平成22年度　　卒業研究論文

Hadoopを用いた分散画像処理の基礎検討

指導教員
松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成19年度入学　19115005番

荒木 孝

平成23年2月8日

目 次

第 1 章　はじめに **1**

第 2 章　研究背景 **3**

2.1 Hadoop の概要 3

2.2 Hadoop Distributed File System 3

2.3 Hadoop MapReduce 4

2.3.1 Hadoop MapReduce の概要 4

2.3.2 MapReduce 全体の流れ 4

2.3.3 map 処理の詳細 6

2.3.4 reduce 処理の詳細 6

2.4 Hadoop を用いた画像処理の問題点 8

第 3 章　実装 **10**

3.1 グレースケール化 10

3.1.1 FileInputFormat,RecordReader の改良 10

3.1.2 処理の流れ 11

3.2 kmeans クラスタリングを用いた減色処理 12

3.2.1 Mahout 12

3.2.2 kmeans によるクラスタリング 12

3.2.3 処理の流れ 13

第 4 章　評価と考察 **15**

4.1 評価環境 15

4.2 グレースケール化 15
4.2.1 評価 15
4.2.2 考察 16
4.3 kmeans クラスタリングを用いた減色処理 18
4.3.1 評価 18
4.3.2 考察 18

第 5 章 まとめと今後の課題 20

第1章

はじめに

近年，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．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，数十個ものコアを持つ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の計算機群(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である．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３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 Passing 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，比較的容易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機を用いて並列処理を記述

するための方法として，OpneMP[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移行も考慮されている．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，分散処理システム Hadoop[4] 上に，画像処理用のライブラリを作成し，そして，分散・並列を意識しない画像処理言語を作成することである．その予備評価として実際に Hadoop 上でグレースケール化と mahout の実装を用いたクラスタリングによる減色処理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．

本論文では，2 章では分散処理システムである hadoop について説明し，3 章では実装した画像処理について述べる．4 章で実装した画像処理の評価と考察を述べ，最後に 5 章で本論文全体をまとめる．

第2章

研究背景

2.1 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[3] 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．

Hadoopのサブプロジェクトの有名なものとして，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[6]，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フレームワークである Hadoop MapReduce[?] がある．以下では，その二つについて説明していく．

2.2 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．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．

2.3 Hadoop MapReduce

2.3.1 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．2004年にGoogleによって紹介されたMapReduceの論文をもと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等，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．実際のジョブの実行においてひとつのJobTrackerと複数のTaskTrackerが協調して動作し，MapReduceプログラムを実行する．また，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．

2.3.2 MapReduce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１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．MapRedece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

