平成23年度　　卒業研究論文

Hadoopフレームワーク上での分散並列画像処理フレームワークの基礎検討

指導教員
松尾　啓志　教授
津邑　公暁　准教授

名古屋工業大学　情報工学科
平成20年度入学　20115143

水野　航

目 次

第1章

はじめに

画像処理において、扱うデータの膨大さ、アルゴリズムの複雑さにより単一CPUでは処理に時間がかかりすぎるという問題点がある。特に、大規模画像処理の分野では扱うデータが巨大であるためより処理時間がかかってしまう。一方、画像処理の分野では、画素ごと、もしくは小領域を単位として並列実行可能なアルゴリズムも少なくない。また、計算機の低価格化および高性能化に伴い、数十個ものコアをもつCPUを搭載したメニーコアマシンの普及や、ネットワークで結合された複数計算機群（クラスタ）を容易に実現することが可能となった。これらの環境を対象として並列コンピュータの研究が盛んに行われている。並列コンピュータのアーキテクチャとして、3つに分類することができる。それぞれ独立したプロセッサとメモリがネットワークで結合したものを分散メモリ型、一つのメモリを複数のプロセッサが共同で使用するものを共有メモリ型、そして、共有メモリがネットワークで結合したものを分散共有メモリ型、以上の三つである。

クラスタ環境、つまり、分散メモリ型並列計算機を用いて並列処理を記述するための方法として、Message Pssing Interface[1]（以下MPI）が有名である。MPIはMPIフォーラムにより規格化されたメッセージパッシングAPI仕様MPIが一般的に用いられている。このライブラリを利用することで、比較的用意に分散処理を記述することができる。しかし、これらは比較的低水準のライブラリとして実装されているため、利用にあたっては明示的に手続きを記述する必要があり、利用には分散処理の知識が必要不可欠となる。

また、メニーコアマシン、つまり、共有メモリ型並列計算気を用いて並列処理を記述するための方法として、OpenMP[2] が有名である。OpenMP は基本となる言語に対する指示文を基本としたプログラミングモデルである。この OpenMP は、従来のマルチスレッドを基本とするプログラムに対して移植性が高く、かつ従来の逐次プログラムからの行こうも考慮されている。しかし、OpenMP においても並列実行、同期をプログラマが明示する必要があり、プログラマの負担は大きい。

本研究の目的として、分散処理フレームワークである Hadoop[3] を用いて分散並列画像処理用のライブラリを作成し、そして、分散・並列を意識しないで画像処理を記述可能な処理系を作成することである。その予備評価として実際に Hadoop を用いて平滑化の画像処理を実行し、24 コアマシンにおいてどの程度高速化が図れるか評価を行った。また、画像処理の記述には既に開発されている Hadoop の画像処理の API である HIPI[4] を用いて記述を行った。

本論文では、2 章では分散処理フレームワークである Hadoop について説明し、3 章では Hadoop の画像処理 API である HIPI の説明を行う。4 章では実装した画像処理について述べ、5 章では実装した画像処理の評価と考察を述べて、最後に 6 章で本論文全体をまとめる。

第2章

研究背景

本章では、本研究の研究背景として大規模画像処理についてと分散並列処理のフレームワークである Hadoop を紹介する

2.1 大規模画像処理

大規模画像処理 [5] とは、巨大な一枚の画像を処理することや、複数枚の画像を処理することである。例えば天文の分野や医療など様々な分野においては、扱う画像の画素数が大きく、また画像数も膨大となってきている。このような画像処理を一台の CPU コアで逐次的に処理を行うと現実的な時間で処理を終了させることが困難である。そのため、複数台のマシンを用いて分散・並列に処理を行うことが効率的である。また、画像処理アルゴリズムの中には画素ごと、または、対象画素周辺の小領域を用いたフィルタ処理など並列に処理を行うことが可能である処理が数多く存在するため分散・並列に処理を行うことに適している。しかし、分散・並列処理の記述には本来の画像処理プログラムの記述以外にも通信やデータの分散、障害時の処理など多くのプログラムの記述が必要であり、多くの知識が必要である。それらの知識を持たないユーザにとってはプログラムの記述が困難である。そこで、本研究では分散並列画像処理の処理系を開発するための基礎検討を行う。

2.2 Hadoopの概要

Apache Hadoopプロジェクトでは、信頼性の高いスケーラブルな分散コンピューティングのためのオープンソースソフトウェアを開発している。Hadoopのサブプロジェクトの有名なものとして、Google の MapReduce[6] や Google File Systemなどのオープンソース実装の開発が進められている。
また、Hadoopのサブプロジェクトにはデータに対して高いスループットでのアクセスを可能にする分散ファイルシステムである Hadoop Distributed File System[7]、膨大なデータセットを計算クラスタ上で分散処理するためのソフトウェアである Hadoop MapReduce[8] がある。以下でその二つについて説明をしていく。

