5.4　分割木に含まれるノード数のヒストグラム (m=100)

5.2.2　結果と考察

結果を図 5.3 に示す．実験中のすべてのデータで map タスクの数は 24 個，reduce タスクの数は 1 個だった．PC 台数を増やし，並列処理を行うことで処理を高速化する事ができた．しかし，4 台以降は台数を増やしても効果が薄く 8 台では 6 台よりも若干処理が遅いという結果となった．これは，ネットワークのオーバヘッドが PC 台数が増えることによって増加したためだとわかる．他の要因としては，map タスクが 24 個実行されている事が挙げられる．Hadoop のパラメータがデフォルトのため，各ノードで同時に動作する map タスクの数が 2 個に制限されており，8 台構成のクラスタでは同時に 16 個の map タスクしか処理できない．しかし，今回のプログラムでは map タスク 24 個実行されている．これでは，map タスク 8 個が別の map タスクが終了した後でなければ実行する事ができない．map タスク数は，データのファイルサイズが約 1.5GB で HDFS のブロックサイズが 64MB，Hadoop の map タスクは通常一度に一つのブロックを処理するので 24 個となる．よって，これは HDFS のブロックサイズを増やすことで実行する map タスクの数を減らし 16 にすることで解決できる．このように Hadoop のパラメータをチューニングすれば今以上に処理を高速

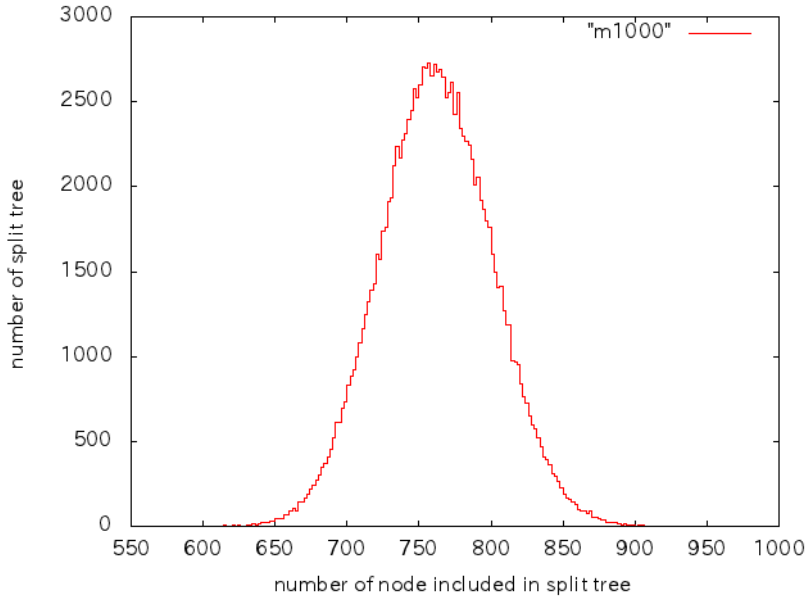

図 5.5　分割木に含まれるノード数のヒストグラム (m=1000)

化することができる可能性がある.