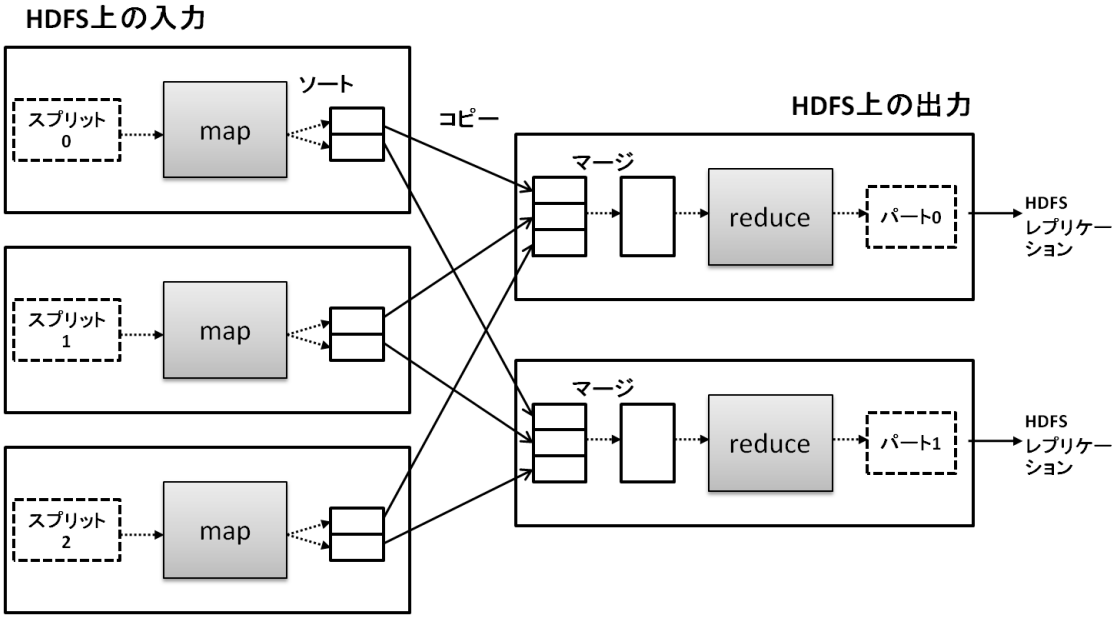

図 2.1: mapreduce

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 に対して，key 順にソート処理を行い，同じ key は key-values という，ひとつの key と一つ以上の value のリストとして reduce フェーズの入力として渡される．

次に Reduce フェーズでは，map の出力を入力として受け取り，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できる．そのときは，map タスクはその出力をパーティション化し，それぞれの reduce タスクに対して一つのパーティションを生成する．同じ key は必ず一つのパーティションに含まれるように生成される．これにより，複数の reduce タスクを同時に実行することができる．

2.3.3 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．入力として与えられたファイルは入力スプリットに分割されると説明したが，この入力スプリットを表すのが，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持っている．その InputSplit に対して一つの 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，RecordReader 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 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，出力する．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 関数が与えられた場合は，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溢れた出力に対して，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．

2.3.4 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，reduce 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．コピーする必要の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-values というデータ形式，言い替えると key と value のリストとし

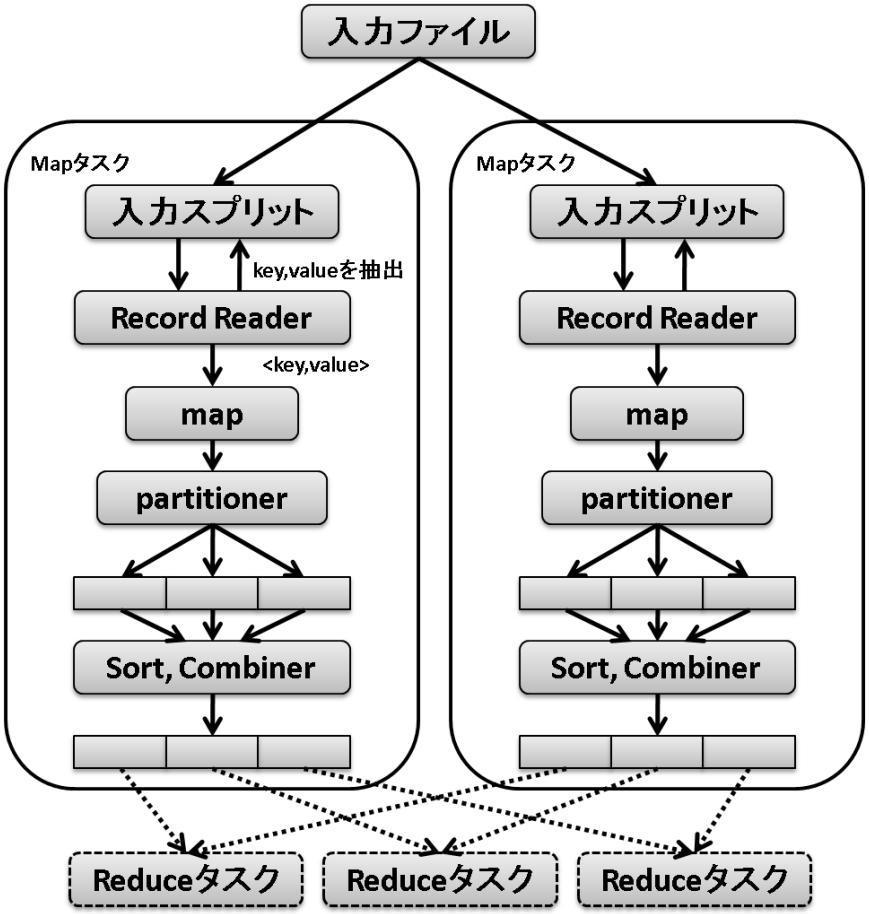

図 2.2: map タスクの詳細なデータフロー

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，出力する．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．

