情報処理学会論文誌 Vol.53 No.3 1189–1203 (Mar. 2012)

MapReduceプログラミングモデルに即したReduceフェーズのData Skew動的緩和方法

中山 誠[1,a)] 山崎 憲一[2] 田中 聡[1]

受付日 2011年8月14日, 採録日 2011年12月16日

概要：MapReduceでは，Mapフェーズから出力された中間データ（Key-Valueペア）がそのKeyによってグループ化され，担当するReduceタスクが決定される．各中間Keyの出現頻度にばらつきがある場合，各Reduceタスクが担当する中間データの量がばらつき，Data Skewの発生原因となる．本論文では，MapReduceプログラミングモデルに即した，MapReduceクラスタの規模によらず適用可能な，Reduceフェーズの Data Skewを緩和する方法を提案し，その評価結果とともに示す．

キーワード：MapReduce，Data Skew，Big Data，負荷分散，並列処理

Alleviation Technique for Data Skew of Reduce Phase with MapReduce Programming Model

Makoto Nakayama[1,a)] Kenichi Yamazaki[2] Satoshi Tanaka[1]

Received: August 14, 2011, Accepted: December 16, 2011

Abstract: In MapReduce, Map phase outputs intermediate key-value pairs which are grouped based on their keys, and each group is assigned to one of Reduce tasks. If occurrence frequency of each intermediate key is skewed, the amount of data to be processed on each Reduce task can also be skewed. This is called as "data skew". In this paper, we propose a new technique which is compliant with the MapReduce programming-model and applicable regardless of MapReduce cluster size to alleviate such data skew of Reduce phase.

Keywords: MapReduce, Data Skew, Big Data, load balancing, parallel processing

1. はじめに

近年，企業や研究機関で取り扱われるデータ量は増加の一途をたどり，1台の計算機で扱える範囲をとうに超えている．それに対抗する一手段として，安価な計算機を大量に調達し，それらを協調動作させる並列分散処理が採用されてきている．そのような並列分散処理を実現するフレームワークとして，特にMapReduce [1] が近年大きな注目を集めている．

[1] NTTドコモ先進技術研究所
Research Laboratories, NTT DOCOMO, INC., Yokosuka, Kanagawa 239–8536, Japan
[2] 芝浦工業大学
Shibaura Institute of Technology, Minato, Tokyo 108–8548, Japan
[a)] nakayamamak@nttdocomo.co.jp

MapReduceでは，1つのJob（アプリケーション）はMapとReduceの2つのフェーズからなり，各フェーズで分散実行される個々のプロセスを各々 "Mapタスク"，"Reduceタスク" と呼ぶ．また各フェーズで実行されるプログラム（以下，各々 "Mapper"，"Reducer" と呼ぶ）は，Jobの目的に応じてその処理内容を自由に記述できる．以下，MapperとReducerを総称する場合，"MapReduceプログラム" と呼ぶ．Jobが起動されると，MapReduceフレームワークは入力データを一定のブロックサイズに分割（以下，"split" と呼ぶ）し，各splitを入力として個々のMapタスクを起動する．起動されるMapタスク数は生成されたsplit数に等しいため，入力データの総サイズがどんなに大きくても，各Mapタスクに割り当てられるデータサイズはほぼ一定である．各Mapタスクは割り当てられ

た split からデータを読み込み，処理を施した後，中間データとして KeyValue ペア（以下，“中間レコード” と呼ぶ）を出力する．同じ中間 Key を共有する中間レコードは 1 つにまとめられ（以下，“グループ” と呼ぶ），Reduce フェーズでは 1 つのグループは 1 つの Reduce タスクによって処理される．このとき，より多くの中間レコードが同じ中間 Key を共有するほど，そのグループのサイズは大きくなる．Reduce タスク数は Job 開始時に固定され，各 Reduce タスクにはユニークな番号（以下，“Partition” と呼ぶ）が与えられる．中間 Key によって識別されるグループごとに Partition を一意に決定することで，各グループを担当する Reduce タスクを決定する．この処理を “パーティショニング” という．各 Reduce タスクは，割り当てられたすべてのグループを受け取り，処理を施した後，最終データを出力する．