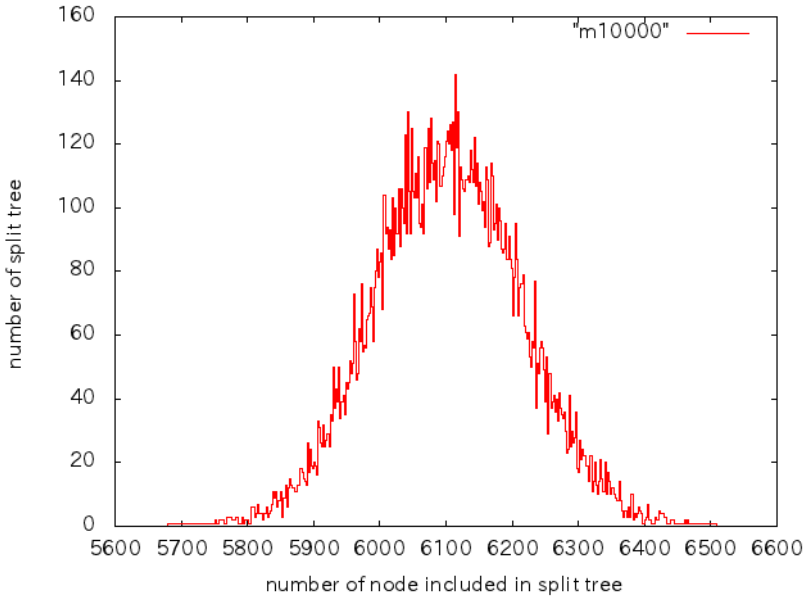

図 5.6　分割木に含まれるノード数のヒストグラム (m=10000)

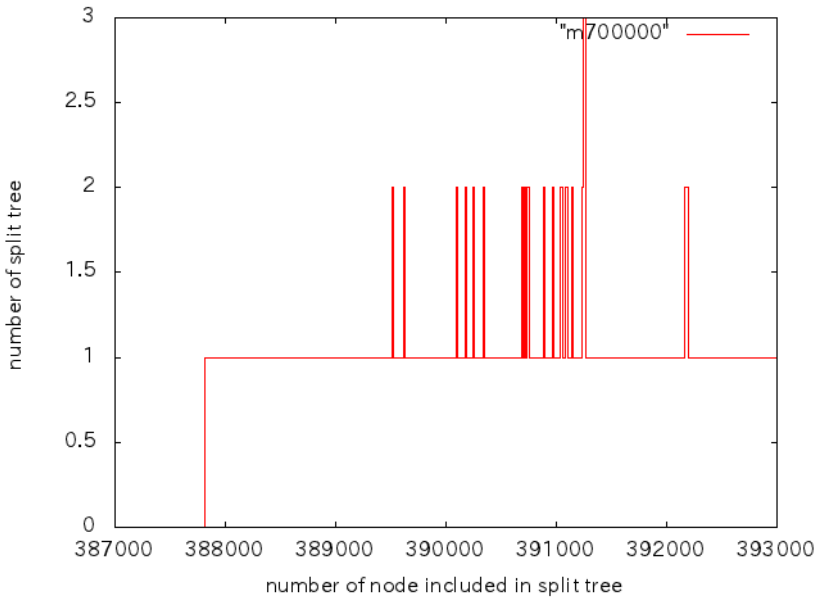

図 5.7　分割木に含まれるノード数のヒストグラム (m=700000)