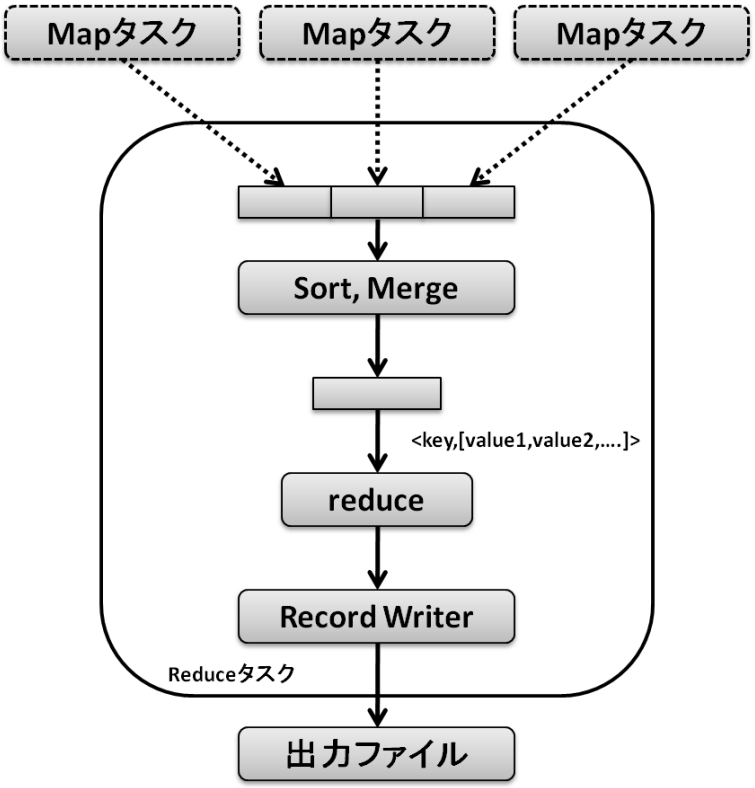

図 2.3: reduce タスクの詳細なデータフロー

2.4 Hadoopを用いた画像処理の問題点

先程説明したように，入力ファイルは InputSplit と呼ばれる単位に分割され，それぞれが複数マシン上で分散して処理される．Hadoop の実装において，InputFormat インターフェースのソースコードは図 2.4 のようになっている．InputFormat 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 タスク数をヒントとして用いるだけで，指定された値と異なる数のスプリットを生成してもかまわない．しかし，画像を分割し，並列に処理するという用途において，決められた通りの分割を行うようにする必要がある．そのため，InputFormat がスプリットを生成するために呼び出す getSplits() メソッドの実装を変更する必要がある．InputFormat はインターフェースであり，FileInputFormat クラスが InputFormat の実装のベースク

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: InputFormat インターフェースのソースコード

ラスとなっている．よって，今回 FileInputFormat の実装を改良した．また，入力スプリットから key-value を生成するために InputFormat は RecordReader を用いるが，画像から key-value を生成するためには独自の RecordReader を実装する必要がある．詳しい実装内容は 3.1.1 で説明する．

第3章

実装

3.1 グレースケール化

3.1.1 FileInputFormat,RecordReader の改良

第 2 章で説明したように，Hadoop は画像から直接 key-value を生成するための API が実装されていない．そこで，画像を入力として与えられるように FileInputFormat と RecordReader の改良を行った．

2 章で述べたように，Hadoop 上で画像処理をするためには，FileInputFormat の getSplits() メソッドを改良する必要がある．getSplits() メソッドは，入力の画像データのヘッダ部からその画像のサイズを取得し，ユーザが指定した分割数に従って，それに従いスプリットを生成する．今回は単純に縦に分割できるように実装を行った．

画像処理用の RecordReader として，ImageRecordeReader を実装した．ImageRecordeReader は，スプリットを受け取り，元の画像においてスプリットがどの範囲を表しているのかという情報を key とし，そのスプリットの全てのデータを表すバイト配列を value として，key-value を生成するようにした．

改良した FileInputFormat の動作例を表したのが，図 3.1 である．今回のグレースケール化の処理では，図の map においてグレースケール化を 10000 回行い，最後に単独の reduce によって，ひとつのファイルに書き出すという処理になる．