MapReduce では，すべての Reduce タスクの処理が完了した時点で Job が完了となる．したがって，一部の Reduce タスクに他よりも多くの中間データが集まった場合（この状態のことを，本論文では Reduce フェーズの “Data Skew” と呼ぶ），その Reduce タスクが完了するまでの時間が長くなり，結果として Job の完了時間が長くなる．この問題はすでに知られており，文献 [2] 等で指摘されている．Reduce フェーズで Data Skew が発生する理由としては，(I) 一部の Partition に他より多くのグループが割り当てられる，(II) 一部のグループのサイズが他と比べて極端に大きい，が考えられる．(I) はパーティショニングの問題であり，その方法を工夫することで解決できる場合がある．(I) は本論文の検討の対象外である．一方，(II) はパーティショニングの工夫では解決できない．(II) については 2 章で詳細に述べる．

本論文では，一部のグループのサイズが極端に大きいことに起因する Reduce フェーズの Data Skew を緩和し，Job 完了時間を短縮する方法について述べる．

2 章では，本論文の検討対象となる Reduce フェーズの Data Skew に関して，既存の取り組みとその問題点を交えて述べる．3 章で提案方法の適用対象を整理した後，4 章で Reduce フェーズの Data Skew を解決するための提案方法とそのプロトタイプシステムについて述べる．5 章で定量評価の結果をもって提案方法の効果を示す．6 章で関連研究との比較を行い，7 章をむすびとする．

2. Reduce フェーズの Data Skew

MapReduce における各 Reduce タスクは，割り当てられたすべてのグループを処理する．すべての Reduce タスクがほぼ同数のグループを割り当てられたと仮定した場合，大きいサイズのグループを多く割り当てられた Reduce タスクほど，処理するデータ量が増えることになり，ここに Data Skew が発生する．一般に，グループ間のサイズのばらつき（以下，“サイズ差異”）が大きいほど，Data Skew が発生しやすくなる．この問題はすでに認識されており，いくつかの先行研究が存在する．

2.1 グループの分割

グループ間のサイズ差異を低減させる有効な方法の 1 つは，サイズの大きなグループを分割（中間 Key を変更）し，その粒度を細かくすることである．たとえば，ある Job において日本国内の地域別の分析を行う場合，中間 Key を都道府県単位とするよりも，市町村単位に変更したほうが，グループ間のサイズ差異を低減させることができる可能性が大きい．グループの分割に関しては，手動 [3] もしくは自動 [4] で行う方法がある．

文献 [3] は，Zipf 分布に従うデータにおける各要素の出現頻度の違いにより生じるグループ間のサイズ差異によって，一部の Reduce タスクが過負荷になる問題に言及している．この問題の解決には，場合によっては MapReduce プログラム全体のアルゴリズムの見直し（中間 Key の選び方の変更）が必要であると述べている．しかしながら，そのようなプログラムの作り直しは手間が大きい．また，MapReduce プログラムの処理内容や入力データによっては，どのような中間 Key の選び方をすれば十分な効果が得られるのかを，MapReduce プログラムの実装者（以下，“プログラマ”）が事前に把握することは困難な場合も考えられる．

文献 [4] は，入力（多次元空間上に配置される点の集まり）をクラスタリングし，個々のクラスタを 1 つのグループとして Partition に割り当てる MapReduce プログラムに特化して論じている．1 つのクラスタ（グループ）による負荷が大きすぎると判断した場合（この判断には，ユーザが別途用意する “コスト関数” を利用する），フレームワークがそれを自動的に再分割する．ところで，MapReduce プログラムの処理内容は自由度が高いため，Mapper が出力した中間 Key を意図的に利用する Reducer や，前段 Job の最終データに含めた中間 Key の内容を意図的に利用する後段 Job を記述することも可能である．この場合，中間 Key の変更をともなうグループ再分割が自動的に行われると，Reducer や後段 Job が意図したとおりに動作しない状況に陥る可能性がある．

以上から，Job 実行中にグループ間のサイズ差異を検出し，中間 Key の変更をともなわずにグループを自動的に再分割する方法が求められる．

2.2 Data Skew の検出方法

文献 [3], [4] に共通するもう 1 つの問題は，Reduce フェーズの Data Skew をどのように検出するかである．文献 [4] では，入力データと同様の特性を持つ（ただしデータサイズは十分小さい）サンプルデータを事前に分析して，Data

Skewの発生を予測する．ここでは，Jobの実行者がサンプルデータをあらかじめ用意することを前提としているが，そのようなものをつねに用意できるとは限らない．文献[3]でも，サンプルデータを事前分析する点は同じだが，サンプルデータの入手方法として“入力データのサンプリング抽出”をあげている．