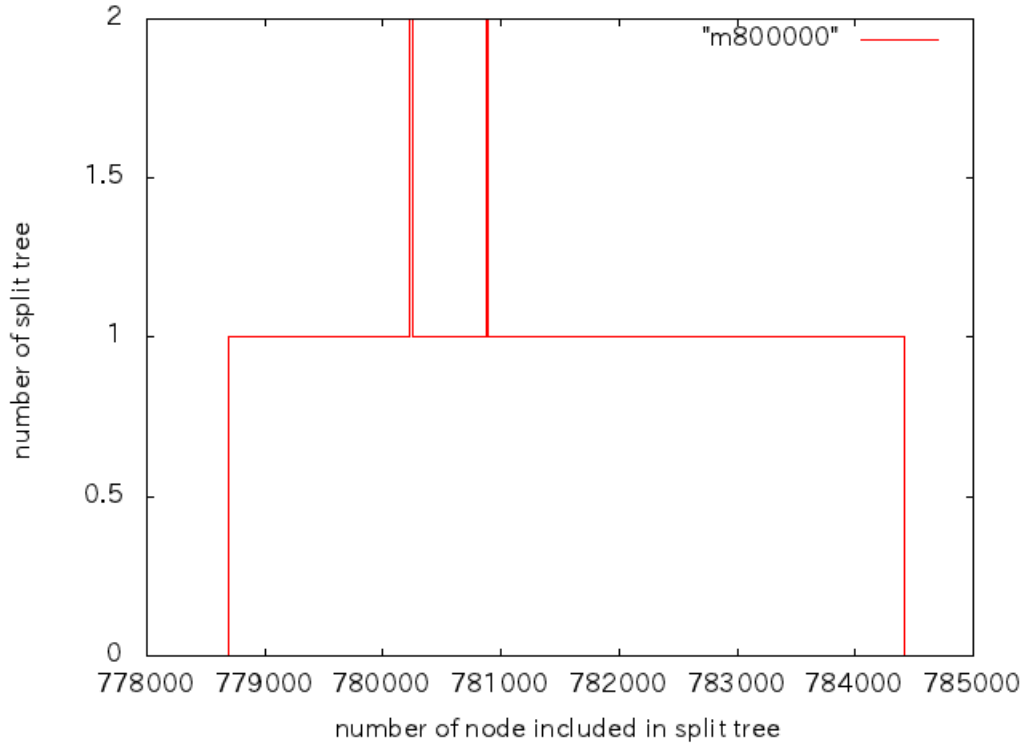

図 5.8　分割木に含まれるノード数のヒストグラム (m=800000)

第 6 章

まとめと今後の課題

6.1 まとめ

本論文では，MapReduce フレームワークを用いた木構造処理を実現した．実現にあたり，並列処理プログラム作成に伴う負荷分散やネットワークの制御を MapReduce フレームワークを用いることで自動化し，木構造処理に伴う兄弟・親子の依存関係の問題を m-bridge を用いることで解決した．よって，プログラマは並列処理，木構造処理に伴う問題を気にすることなく並列プログラミングを行えるようになった．そして，実際にどの程度効果があるのかを木の高さを求める並列計算プログラムを 8 台構成のクラスタを用いて実行して処理時間を計測する評価実験を行い検証した．結果としては，PC 台数を増やすことで，処理を高速化する事ができた．しかし，6 台以降は並列化の効果が薄くなっていた．原因としては，ネットワークのオーバヘッドの増加と Hadoop のパラメータ設定のチューニング不足である．

6.2 今後の課題

今回は全二分木に対して並列処理を行えるように実装を行ったが，m-bridge は全二分木以外の木や木以外にも適用することができる．そのため，今後は全二分木以外の木に適用できるようにしたい．今回の実験結果は，一つの木に対して m を変更しながら分割してデータを取ったため，木の左右の偏りといった木の形の違いによる処理時間の変化に関してはデータを取っていない．そのため，今後は今回実現した方法で，どのような形の木でも一定の効果があるのかを検証していきたい．そして，今回は m-bridge を前処理として逐次処理

で行ったが，今後は前処理も含めた並列化も可能にしたい．

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，副査をしていただいた山際伸一准教授，岩田誠教授にも御礼を申し上げます．そして，お忙しい中，東京から来て研究に関して助言していただいた美添一樹先生に感謝致します．最後に，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および吉田研究室の滝優基氏に感謝致します．

有難うございました．

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms Third Edition. pp. 286–307, 2009.

[2] 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.

[3] P. パチェコ 著, 秋葉　博 訳. MPI 並列プログラミング. 株式会社培風館, 2007.

[4] J. Dean and S. Ghemawat.　MapReduce: simplified data processing on large clusters.　In *OSDI*, pp. 137–150, 2004.

[5] H. Gazit, G. L. Mille and S.-H. Teng. Optimal Tree Contraction in the EREW Model. In *Concurrent Computations: Algorithms, Architecture and Technology*, pp. 139–156, 1998.

[6] 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, 2010.

[7] 野村　芳明, 江本　健斗, 松崎　公紀, 胡　振江, 武市　正人. 木スケルトンによる XPath クエリの並列化とその評価. 日本ソフトウェア科学会, コンピュータソフトウェア別冊 Vol.24, No.3, pp. 51–62, 2007.

[8] Bryan Catanzaro, Narayanan Sundaram and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *STMCS'08 Boston, Massachusetts USA*, 2008.

[9] Abhinav Sarje and Srinivas Aluru. A MapReduce Style Framework for Computations on Trees. In *ICPP*, 2010.

[10] Hubert Nguyen 著, 加藤　諒　編集, 中本　浩　翻訳. GPU Gems 3 日本語版. ボーンデジタル, 2008.