2.3 Hadoop Distributed File System

Googleが使っている分散ファイルシステム GFS のオープンソースによる実装が Hadoop の Hadoop Distributed File System（HDFS）である。HDFS はコストの低い複数のマシン上に配置される。HDFSはひとつのデータをブロックの単位に分割して扱うが、これらのブロックをデータノードと呼ばれるノードが管理し、それらデータノードをネームノードが管理する。ネームノードはファイルの入出力命令があった場合にデータノードから要求されたファイルのブロックを集めたり、分配する。また、一定時間毎にデータノードとハートビート通信と呼ばれる通信を行い正常に動作しているかどうか確認を行う。
HDFSの特徴として大容量、スケーラビリティ、高スループット、耐障害性などが挙げられる。HDFSは大容量で高スループットであるため、サイズが大きなデータを扱うことに適しているが、逆に小さなデータを扱うことには適していないという欠点がある。HDFSは先に挙げたように高い耐障害性を持つ。HDFSのノードやサーバに異常が発生した場合には自動でデータの復旧を行う。異常の検知は先ほど説明したハートビート通信の有無によって判断する。また、あるノードが故障した場合に備えてノー

ド間で自動でデータのレプリケーションを行う。このようにデータの入出力や障害発生時の対処は HDFS が自動で行うためユーザはあたかも一つのファイルシステムを利用しているように分散ファイルシステムである HDFS を利用することが可能である。

2.4 Hadoop MapReduce

2.4.1 Hadoop MapReduceの概要

MapReduce とは、ひとつのマシンでは処理できない、もしくは時間がかかるような膨大なデータに対する処理をより小さい処理の単位に分割し、大規模な計算ノード・クラスタ上で実行することで高速に処理するために Google によって開発されたプログラミングモデルである。そして、2004 年に Google によって紹介された MapReduce の論文を元にオープンソースとして実装されたのが Hadoop MapReduce（以下 MapReduce）である。MapReduce のジョブは、クライアントが実行を要求する作業単位であり、Hadoop はジョブを並列処理可能なタスクに分割して実行する。そして、このタスクを空いた CPU に順次割り当てることにより、資源を効率よく利用し高速に処理を行う。

ジョブの実行プロセスを制御するノードは 2 種類存在する。タスクのスケジューリングを行ったり、ジョブの進行状況を把握するなど、ジョブの実行を管理する JobTracker と、タスクを実際に実行する TaskTracker である。TaskTracker はタスクを実行する際にそれぞれで JVM を起動しそれらにジョブを行わせる。実際のジョブの実行においてひとつの JobTracker と複数の TaskTracker が強調して動作し、MapReduce プログラムを実行する。MapReduce のデータフローを表したものが図 2.1 である。

MapReduce ではすべてのデータを、map,reduce という 2 つのフェーズに分けて処理を行っている。それぞれのフェーズにおいて同時に実行される map タスク、reduce タスクは全て独立した処理として実行することができる。これにより、大規模な並列分散処理を可能としている。また、MapReduce はこのようにアルゴリズムが単純明快で

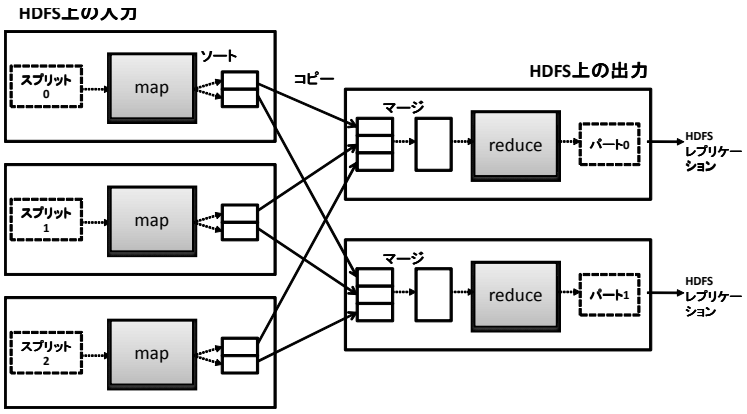

図 2.1: データフロー

あるためこれを用いて様々な処理を行うことができる。

MapReduce の処理全体の流れ

MapReduce のデータフローを表したものが、図 2.2 である。まず、map フェーズにおいて、Hadoop は MapReduce ジョブへの入力を、入力スプリット（あるいは単にスプリット）と呼ばれる固定長の断片に分割する。そして、各スプリットに対してひとつの map タスクを生成し、map タスクはスプリットの各レコードに対して map 関数を実行する。MapReduce では、すべてのデータは key-value というシンプルなデータ構造で扱われる。ひとつのレコードからひとつの key-value ペアが生成され、map 関数に渡されることになる。map フェーズでは受け取ったデータから必要な情報を取り出し、中間的なデータとして出力する。map フェーズで出力された key-value ペアに対して key 順にソート処理を行い、同じ key は key-values ペアというひとつの key に対してひとつ以上の value を持つペアとして reduce フェーズの入力値として渡される。

次に reduce フェーズでは、map の出力値を入力値として受け取り、そのデータに対して reduce 関数を実行することで map の出力を集約する。reduce フェーズにおいても複数の reduce タスクを同時に実行することが可能である。その場合は map タスク