たとえば，Jobへの入力データの各レコードに，中間Keyそのものが含まれている場合，入力データのサンプルを事前分析する方法は簡単かつ有効である．しかしMapperの処理内容によっては，処理中に動的に生成したものを中間Keyとして使用する．そのようなMapper（3.2節に例を示す）を用いる場合，サンプルデータの事前分析処理をMapperの処理内容に応じて作り直す手間が生じる．また，Mapperの処理内容によっては，本来の入力データとサンプルデータとではその振舞いが変化し，本来の中間データにおけるばらつきを正確に予測できない場合も考えられる．したがって，Data Skewの発生予測に対し，サンプルデータの事前分析という手段がつねに簡単かつ有効なわけではない．

また，入力データのサンプリング抽出をどのように行うかも課題となる．たとえば，MapReduceのオープンソース実装であるHadoop[5]では，InputSampler（文献[6]のp.225）機構が提供されている．InputSamplerでは，Mapフェーズに先んじて入力データからのサンプリング抽出を行うが，この抽出処理はMapReduceクラスタ（以下，“クラスタ”と呼ぶ）全体で並列分散実行されず，単一ノード上で行われる．抽出方法のアルゴリズムにもよるが，このサンプリング抽出には一般に，入力データのサイズに比例した時間を要する．MapReduceはそもそも大量のデータを対象とするため，サンプリング抽出の所要時間は無視できないものとなり，オーバヘッドとしてJob完了時間にそのまま加算されるため，問題となる．

さらに，Reduceフェーズの Data Skewが発生しない入力データが与えられる場合もある．その場合，Data Skew検出のオーバヘッドがそのままJob完了時間に加算される一方，Data Skew緩和による効果は発生しない．したがって，Data Skewの程度の多寡にかかわらず適用可能な技術であるためには，オーバヘッドの小さいData Skewの検出方法が必要となる．

以上から，Mapperの処理内容に依存せず，かつ，大量の入力データが与えられた場合でもオーバヘッドの小さい，Data Skewの検出方法が求められる．

3. 提案方法の適用対象

3.1 パーティショニングのカテゴリ

MapReduceではパーティショニング方法のカスタマイズが可能であり，Jobの処理内容に応じた方法を採用できる．我々は提案方法の検討にあたり，様々なパーティショニング方法を以下の3つのカテゴリに分類した．

(A) グループとPartitionの間に関連がある．
(B) グループとPartitionの間に関連はないが，グループ内の中間レコード間に関連がある．
(C) グループとPartitionの間に関連はなく，グループ内の中間レコード間にも関連がない．

(A)に該当するJobでは，特定の中間Keyを持つグループを特定のPartitionに関連付ける．したがって，フレームワークがグループとPartitionの関連付けを変更することは許されない．(A)の具体例はTotal Sort（文献[6]のp.223）である．これはJobの入力データ全体をソートして出力する処理であり，序列の若い中間Keyほど，若いPartitionに割り当てる．したがって，グループとPartitionの関連付けを変更すると，その部分でソートの順序が破綻する．

(B)に該当するJobでは，特定の中間Keyを持つグループがどのPartitionに割り当てられてもよいが，Job固有の理由により，グループ内の中間レコード間には何らかの結び付きがある．Job固有の理由に適う場合に限り，1つのグループを複数のPartitionに分割することが許されるが，その方法もまたJobごとに異なる．(B)の具体例は文献[4]である．

(C)に該当するJobでは，特定の中間Keyを持つグループがどのPartitionに割り当てられてもよく，グループ内の中間レコード間に何の結び付きもない．したがって，1つのグループを，その中間Keyを変更せずに複数のPartitionに分割することが許される．この場合，同じ中間Keyを全Partitionに均等分割するパーティショニング方法（以下，“単純分割法”）を用いることも可能である．ところでMapReduceでは，2つ以上の入力データから同じ中間Keyを共有する中間レコードを同じPartitionに集める，Reduce-Side Joins（文献[6]のp.235）テクニックがよく用いられる．このとき，一部の入力データから発生した中間データが(C)に該当する場合がある．そのような具体例を3.2節で紹介する（また，RDBのInner Join演算に相当する処理も，そのような事例の1つである）．しかしながら，(C)に該当する一部の中間データに単純分割法を適用すると，残りの中間データを全Partitionに複製する必要が発生し，4.1節で述べる問題が発生する．