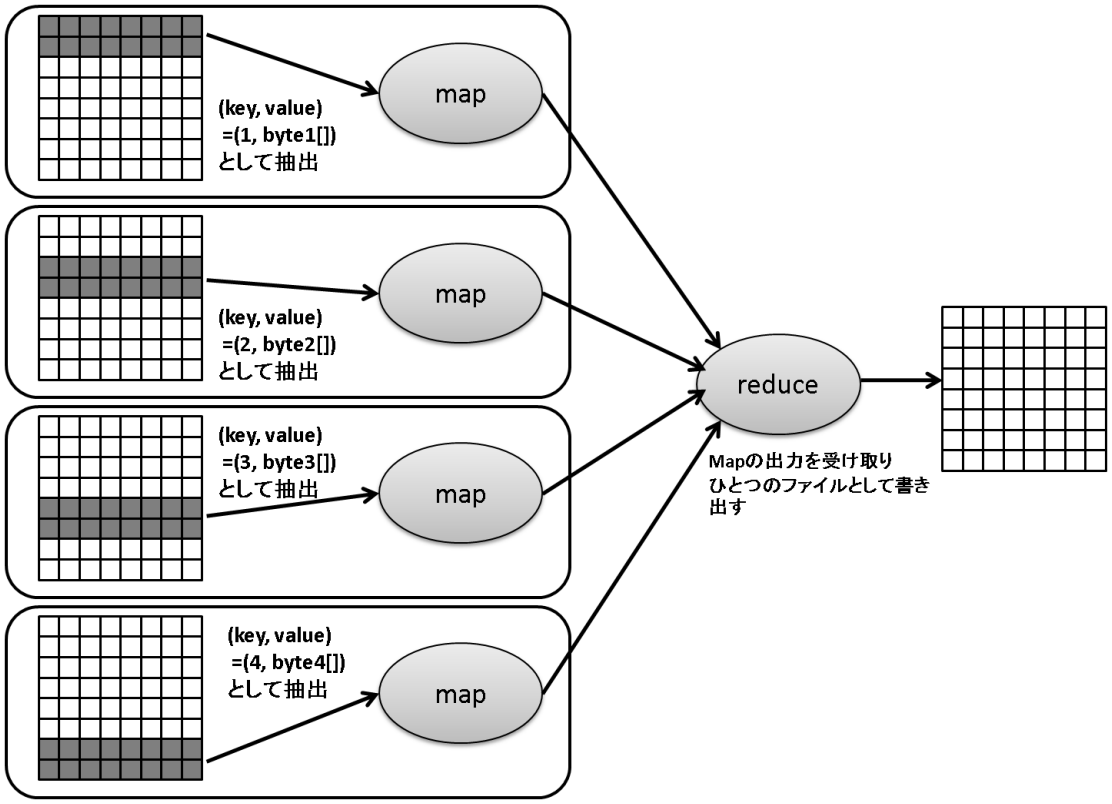

図 3.1: 改良した FileInputFormat の処理例

3.1.2 処理の流れ

改良した InputFileFormat を使用して，グレイスケール化を行う処理を実装した．グレイスケール化は軽い処理のため，各ピクセルに対して 10,000 回グレイスケール化を繰り返すようにし，擬似的に重い画像処理を再現した．

改良した FileInputFormat，ImageRecordReader を利用し，入寮画像から，key-value を生成し，map の入力に渡される．map 処理では，入力として渡された分割された画像のそれぞれのピクセル毎にグレイスケール化を 10,000 回繰り返した．これにより擬似的に重い処理を実現した．reduce 処理では map から変換後のバイト配列を受け取り，分割した画像をひとつのファイルに書き出している．

3.2 kmeansクラスタリングを用いた減色処理

3.2.1 Mahout

Mahoutは Apacheライセンスで拡張性のある機械学習用のツールを提供することを目的としたライブラリである．Mahoutのいくつかの実装は Hadoopの MapReduce上で使用することができる．Mahoutの主な実装として，