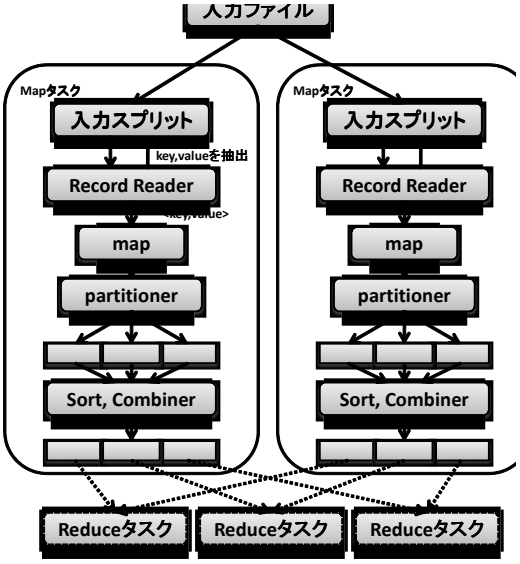

図 2.2: map タスクの詳細なデータフロー

はその出力をパーティション化しそれぞれの reduce タスクに対してひとつのパーティションを生成する。同じ key は必ずひとつのパーティションに含まれるように生成される。これによって、複数の reduce タスクを同時に実行することが可能になる。

2.4.2 map 処理の詳細

map フェーズにおける詳細なデータフローを表したものが図 2.2 である。入力としてあたえられたファイルは入力スプリットに分割されると先ほど説明したが、この入力スプリットを表す部分が InputSplit である。InputSplit はバイト単位の長さとストレージ上の位置を保持している。その InputSplit に対してひとつの map タスクが割り当てられ、スプリットからレコード単位で key,value を取り出すために RecoordReader が生成される。この RecordReader はレコードに対する単なるイテレータ以上のもので、map タスクはこれを用いて map 関数に渡す key-value ペアを生成する。key,value を受け取った map 関数はそれぞれの key,value に対してユーザが定義した処理を行い、

出力を行う。map 関数の出力は partitioner により複数のパーティションに分割される。デフォルトでは key のハッシュ値を用いて分割される。各パーティション内では key によるソートが行われ、combiner 関数があたえられた場合はソートの出力に対してその関数が実行される。combiner 関数はメモリバッファからあふれた出力に対して map タスクの出力をコンパクトにするために用いられる。そして最後に、パーティションで区切られたデータはそれぞれ別の reduce タスクの入力として与えられる。

2.4.3 reduce 処理の詳細

reduce フェーズにおける詳細なデータフローを表したものが、図 2.3 である。map の出力ファイルは map タスクが実行されたローカルディスク上にある。そのため、reuce タスクは複数の map タスクから自分に割り当てられたパーティションのファイルをコピーする必要がある。コピーする必要があるファイルは複数のマシン上で実行される map タスクから取得する必要があり、またそれらの map タスクが終わる時間が異なる可能性がある。よって、reduce タスクはそれぞれの map タスクが終了するとすぐにその出力のコピーを開始する。reduce タスクは少数のコピースレッドを持ち、並列に map タスクの出力を取得することができる。自分が必要な map の出力をすべてコピーし終わると reduce タスクは入力を map のソート順序を保証しながらマージする。そして、key-value ペアというデータ形式、言い替えると key と value のリストとして各 key ごとに reduce 関数に渡され、ユーザが定義した処理を行い、そして出力を行う。最後に、reduce の出力は RecordReader によって書き込まれ reduce タスクは終了する。

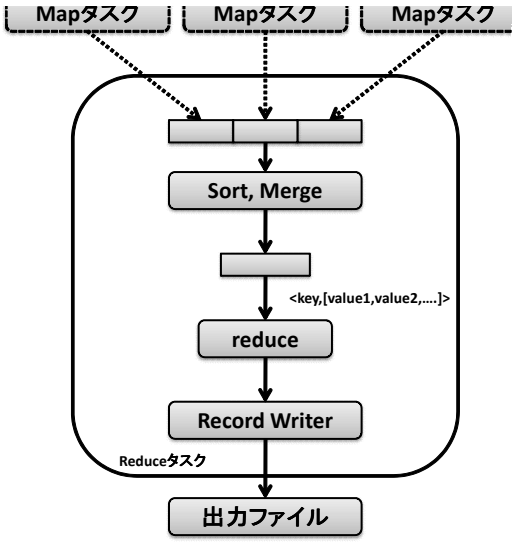

図 2.3: reduce タスクの詳細なデータフロー

2.5 Hadoopを用いた画像処理の問題点

先ほど説明したように、Hadoop の入力ファイルは InputSplit と呼ばれる単位に分割され、それぞれが複数のマシン上で分散されて処理される。InputFromat は入力スプリットの生成を行い、それをレコードに分割する役割を果たす。ユーザは利用したい map タスク数を指定することができるが、InputFormat は指定された map タスク数をヒントにして用いるだけで指定された値と異なる数のスプリットを生成しても構わない。しかし、画像データを分割し、並列に処理するという用法において決められたとおりの分割を行うようにする必要がある。そのため、新しく画像データを指定された map タスク数ごとに分割するための処理を実装する必要がある。この問題を解決するために ImageSplit を実装した。この実装の詳細については第 4 章で紹介する。