提案方法は，Reduce-Side Joinsの一部の中間データが(C)に該当するJob（MapReduceプログラム）を対象とする．ただし，それぞれのMapReduceプログラムがこの条件に該当するかを自動判別することはできない．したがって，その判断はプログラマによってなされる必要があり，提案方法を適用するか否かを4.6節に従ってプログラム内で指定する必要がある．

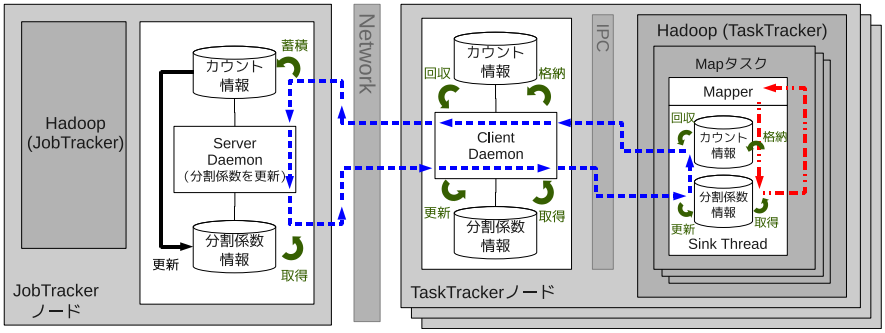


図 2 アーキテクチャの概要
Fig. 2 Overview of architecture.

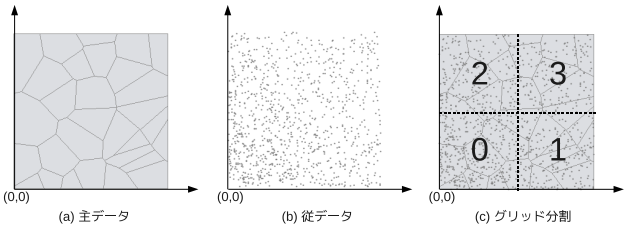


図 1 Reduce-Side Joins の具体例
Fig. 1 An Example of Reduce-Side Joins.

3.2 Reduce-Side Joins の具体例

Reduce-Side Joins を用いると，主/従データ（後者のほうが前者よりもデータサイズが大きいものとする）から発生した対応する中間データどうしを，同じ Partition に集めて処理できる．たとえば，複数の領域を主データ（**図 1** (a)），複数の点を従データ（図 1 (b)）とし，各点を包含する領域を探索する処理を考える．Reduce-Side Joins で実現するには，Map フェーズで図 1 (c) のように定義域を等分割（以下，“グリッド” と呼ぶ）し，グリッドに割り当てた識別子（0〜3）を中間 Key として，各グリッドに属する領域および点をグループ化する方法が考えられる．このとき，複数のグリッドに跨る領域は複製して各グループに含める．Reduce タスクは，グループ内の領域と点との間で包含関係の探索を行えばよい．グループ内の点どうしには何の結び付きもないため，点の中間データはカテゴリ (C) に該当し，複数の Partition へ分割することが許される．ただし点の中間データを複数の Partition へ分割した場合，対応する領域の中間データを分割先 Partition へ複製する必要が生じる．以上の方法を実現するためのプログラミングモデルについては，4.6 節に示す．

4. Data Skew の動的緩和方法

本章では，Reduce フェーズの Data Skew を緩和するための提案方法を説明する．提案方法のプロトタイプシステム（以下，提案システム）は，MapReduce のオープンソース実装である Hadoop [5] 上に構築した．また Job への入力データは，Hadoop のデフォルトのストレージシステムである HDFS [7] 上に用意されているものとする．

本論文では簡略化のため，クラスタ上で一時に 1 つの Job だけが実行されるものとする．つまり，クラスタ全体で一時に実行できる Map タスクの最大数（Map タスクプール）を M_{pool} とすると，実行中 Job のある時点での Map タスク数は $M_{cur} = min$（未完了 Map タスク数，M_{pool}）となる．複数 Job の同時実行に対応するには，M_{cur} の算出方法と，カウント情報/分割係数情報（いずれも後述）を Job ごとに識別する仕組みを導入すればよい．提案システムの概要を**図 2** に示す．図中の灰色は既存部分を，白色は提案方法の構成部分を表す．ClientDaemon（以下，CD），ServerDaemon（以下，SD），SinkThread（以下，ST）が提案方法の主要部分である．ここでは SD を，TaskTracker を統括する JobTracker ノード上に記載したが，他のノード上に配置してもよい．