- Tasteによる協調フィルタリング
- 分散クラスタリングの実装
- 単純ベイズの実装

今回は Mahoutの kmeansクラスタリングの実装を用いて，減色処理を行った．

3.2.2 kmeansによるクラスタリング

クラスタリングとは

クラスタリングとは，分類したい対象の集合を，対象の要素の類似度に従って，部分集合 (クラスタ) に分割することが目的である．基本的なデータ解析手法のひとつであり，多くの分野で用いられる．

kmeans法

クラスタリングの手法は大きく，階層的手法と分割最適化手法の二つに分けられる．今回用いたクラスタリングの手法である kmeans法は分割最適化手法のひとつである．

クラスタ数を k，分類する分類対象の集合を X，集合の要素を $x_i \in X$ とすると，kmeans法のアルゴリズムは以下の通りである．

(step1) 初期値として k個の代表点 c_1,c_2,...,c_n をランダムに選択

(step2) X の要素 x ごとに，全てのクラスタの代表点とのユークリッド距離を計算し，最小となるクラスタにその要素を割り当て

(step3) もし代表点への割り当てが変化しなければ処理を終了し，そうでなければ各クラスタの重心を新たな代表点として step2 へ

3.2.3 処理の流れ

前処理として，画像をピクセルごとに key-value に変換し，ファイルに書き出す．key はピクセルの位置,value は Vector という，double 型の変数のリストを持つクラスとして書き出される．全体の処理の流れとして，KmeansMapper → KmeansCombiner → KmeansReducer という処理を，すべてのピクセルがクラスタを変更しなくなる，もしくは指定された回数だけ処理を繰り返し，最後に KmeansClusterMapper の処理によりピクセルごとのクラスタ ID がファイルに出力される．

KmeansMapper

(入力)　key:0　value:ピクセルのベクトル

(出力)　key:最も近いクラスタの識別子　value:KMeansInfo
(KMeansInfo:足し合わせたベクトルの個数と，ベクトルのそれぞれの成分の合計を持つクラス)

map 処理において実行されるのは KmeansMapper と KmeansCombiner です．KmeansMapper では初期値として与えた (2 回目以降は更新した) クラスタの中心を取得し，それぞれの入力ベクトルと全てのクラスタとの距離を計算し，最も近いクラスタを割り当てます．そしてここで key 順のソートが行われ，KmeansCombiner に渡される．

KmeansCombiner

(入力)　key,value:KmeansMapper の出力

(出力)　key:クラスタ識別子　value:KMeansInfo

map の出力をコンパクトにするために，KmeansCombiner の処理が行われます．KmeansCombiner では，それぞれのクラスタごとの入力ベクトルの成分の和を計算し，出力します．

KmeansReducer

(入力)　key,value:KmeansCombiner の出力

(出力)　key:クラスタ識別子　value:新たなクラスタの中心

複数の map から出力を受け取り，クラスタに所属するベクトルの成分を足しあわせて，新たなクラスタの中心を計算する．

KmeansClusterMapper

(入力)　key:0　value:ピクセルのベクトル

(出力)　key:ピクセルの識別子　value:最も近いクラスタ ID

処理の内容としては KmeansMapper と同じで，最後に出力される key-value が異なる．この処理によりピクセルごとのクラスタ ID がファイルに書き出される．最終的に書き出されたファイルをもとに，ピクセルの輝度値を書き換えて，減色処理を実現した．

第4章

評価と考察

4.1 評価環境

クラスタ環境で評価を行った場合，外乱が発生するため，評価結果がわかりずらくなってしまう．今回は，単純にどの程度並列に処理できるのかを評価するため，24コアマシンを用いて，コア数を変えて評価を行った．評価に使用した環境は以下の通りである．

OS	CentOS 5.5
karnel	x86_64 GNU/Linux 2.6.18
CPU	(AMD Opteron 12-core 6168 / 1.9GHz) x2
Memory	32GB x2

4.2 グレースケール化

4.2.1 評価

画像を縦に32分割し，mapタスクを32個生成し，同時に実行できるmapタスク数をコア数と同じにして実行した．reduceタスク数は1．また，24ビットカラー，1600x1200のBMPファイルを処理する画像として使用した．コア数1のときの処理のスループットつまり，処理データを実行時間で割ったものをを1として，コア数を1,2,4,8,16と変えて評価を行った．全体の処理時間のスループットを評価した結果が図4.1である．また，ピクセルに対する10000回のグレースケールの計算がどの程度コア数に応じて高速化されているか調べるため，map処理においてグレースケールを1回だけ行うもの