また、Hadoop で画像処理を行う場合の問題点として、Hadoop には画像データを key-value ペアとして扱うための専用の機構が存在しないことである。Hadoop では SequenceFile というものを用いてバイナリデータを key-value ペアとして扱うことができるが、これは画像データ用の機構ではないため画像処理プログラムを記述するさいに

InputFormat

```
public interface InputFormat<K, V>{

  InputSplit[] getSplits(jobConf job, int numSplits);

  RecordReader<K, V>getRecordReader(InputSplit split,
                                    jobConf job,
                                    Reporter reporter);
}
```

図 2.4: InputSplit インターフェースのソースコード

直感的に扱いにくい。そこでこのような問題を解決するために、Hadoop で画像処理を行うための API である HadoopImageProcessingInterface（HIPI）を用いることにした。HIPI については第 3 章で詳しく説明を行う。

第3章

Hadoop Image Processing Interface

3.1 Hadoop Image Processing Interface

Hadoop Image Processing Interface(以下 HIPI) はヴァージニア大学で開発された Hadoop で画像処理プログラムを記述するための API である。元来 Hadoop では画像データを専用に扱う機構は存在しなかったため、画像データは扱いずらかった。HIPI はこの問題を解決し、Hadoop を用いたプログラムでも画像データを直感的に分かりややすく扱うことができるようにすることを目的に開発された API である。

3.1.1 FloatImage クラス

FloatImage クラス [9] は画像ファイルをピクセルの集合として表すためのクラスで、値は単精度浮動小数点で表される。FloatImage は実際には三次元の組によって浮動小数点を表している。これにより、三次元配列のように画像データを扱うことができる。また、このクラスには画像データを扱うためのいくつかのメソッドが用意されている。図 3.1 に FloatImage クラスのコンストラクタと主なメソッドを示す。コンストラクタでは画像の横幅、高さ、そしてバンド幅を指定する。ここで、getPixel と setPixel の引数について説明をする。それぞれの第 1 引数と第 2 引数は順番に画像データの x 座標、y 座標を指定する。そして、第 3 引数は RGB 配色のうちどの色の画素を取得また

FloatImage クラスの主なメソッド

```
・Constructer
FloatImage(int width, int height, int bands)

・add(FloatImage image)
Adds a FloatImage to the current image

・convert(int type)
Convert between color types(blacj and white,etc.)

・getHeight()
Gets the Image Height

・getWidth()
Gets the Image Width

・getPixel(int x, int y, int c)
Get the Pixel indicated by (x,y,c)

・setPixel(int x, int y, int c, float val)
Set the Pixel as val to (x,y,c)
```

図 3.1: FloatImage クラスの主なメソッド

は変更するかを表す。赤、緑、青の順番に 0,1,2 と数字が割り当てられている。

setPixel の第 4 引数は書き込む画素値を与える。

このように、FloatImage クラスを用いることで画像データを直感的に扱うことが容易に可能になった。

HipiImageBundle クラスの主なメソッド

```
・Constructer
public HipiImageBundle(org.apache.hadoop.fs.Path file_path,
                       org.apache.hadoop.conf.Configuration conf)

・addImage(java.io.InputStream image_stream,
                       ImageHeader.ImageType type)
Add images to an already existing HIB

・append(HipiImageBundle bundle)
Merge two HIBs
```

図 3.2: HipiImageBundle クラスの主なメソッド

3.1.2 HipiImageBundle クラス

HipiImageBundle(以下 hib)[10] は複数の画像データをひとつの hib 形式のファイルにまとめたものである。これは、UNIX の tar 形式のファイルと似ている。hib は HipiImageBundle クラスによって実装されている。hib は HIPI を用いた画像処理プログラムでは入力に用いられる。入力された hib は画像ごとの単位で map タスクに割り当てられる。また、hib を用いることで HDFS を効率的に利用することが可能になる。HDFS はブロック単位でデータの管理を行っているがブロックひとつは 64MB という大きなサイズである。一方、多くの画像データは数十 kb 程度とそれほど大きなサイズではない。よって、画像データひとつひとつを HDFS に格納するよりも、ひとつの hib に複数の画像データを格納しておきその hib を HDFS に格納した場合のほうが効率的である。

図 3.2 に HipiImageBundle のコンストラクタクラスと主なメソッドを示す。

HIPIを用いたグレースケール化プログラム

```
FloatImage gray = new FloatImage(image.getWidth(),
                                        image.getHeight(), 3);
for(int x=0; x < image.getWidth(); x++){
   for(int y=0; y < image.getHeight(); y++){
     float red = image.getPixel(x, y, 0);
     float green = image.getPixel(x, y, 1);
     float blue = image.getPixel(x, y, 2);

     float avg = (red + green + blue)/3;

     gray.setPixel(x, y, 0, avg);
     gray.setPixel(x, y, 1, avg);
     gray.setPixel(x, y, 2, avg);
    }
}
```

図 3.3: HIPIを用いたグレースケール化プログラム

3.1.3 HIPIを用いたプログラムの実装例

グレースケール化処理を実装した。mapタスクで入力画像を受け取り、それをグレースケール化しreduceタスクで処理済みの画像を出力する。
図3.3にHIPIを用いて記述したグレースケール化プログラムを示す。