4.1 グループ分割方法の検討

3.2 節の処理において，従データに単純分割法を適用すると，主データの全内容も全 Partition へ複製する必要が発生し，その分だけ中間データが増加する．グループ間のサイズ差異がまったくない従データであった場合，中間データの増加というデメリットだけが発生する．また MapReduce では，1 つの split は 1 つの Map タスクが担当する．たとえば Partition 総数が 100 である場合，64 MByte の主データの split を担当する Map タスクには，複製によって 6 GByte 余りの出力が集中してしまう．提案方法では，Map フェーズの最中にグループ間のサイズ差異を検出し，大きなグループだけをその大きさの程度に応じて分割することで，中間データの増加と，特定 Map タスクへの出力の集中を抑えるようにする．

従来の MapReduce では，中間レコードを出力するごとにその Partition を決定（パーティショニング）し，各 Map タスクが完了するたびに，当該 Map タスクが出力した中間データを Map フェーズとオーバラップして Reduce タスクへ送達する．したがって，中間レコードの Partition 決定を保留して全 Map タスクの中間データをいったん出力し，中

```
事前条件： Q を指定 (1 ≥ Q > 0)，B = ブロックサイズ，カウント情報をゼロに初期化
事前条件： 処理済み入力データサイズ s_in，出力済み中間データサイズ s_out をゼロに初期化
 1: Map タスク起動時に CD が保持している分割係数情報を取得
 2: while Map フェーズ中 do
 3:    入力レコードを 1 件処理し，パーティショニング (4.5 節) を行い，中間レコードを出力
 4:    カウント情報を更新，s_in および s_out を更新， 入力データ伸縮率 φ = s_out / s_in を算出
 5:    if φBQ ≤ (現在の s_out − 前回格納時の s_out) then
 6:       カウント情報を ST へ格納し，ST から分割係数情報を取得
 7:       Mapper のカウント情報をゼロにリセット
 8:    end if
 9: end while
10: if 未格納のカウント情報がある then
11:    カウント情報を ST へ格納
12: end if
```

図 3 Mapper によるカウント処理
Fig. 3 Size counting algorithm of Mapper.

間 Key の出現頻度のばらつきを把握してから Partition を決定すると，中間データの Reduce タスクへの送達を Map フェーズとオーバラップして行えず，Job の処理効率が低下する．提案方法では，Partition 決定を保留せず，Map フェーズ序盤では従来の方法で Partition を決定して中間レコードを出力し，あるグループのサイズが大きいことを検出した時点で当該グループの Partition を変更し，以降の中間レコードを出力するようにする．

提案方法は，(i) MapReduce プログラミングモデルに即し，(ii) MapReduce クラスタの規模によらず適用可能で，(iii) Reduce フェーズの Data Skew を動的に検出/緩和する，ことを特徴とする．(i) については 4.6 節で，(ii) については 4.4.1 項で，(iii) については 4.2 節〜4.5 節にかけて，それぞれ詳細を述べる．

4.2 Mapper によるカウント情報の収集

Map フェーズの処理は各 TaskTracker 上で独立に実行される Map タスクで行われる．入力データ全体がブロックサイズ（“B” で表す）ごとの split に分割され，個々の Map タスクへの入力となる．Map タスクが出力する中間データに関し，Job 全体におけるグループごとのサイズを把握するには，各 Mapper でサイズをカウントし，それらを集計する必要がある．各 Mapper がカウントする情報（以下，“カウント情報”）は次の 3 つである．

- 処理済みの入力データのサイズ（以下，“入力カウント”）
- 出力済みの中間データのサイズ（以下，“出力カウント”）
- グループごとの中間データのサイズ（以下，グループごとの “カウント量”）