付録 A

実験で用いた Hadoop プログラムの一部

```
import java.util.*;
import java.lang.*;
import java.io.*;

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.filecache.DistributedCache;
import org.apache.hadoop.mapred.JobStatus;

public class mr_treecompute{
  public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    DistributedCache.addCacheFile((new Path(args[2])).toUri(),conf);
    Job job = new Job(conf, "mapreduce treecompute");
    job.setJarByClass(mr_treecompute.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setOutputKeyClass(IntWritable.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,IntWritable,Text>{
  private Text tree = new Text();
  protected void map(LongWritable key,Text value,Context context)
  throws IOException,InterruptedException{
    Node root;
```

```
    IntWritable finalkey = new IntWritable(1);
    String line = value.toString();
    String[] sv=line.split(":");
    String[] n = sv[1].split(" ");
    root=Create_binary_tree(n);
    tree.set(TreeCompute(Integer.parseInt(sv[0]),root));
    context.write(finalkey,tree);
  }
  public String TreeCompute(int serial,Node r){
    int max_hight=0;
    int current_hight=1;
    int mark_hight=0;
    Node current=r;
    ArrayDeque<Node> stack=new ArrayDeque<Node>();
    stack.push(current);
    while(true){
      current=stack.pop();
      current.set_calcflag(true);
      if(current.get_mark()){
           mark_hight=current_hight;
      }
      if(max_hight<current_hight){
           max_hight=current_hight;
      }
      if(current.get_left()==null && current.get_right()==null){
          if(stack.isEmpty()){break;}
          stack.push(current.get_parent());
          current_hight--;
      }else if(current.get_calcflag() && current.get_left().get_calcflag()
                && current.get_right().get_calcflag()){
          stack.push(current.get_parent());
          current_hight--;
      }else if(current.get_left().get_calcflag() ||
                  current.get_right().get_calcflag()){
          current_hight++;
      }else{
          stack.push(current.get_right());
          stack.push(current.get_left());
          current_hight++;
      }
      if(stack.isEmpty()){break;}
    }
    String maxh=Integer.toString(max_hight);
    String markh=Integer.toString(mark_hight);
    return new String(serial+":"+max_hight+","+mark_hight);
  }
}

class Reduce extends Reducer<IntWritable, Text, IntWritable ,IntWritable>{
  protected void reduce(IntWritable key,Iterable<Text> values,Context context)
                        throws IOException,InterruptedException{
    HashMap<Integer,Node> hashmap=new HashMap<Integer,Node>();
    IntWritable hight = new IntWritable();
    Node root=null;
    String line;
```

```
    for(Text val : values){
      line=val.toString();
      String[] sv=line.split(":");//serialnumber と value に分離
      String[] mhmkh=sv[1].split(",");//mhmkh は [0] に maxh,[1] に markh
      hashmap.put(Integer.parseInt(sv[0]),new Node(Integer.parseInt(mhmkh[0]),
                  Integer.parseInt(mhmkh[1])));
    }
    try{
      Configuration conf = context.getConfiguration();
      Path[] treepatternFile = DistributedCache.getLocalCacheFiles(conf);
      BufferedReader fis = new BufferedReader(new FileReader(
                           treepatternFile[0].toString()));
      String treepattern = fis.readLine();
      String[] treepatternArray = treepattern.split(" ");
      root = Create_binary_tree(treepatternArray);
    }catch(IOException e){
      System.err.println("treepattern error:"+e);
    }
    hight.set(TreeCompute(hashmap,root));
    context.write(key,hight);
  }
  public int TreeCompute(HashMap<Integer,Node> hashmap,Node node){
    int result,L,R;
    Node current;
    current=(Node)hashmap.get(node.get_value());
    if(node.isEmpty()){
       return Math.max(current.get_maxh(),current.get_markh());
    }
    L=TreeCompute(hashmap,node.get_left());
    R=TreeCompute(hashmap,node.get_right());
    result=Math.max(L,R)+current.get_markh();
    return Math.max(current.get_maxh(),result);
  }
}
```