このプログラムでははじめにFloatImage型の変数grayを宣言する。初期化に用いられているimageという変数はFloatImage型の変数で入力された画像が格納されている。このgrayの横幅、高さはそれぞれ入力画像の横幅と高さと同じである。先ほど説明したようにこのFloatImage型の変数は多重配列のように扱うことが可能である。次の二重forループの中ではgetPixelメソッドを用いて各RGB配色ごとの画素を取得している。getPixelの返り値はFloat型であるためそれぞれの変数はFloat型で宣言する。画素値を取得したらそれらの平均値をとる。そして、その値をsetPixelを用いて画像に書き込む。

このように、HIPIを用いることによってプログラマはHadoopを用いて容易にプログラムを記述することが可能である。

3.2 HIPIを用いた実装の問題点

HIPIを用いることで、画像データを直感的に扱うことが可能になった。それにより、Hadoopで画像処理プログラムを記述することが容易になった。しかしながら、HIPIを用いた実装には問題点がある。それは、HIPIではひとつの画像を切り分けて処理をすることができないということである。HIPIではhibに複数の画像データを格納し、それをひとつひとつの画像データに分割して処理をすることは可能であるがひとつの画像を分割するようには実装されていない。これによって、大規模画像処理で行われる巨大な一枚の画像の処理を行う場合には適していない。この問題を解消するために、新しくImageSplitを実装することにした。ImageSplitについては次章で詳しく解説をする。

第4章

実装

4.1 ImageSplitの実装

第3章で説明したようにHIPIではひとつの画像を複数に分割することはできない。そこで、そのような分割を可能にするためにImageSplitを実装した。
ImageSplitではまず画像データをHIPIのFloatImageへと変換し、FloatImageクラスに実装されているメソッドであるgetHeight,getWidthによって画像の縦、横の長さを取得する。そして、mapタスクの数分に分割を行った。今回は単純に画像を横切りにし分割した。そして、分割した画像データを一旦HDFSに出力しファイル名を分割された位置を表す番号にする。そのファイル名のリストを画像処理を行うプログラムのmapタスクに渡すようにした。図4.1にImageSplitのプログラムを示す。
プログラムでははじめに分割数numを指定する。ここで指定された分割数numで画像の高さを割り、分割された画像の一つ分の高さをsizeに格納する。次に分割数分forループを回して画像を分割する。ここでは分割した画像データを格納するためのFloatImage型の変数であるsplitが宣言されている。引数は入力された画像の横幅と分割された画像一つ分の高さであるsizeを用いる。次の二重forループの中では入力画像であるvalueのピクセルごとの値をgetPixelによって取得し、それをsetPixelによって分割画像であるsplitに書き込むという処理を行う。この際、getPixelの第二引数で指定する必要がある画像のy座標は三つのforループの外側である分割数分ループを回す部分で用いられる変数であるiと分割画像一つ分の高さを示す変数sizeを用いて表される。これは、変数iは現在処理している画像が分割画像の何番目に当たる

ImageSplit の画像分割部分

```
//map タスク数を指定し、入力画像をその数分横切りにする
int num_maptask = 16; ////map タスク数をここで指定する！
int size = value.getHeight() / num_maptask;

for(int i=0; i < num_maptask; i++){
    //入力された画像を map タスク数分に分割する
    //新しい FloatImage 型の変数 split に格納する
    FloatImage split = new FloatImage(value.getWidth(), size, 3);

    for(int y = 0; y < size; y++){
for(int x = 0; x < value.getWidth(); x++){
    float red = value.getPixel(x, y+size*i, 0);
    float green = value.getPixel(x, y+size*i, 1);
    float blue = value.getPixel(x, y+size*i, 2);
    split.setPixel(x, y, 0, red);
    split.setPixel(x, y, 1, green);
    split.setPixel(x, y, 2, blue);
}
    }

    ////画像出力部////
    ImageEncoder encoder = JPEGImageUtil.getInstance();
    //a neccessary step to avoid files with duplicate hash values
    Path outpath = new Path(path + "/" + i + ".jpg");
    String tmp_path = outpath.toString();

    while(fileSystem.exists(outpath)){
String temp = outpath.toString();
outpath =
     new Path(temp.substring(0,temp.lastIndexOf('.')) + i +".jpg");
tmp_path = temp.substring(0,temp.lastIndexOf('.')) + i +".jpg";
    }

    FSDataOutputStream os = fileSystem.create(outpath);
    encoder.encodeImage(split, null, os);
    os.flush();
    os.close();
```

図 4.1: ImageSplit の画像分割部分

かを表しているため、i × size を行えば入力画像の必要な部分の高さを表すことが出来るからである。二重 for ループを抜けると画像の出力をする。HIPI では画像の出力 JPEGImageUtil クラス [12] の encodeImage メソッドを用いる。この encodeImage メソッドは引数に FloatImage、ImageHeader,OutputStream が必要である。この中で ImageHeader は null でも良い。FloatImage には出力する画像データを用いれば良いため今回は split を用いる。OutputStream は新しく指定する必要がある。今回は画像が前から何番目に分割されたかを表す番号をファイル名にするようにした。次に OutputStream である os を定義したが、Hadoop のファイルシステムである hdfs にファイルを出力する場合には FSDataOutputStream を用いる必要がある。逆に hdfs からのファイル入力には FSDataInputStream を用いる必要がある。そして、定義した os を encodeImage メソッドの第三引数に用いて画像を出力する。