Mapper は**図 3** に従い，カウント情報の収集と ST への格納（図 2 右側の破線ループ：以下，“格納ループ”），および，ST からの “分割係数情報”（どのグループを何分割するかを表す情報）の取得を繰り返す．分割係数情報はその後のパーティショニングに利用（4.5 節）する．ここで，カウント情報を ST へ格納するタイミングを決定するために，独自のパラメータとして格納レート Q を導入する（Q の値は，プログラマや Job の実行者が Job 開始時に指定する必要がある）．Q と入力データ処理率 $\frac{s_{in}}{B}$ の比較ではなく，図 3 の 5 行目の条件式を用いることで，中間データが出力されない（入力データを読み飛ばす）場合に無駄な格納契機が発生しなくなる．Q は 0 から 1 の間の値をとり，値が小さいほどカウント情報が頻繁に ST へ格納されることになり，処理のオーバヘッドが増加する．反対に大きいほど ST から分割係数情報を取得する回数が減り，パーティショニングへ反映（4.5 節）するタイミングが遅くなる．Q の変動が提案方法に及ぼす影響については 5 章で検証する．

4.3 CD によるカウント情報と分割係数情報の中継

提案方法では，すべての TaskTracker 上で CD を動作させる必要がある．CD は**図 4** に従い，SD との間でカウント情報の報告と分割係数情報の取得（図 2 左側の破線ループ：以下，“報告ループ”），および，取得した分割係数情報の ST への配布を繰り返す．カウント情報を載せた，CD から SD へ送信される信号を “報告メッセージ” と呼ぶ．

図 2 の 2 つのループは互いに非同期に動作することに注意されたい．初期の提案システムでは 2 つのループを同期させていたが，同期待ちによる Mapper の処理効率低下が著しかったため，非同期で動作するように変更した．しかし，これにより新たな問題が発生する．Mapper が ST から取得する分割係数情報には，そのときに ST へ格納したカウント情報は反映されていない．またその時点では，SD による最新の分割係数情報が ST へ未到達の可能性もある．本問題への対応については 4.4.1 項で述べる．なお，報告

事前条件： CD のカウント情報と分割係数情報をゼロに初期化
1: **loop**
2: 同ノード上の全 ST からカウント情報を回収し，集計
3: **if** SD へ報告すべきカウント情報がある **then**
4: 集計済みのカウント情報を SD へ送信 (CD および ST のカウント情報はゼロにリセット)
5: 応答として SD から分割係数情報を受信し，CD 内に保存
6: 同ノード上の全 ST へ分割係数情報を配布
7: **end if**
8: **end loop**

図 4 CD による中継処理
Fig. 4 Information relaying algorithm of CD.

表 1 記号の定義
Table 1 Definition of symbols.

グループの平均サイズ	a	実行中 Map タスク数	M_{cur}	ブロックサイズ	B
グループの予測最終平均サイズ	A	Map タスクプール	M_{pool}	グループ総数	N
処理済み入力データサイズ	s_{in}	グループ g のサイズ	s_g	Partition 総数	R
出力済み中間データサイズ	s_{out}	遅延カウント量	f_g	TaskTracker 数	T
Job の総入力データサイズ	S_{In}	予測残りカウント量	e_g	格納レート	Q
Job の総中間データサイズ	S_{Out}	積算カウント量	d_g	閾値	U
メッセージあたりの出力カウント	C_{msg}	n 回目の分割係数	$x_{g,n}$	回転比	Z
メッセージあたりの Map タスク数	M_{msg}	Map フェーズ進捗率	p	入力データ伸縮率	ϕ

ループは基本的にベストエフォートで回り続けるが，報告すべきカウント情報がない場合は，CD は報告すべきカウント情報が発生するまで報告ループを一時的に停止させ，無駄な通信の発生を抑える．

4.4 SD によるグループの分割

SD の役割は，報告メッセージのカウント情報を集計（4.4.1 項）し，(α) 分割するグループと，(β) その分割係数，を決定（4.4.2 項）することである．図 2 の 2 つのループは非同期に動作するため，Map タスクのカウント情報が SD へ到達するまで，および，SD で決定した分割係数情報が実行中の全 Map タスクに行きわたるまでに，タイムラグが発生する．タイムラグの間，各 Map タスクは古い分割係数情報に従ってパーティショニングを行うことになる．ところで SD は，TaskTracker と同数の CD からの報告メッセージを処理する必要があるため，十分な能力（CPU，メモリ，NW 帯域）を有するべきである．しかし規模の大きい（TaskTracker が多い）クラスタを用いるほど SD にかかる負荷も比例して大きくなり，報告メッセージへの応答時間（タイムラグ）が伸びる．したがって (α) や (β) の処理では，このタイムラグを考慮すべきである．

本節で述べる方法は，2.2 節の課題も解決する．Map タスクが実際に入出力したデータのカウント情報を用いるため，Mapper の処理内容に依存せずにグループ間のサイズ差異を検出できる．また，入出力データのカウント処理を全 Map タスクで分担するため，単一ノードでサンプリング抽出する方法と比べ，より大量の入力データに対応できる．