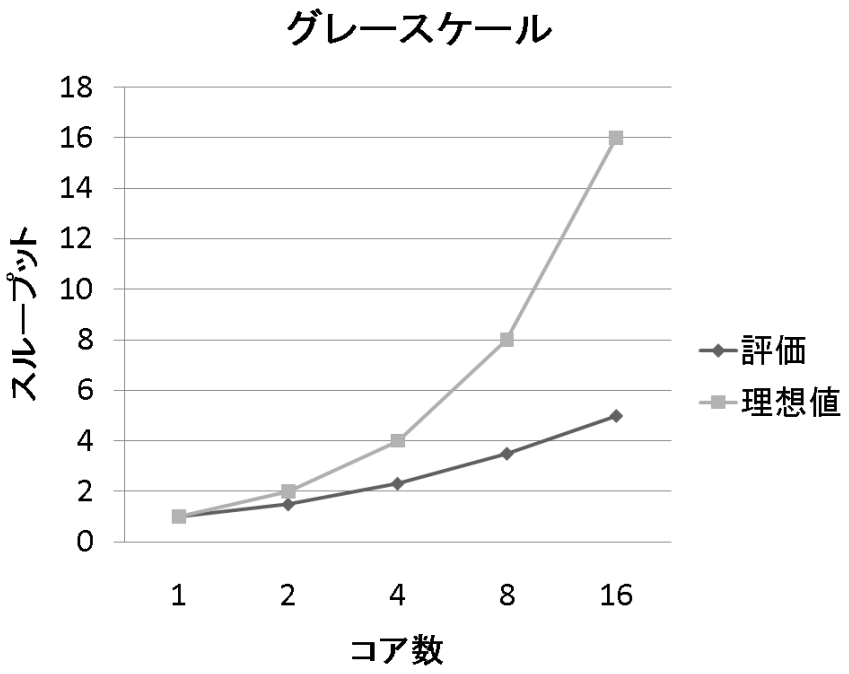

図 4.1: コア数によるスループット

と，10000 回行うものの二つの処理の差を求め，理論的にその処理時間を並列処理部分の実行時間として，コア数 1 のときを 1 として，スループットを求めたものが表 4.1，コアが 2 倍になったときの速度向上比を求めたのが表 4.2 である．

4.2.2 考察

図 4.1 から分かるように，全体の処理スループットはコア数 16 のとき，コア数 1 のときと比べて全体として 5 倍程度の速度向上となっており，理想値とはかけ離れた結果となった．このような結果になった理由として考えられるのは，ファイルを書き出している reduce 処理が並列に実行できていないためだと考えられる．そこで，reduce 処理をなくし，map タスクがそれぞれ独立に，別のファイルに出力を書き出すようにして処理を行った結果，処理時間に大した差は生まれなかった．表 4.2 を見ると，理想としてはコアが 2 倍になったとき処理スループットも 2 倍になるべきところで平均で 1.5 倍程度の向上しかしていない．処理自体は並列に処理できているのだが，複数