今回は平滑化処理を行うため、処理を行うピクセルとその近傍のピクセルが必要である。今回の平滑化処理には 11 × 11 のマスクを用いるため 11 ピクセル分多めに画像を分割した。
実際の動作の流れを図 4.2 に示す。

この ImageSplit を用いて画像の平滑化処理を実装した。

4.2 平滑化処理

平滑化処理 [11] とは画像の雑音を取り除きたい場合や画像をぼかしたい場合に用いられる処理のことである。この平滑化処理には平均値フィルタと呼ばれるフィルタ処理を用いる。これは、画像の輪郭や込み入った部分などの高い周波数成分をなくして逆に低い周波数成分を残す処理である。実際の処理を行う場合には処理対象ピクセルをそのピクセルと周辺ピクセルの平均値にすることで実現する。周辺領域はマスクと

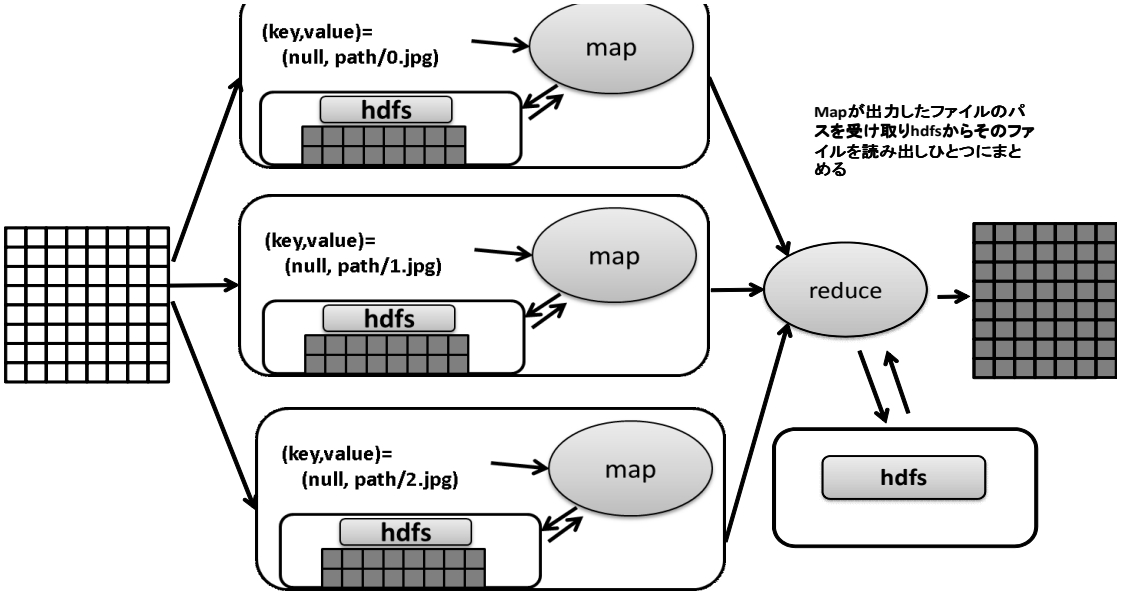

図 4.2: ImageSplit の処理の流れ

呼ばれる。今回の平滑化処理には 11 × 11 のマスクを用いた。

4.2.1 処理の流れ

先ほど 3 章で説明したグレースケール化プログラムと同じような処理の流れである。まず、改良した ImageSplit から画像を map タスクで平均値フィルタを用いて平滑化処理し、そして reduce タスクでそれらの画像をまとめて出力した。

第5章

評価と考察

5.1 評価環境

今回はどの程度並列に処理することが可能であるかを評価するために 24 コアマシンを用いて、map 数を変化させて評価を行った。クラスタ環境で評価を行った場合外乱が発生するため、評価結果が分かりづらくなってしまう。そのため、評価に使用した環境は以下のとおりである。

OS	CentOS 5.5
kernel	`x86_64` GNU/Linux 2.6.18
CPU	(AMD Opteron 12-core 6168 / 1.9GHz) × 2
Memory	32GB × 2

5.2 平滑化処理

今回の評価には大規模画像処理で扱われる画像を想定して 7000 × 4000 ピクセルの画像を用いた。画像を横向きに 20,40,80,125 と分割した画像をそれぞれ用いた。また、map タスク数は 1,2,4,8,16 ,24 と変化させ、reduce タスク数は常に 1 でそれぞれ評価を取った。これにより、map 数とデータの分割数が変化することにより処理速度にどの

	分割数 20	分割数 40	分割数　80	分割数 125
1	140	228	392	599
2	128	221	386	587
4	107	207	381	579
8	124	204	379	573
16	125	207	389	584
24	133	209	393	577