以降の説明で用いる記号を，**表 1** にまとめる．

4.4.1 クラスタ規模とタイムラグを考慮したカウント情報の集計

SD は**図 5** に従い，報告メッセージに含まれるカウント情報の集計（および，その後の (α)，(β) の処理）を行う．報告メッセージ 1 件に含まれるカウント情報の生成に寄与した Map タスク数は，$M_{msg} = max(\frac{M_{cur}}{T}, 1)$ となる．1 と比較する理由は，報告すべきカウント情報がない（実行中の Map タスクが当該ノードに 1 つも存在しない）場合，CD は報告メッセージを生成しない（4.3 節）ためである．

ここで，タイムラグの期間中，Map タスクが古い分割係数情報を用いてパーティショニングを行う間にカウントされる，グループ g のカウント量 f_g（“遅延カウント量” と呼ぶ）を考える．クラスタ全体では一時に T 件の報告メッセージが図 2 の報告ループ上にある．そのうちの 1 件を契機とした (α) 処理によりグループ g が分割されたとする．直後の (β) 処理により分割係数情報が更新されるが，他の $(T-1)$ 件の報告メッセージは当該分割係数情報を反映する前のものである．さらに，各 Map タスクは報告ループとは非同期に処理を継続しているため，次の T 件の報告メッセージにも当該分割係数情報が反映されないことになる．よってタイムラグで考慮すべき報告メッセージ数は $(2T-1)$ 件である．ただし f_g では，報告メッセージ数ではなく，そこに含まれるカウント情報の生成に寄与した Map タスク数を基準とするため，代わりに $(2M_{cur} - M_{msg})$ を用いる．以上をふまえて f_g は，

事前条件： S_{In}，R，T を Hadoop フレームワークから取得， s_{in}，s_{out}，s_g をゼロに初期化
1: **loop**
2: 報告メッセージを受信し，そのカウント情報から s_{in}，s_{out}，s_g を集計，C_{msg} を取得
3: N = 出現済みのグループの総数
4: $s_{out} = \sum_g^N s_g$， $a = \frac{1}{N} \times \sum_g^N s_g = \frac{s_{out}}{N}$， $p = \frac{s_{in}}{S_{In}}$ が成立， $\phi = \frac{s_{out}}{s_{in}}$ を算出
5: $S_{Out} = \frac{s_{out}}{p} = S_{In} \times \frac{s_{out}}{s_{in}} = \phi S_{In}$， $A = \frac{S_{Out}}{N} = \frac{s_{out}}{p \times N} = \frac{a}{p}$ を算出
6: M_{cur} を更新し，$M_{msg} = max(\frac{M_{cur}}{T}, 1)$ を算出
7: Z，および，全てのグループ g について f_g，e_g を算出
8: (α) グループ分割判断，および，(β) 分割係数情報の更新 (4.4.2 項)
9: 更新された分割係数情報を報告メッセージへの応答として返信
10: **end loop**

図 5 SD によるカウント情報の集計
Fig. 5 Count-information aggregation algorithm of SD.

$$f_g = Z(2M_{cur} - M_{msg}) \times \phi QB \times \frac{s_g}{s_{out}} \quad (1)$$

となる．Z（“回転比” と呼ぶ）は，1 件の報告メッセージが，各 Map タスクからのカウント情報を平均で何格納回数分含んでいるか，いい換えると，CD の報告ループが 1 回転する間に各 Map タスクの格納ループが平均で何回転するかを表す．Z の算出方法は後述する．したがって $Z(2M_{cur} - M_{msg})$ は，古い分割係数情報を用いてパーティショニングを行う間に Mapper が ST へカウント情報を格納する，クラスタ全体での総回数となる．そこへ，Mapper が ST へ 1 度に格納する出力カウント（ϕQB）と，現在までの出力カウントに対してグループ g が占める割合（$\frac{s_g}{s_{out}}$）を乗じる．M_{cur} はクラスタ規模に比例し，次に述べる Z もクラスタ規模に応じて変化するため，式 (1) によってクラスタ規模に応じた遅延カウント量を算出できる．提案方法では，(α) および (β) の処理に遅延カウント量 f_g を利用し，クラスタ規模に応じたタイムラグの影響を考慮する．