表 4.1: 並列処理部分のスループット

コア数	1	2	4	8	16
スループット	1.000	1.430	2.258	3.511	5.352

表 4.2: 並列処理部分の速度向上比

	2/1	4/2	8/4	16/8
速度向上比	1.430	1.579	1.555	1.524

の map タスクによるメモリアクセスのタイミングが重複することにより，共有メモリへのアクセスがボトルネックとなっているということが考えられる．

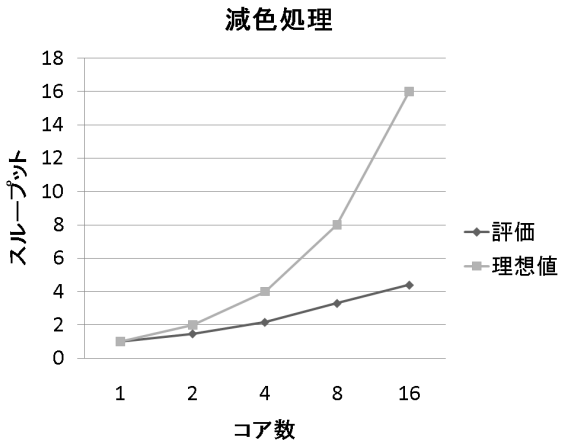

図 4.2: コア数による処理スループット

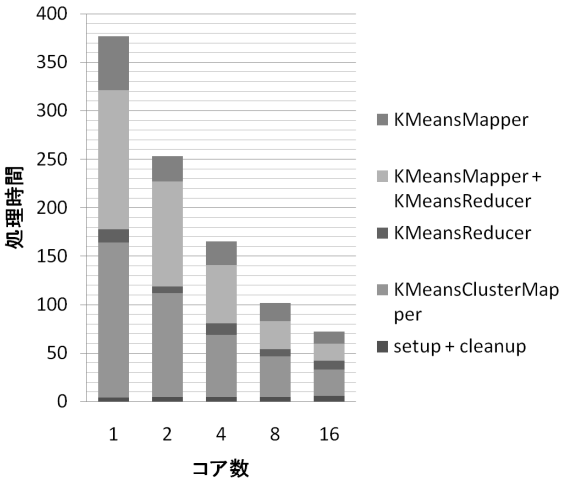

図 4.3: 処理時間の内訳

4.3 kmeans クラスタリングを用いた減色処理

4.3.1 評価

画像を，key をピクセルの ID，value をピクセルの Vector として変換し，mahout の kmeans クラスタリングのプログラムを使用し，実行した．map タスク数は〜，reduce タスク数は 1 とした．また，kmeans 法における繰り返し回数を 1 とした．また，24 ビットカラー，1600x1200 の BMP ファイルを処理する画像として使用した．コア数 1 のときの処理のスループットを 1 として，コア数を 1,2,4,8,16 と変えて評価を行ったものが図 4.2 である．また，それぞれの処理時間の内訳は，図 4.3 である．また，並列実行部分のみの処理時間の速度向上比を示したのが，表 4.3 である．

4.3.2 考察

図 4.2 から分かるように，全体の処理スループットはコア数 16 のとき，コア数 1 のときと比べて全体として 4.4 倍程度の速度向上となっている．評価結果として大きな効果が得られなかった原因として，タスクの生成やファイルへのアクセスなど，並列に処理できていない部分が全体の処理時間に対して，比較的多くの部分を占めている

表 4.3: 並列処理部分の速度向上比

	1	2	4	8	16
KMeansMapper	1.000	1.498	2.386	4.125	6.600
		1.498	1.593	1.729	1.600
KMeansClusterMapper	1.000	1.495	2.500	3.810	5.926
		1.495	1.672	1.524	1.555

からではないかと考えられる．しかし，処理時間の内訳を表した図 4.3 を見ると，タスクの生成や，reduce 処理は処理全体の数%程度でしかない．並列処理部分のみの速度向上比の評価である表 4.1 において，KMeansMapper，KMeansClusterMapper それぞれにおいて，コア数が 2 倍になっても，処理時間自体は 1.5〜1.6 倍程度しか速度向上していないことが分かる．この原因として，グレースケール処理と同じように，共有メモリへのアクセスがボトルネックとなっていることが考えられる．

第5章

まとめと今後の課題

本論文ではまず，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，それを解消するために FileInputFormat の改良を行った．そして改良した FileInputFormat を用いて，画像のグレースケール化の処理を実装し，評価を行った．また，mahout のクラスタリングの実装を用いて画像の減色処理を実装し，評価を行った．

今後の課題として，速度向上の原因と考えられる部分を改善し，スケールさせることが必要だと考えられる．また，様々な画像処理を実装しライブラリとして提供し，最終的に並列・分散を意識しない画像処理言語を Hadoop 上に作成することが課題となる．

謝辞

本研究のために、多大なご尽力を頂き、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、松井俊浩助教に深く感謝いたします。また、本研究の際に多くの助言、協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝いたします。

参考文献

[1] Message Passing Interface Forum，“A Message-Passing Interface Standard”， 1995

[2] OpenMP，http://openmp.org/wp/

[3] Jeffrey Dean and Sanjay Ghemawat，“MapReduce:Simplified Data Processing on Large Clusters”， 2004

[4] ApacheHadoopProject，http://hadoop.apache.org/

[5] ApacheHadoopProject:HadoopDistributedFileSystem http://hadoop.apache.org/hdfs/

[6] ApacheHadoopProject:HadoopMapReduce http://hadoop.apache.org/mapreduce/

[7] ApacheHadoopProject:HadoopMapReduce http://mahout.apache.org/