図 5.1: 平滑化処理の評価結果

ような変化が現れるか調べた。図 5.1 に処理結果を示す。

表の横軸は画像データの分割数を表し、縦軸は map 数を表す。そして、計測結果はすべて秒で表されている。
以上の結果を見ると、分割数が小さいほうが処理速度がはやいことが分かる。しかし、分割数が増えることにより各 map タスクで処理する画像のピクセル数が小さくなるため処理時間は分割数が大いほど高速化できると予想していたが、それとは大きく異なった結果になってしまった。この原因を調べるため Hadoop の処理ごとの処理時間を計測した。図 5.2 に分割数ごとの map の処理時間と reduce の処理時間の合計を示し、図 5.3 に Hadoop の処理ごとの処理時間を示す。

5.2.1　考察

図 5.1 では分割数ごとの処理時間の内訳を示している。図では各 map タスク数につき三つのグラフがあるが、これは左から順にデータの分割数が 20,40,80 である。また、グラフはそれぞれ map 処理、map ＋ reduce 処理、reduce 処理、setup ＋ cleanup と分かれているが、map 処理は map 処理だけが行われている状態、map ＋ reduce 処理は map と reduce が並行して行われている状態、そして、reduce 処理は reduce 処理だけ

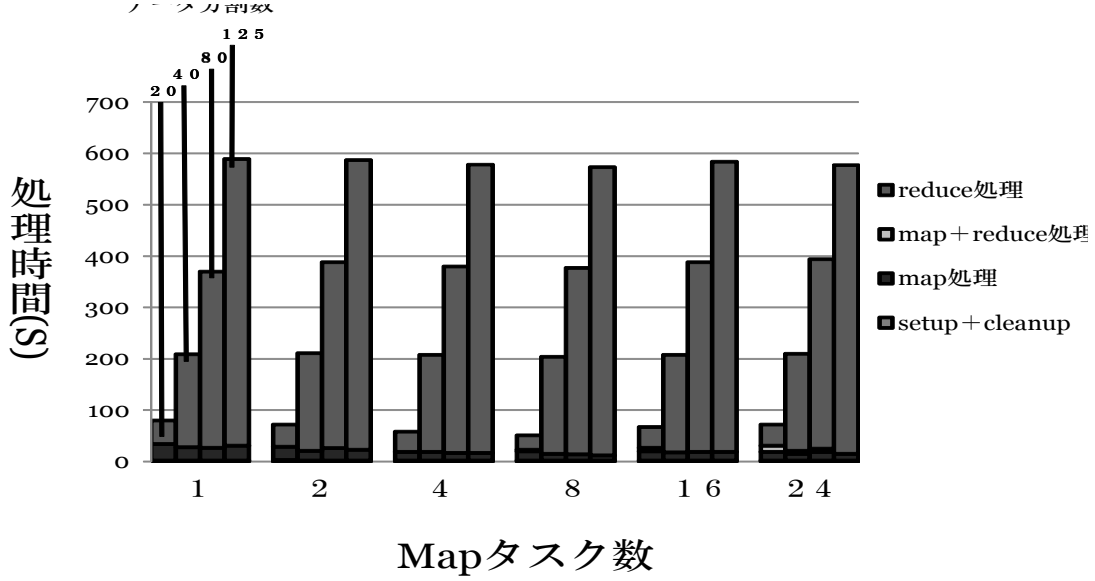

図 5.2: Hadoop の処理時間の内訳

が行われている状態を表してる。setup + cleanup は setup と cleanup という処理時間の合計時間を意味する。setup とは Hadoop がジョブを実行する前にタスクトラッカーにタスクを割り当てたり、map タスクへのデータの割り当てを行ったりする時間のことであり、cleanup はジョブ終了後に中間データを削除したりすることである。図を見ると reduce タスクに非常に多くの時間が費されていることが分かる。これは、今回の実装では map タスクで処理した画像データを一旦ファイルシステムに出力し、reduce タスクでそれを開いてからまた処理をするためであると考えられる。分割数が増えれば増える程ファイルの入出力が増えるためこのようにオーバーヘッドが大きくなっていると考えられる。

図 5.2 では分割数別の map タスクの処理時間を示した。これを見ると分割数が増えると処理時間は減少していることが分かる。これは、処理するデータが小さく分割された結果 map タスクひとつあたりの処理時間が減少したためだと思われる。データの分割数が 20 のときは map 数が 4 つのときが最も処理時間が短いがそれ以降は増加している。それに比べ他の二つは map 数が 8 のときに処理時間が最も短くなっている。これはデータの分割による処理時間の減少よりもジョブトラッカーとタスクトラッカー

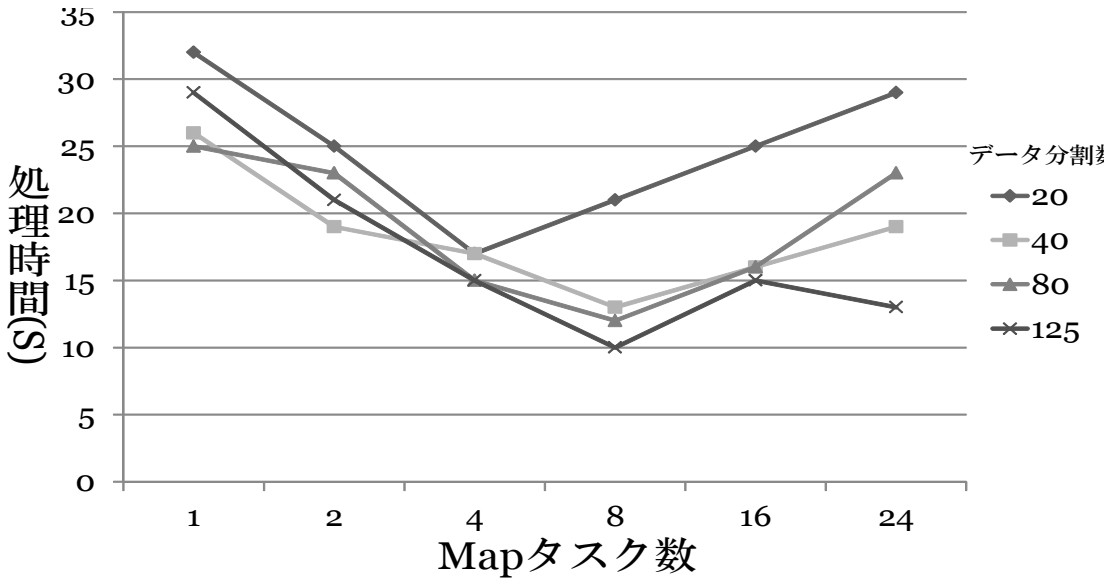

図 5.3: 分割数ごとの map 処理時間

間の通信やファイルの入出力によるオーバーヘッドが大きくなってしまったためであると考えられる。

よって、データの分割数や map 数増やすことによって map 処理の処理時間は高速化出来ていることが分かる。今後は reduce タスクの処理時間を減少させる手法を実装しなければならない。

5.2.2 今後の課題

データの分割数を増やすと各 map タスクでの処理時間を削減することが出来る一方で reduce タスクの処理時間が大幅に増加してしまうことを先ほど考察した。そこで、今後は reduce タスクの処理時間を削減する必要がある。現在の実装での問題点は map タスクから reduce タスクへデータを渡す際に一度ファイルシステムを経由している点である。よって、この部分を改良しなければならない。改良案としては画像デー

タをファイルシステムを経由せずに直接 map タスクから reduce タスクへと渡すことである。これを実現するためには HIPI の FloatImage のデータ構造を理解し、データの分割情報を追加できるように改良しなければならない。また、他の改良案としては reduce タスクを並列化する方法である。今回の実装では map タスクは並列実行しているが reduce タスクは一つしかなかった。そのため、処理が集中しデータの分割数が増えるほど処理時間が増加してしまった。よって、今後は reduce タスクを並列化することも考える必要がある。

第6章

おわりに

本研究ではまず大規模分散処理システム Hadoop について述べ、Hadoop を用いて画像処理をする上で問題となる点を述べた。そして、その解決策として Hadoop の画像処理ライブラリである Hadoop Image Processing Interface の利用と画像データを分割するために新たに ImageSplit を実装した。それらを用いて画像の平滑化の処理を実装し評価を行い、map タスクの処理速度を最大 30% 削減できた。しかし、reduce タスクのオーバーヘッドが非常に大きいため今後改良をする必要がある。

謝辞

本研究のたに多大な尽力を頂き、日頃から熱心な御指導を賜わった名古屋工業大学の松尾啓志教授、津邑公暁准教授、齋藤彰一准教授、松井俊浩准教授に深く感謝致します。また、本研究の際に多くの助言、協力をして頂いた松尾・津邑研究室、齋藤研究室ならびに松井研究室の皆様に深く感謝致します。

参考文献

[1] Message Pasiing Interface Forum, ”A Message-Passing Interface Starndard ”, 1995

[2] OpenMP, http://openmp.org/wp/

[3] hadoop, http://hadoop.apache.org

[4] HIPI hadoop image processing interface, http://hipi.cs.virginia.edu/

[5] 外山　篤史, ”大規模画像処理のための並列プログラミングインターフェース ”,http://ci.nii.ac.jp/ncid/AA11943613, 2009

[6] Jeffery Dean and Sanjay Ghemawat, ”MapReduce:Simplified Data Processing on Large Clusters ”, 2004

[7] `Apache Hadoop Project:Hadoop Distributed File System`, http://hadoop.apache.org/hdfs/

[8] Apache Hadoop Project:Hadoop MapReduce, http://hadoop.apache.org/mapreduce/

[9] Class FloatImage, http://hipi.cs.virginia.edu/doc/api/hipi/FloatImage.html

[10] Class HipiImageBundle, http://hipi.cs.virginia.edu/doc/api/hipi/imagebundle/HipiImageBundle.html

[11] アルゴリズム入門:第４章　画像処理入門２,http://msdn.microsoft.com/ja-jp/academic/cc998606

[12] Class JPEGImageUtil,
http://hipi.cs.virginia.edu/doc/api/hipi/image/io/JPEGImageUtil.html