回転比 Z は，報告メッセージ 1 件に含まれる出力カウント C_{msg} を用いて，

$$Z = \frac{C_{msg}}{M_{msg}} \times \frac{1}{\phi QB} \quad (2)$$

で算出できる．$\frac{C_{msg}}{M_{msg}}$ は，1 件の報告メッセージにおける，Map タスクあたりの出力カウントの総和である．それを Mapper が 1 度に ST へ格納する出力カウント（ϕQB）で除する．ところで，クラスタ全体には T 個の CD が存在する．CD のノードの処理能力や，SD のノードとの間の通信経路の差異により，C_{msg} は送信元 CD によって変化しうる．そこで実際には，最新の T 個（各 CD から平均 1 つの C_{msg} を採用）の式 (2) 算出値の平均を Z として使用する．クラスタ規模が変化すると SD が応答すべき CD 数も変化し，各 CD から見た報告ループの応答時間（つまり，報告ループの回転周期）も変化するが，各ノードで実行される Map タスクの実行速度（つまり，格納ループの回転周期）はクラスタ規模には影響されない．したがって C_{msg} がクラスタ規模に応じて変化し，Z はそれに比例する．

さらに SD はグループ g について，(α) 処理による分割後から Map フェーズ終了までに報告されてくると予測される残りのカウント量を $e_g = (\frac{1}{p} - 1)s_g - f_g$ と予測する．ここで，遅延カウント量はグループ分割前に属するデータサイズであるため，f_g を減じている．

4.4.2 グループ分割判断と分割係数決定

前項をふまえ，(α) および (β) の処理を説明する．以降では，分割済みのグループを除いた，未分割のグループのみに関する予測最終平均サイズを A' で表す．また，いくつかの記号に関し，進捗率 p における値を A_p や $s_{g,p}$ と表す．分割係数 $x_{g,n}$ は，グループ g に関する n 回目の分割判断時に決定される整数値であり，グループ g を何分割すべきかを表す．d_g は，グループ g が分割された時点からの，グループ g のカウント量の積算である．

さらに提案方法の独自パラメータとして，“Reduce フェーズの処理時間の観点から，無視できる程度のグループサイズの差異” を表す，閾値 U を導入する．(α) 処理では，平均サイズを超過したグループを分割すると判断する．たとえば，グループ総数が非常に多く，グループ間のサイズ差異も小さい場合，全グループのサイズがほぼ一定となり，そもそも分割を行う必要がない．そのような状況で，平均サイズをわずかに超過したグループをいちいち分割するのは無駄である．そこで提案方法では，平均サイズに U を加算したものを分割判断の基準とし，瑣末なサイズ超過を無視する．また (β) 処理では，分割判断の下ったグループが A' 程度の大きさに分割されるように分割係数を決定するが，A' が極端に小さい場合，分割係数が過度に増え，Reduce-Side Joins における主データの複製数が増えてしまう．そこで提案方法では，グループが $A' + U$ の大きさ（Reduce フェーズの処理時間の観点からは $A' \simeq A' + U$ と見なせる）に分割されるように分割係数を決定する．以上から，U の値は小さければよいというものではなく，過度な分割係数の増加を抑制するためにはそれなりに大きい値を設定すべきであるが，反対に大きすぎても，グループの分割判断が下りにくくなるため問題となる．しかしなが

事前条件： U を Job 開始時に指定，最初は全てのグループを “未分割グループ” として扱う
1: **for all** グループ g **in** 分割済みグループ **do** /* 分割更新判断 */
2: 報告メッセージのカウント情報から，グループ g のカウント量を d_g に加算
3: **if** グループ g について，式 (5) による分割更新判断が成立 **then**
4: 式 (4) にてグループ g の新たな分割係数 $x_{g,n}$ を決定，d_g を $-f_{g,p}$ にリセット
5: **end if**
6: **end for**
7: **for all** グループ g **in** 未分割グループ **do** /* 分割開始判断 */
8: **if** グループ g について，式 (3) による分割開始判断が成立 **then**
9: グループ g を除外して A' を更新し，次回から “分割済みグループ” として扱う
10: 式 (4) にてグループ g の最初の分割係数 $x_{g,n}$ を決定，d_g を $-f_{g,p}$ に初期化
11: **end if**
12: **end for**

図 6 SD によるグループ分割判断と分割係数決定
Fig. 6 Group division and division factor calculation algorithm of SD.

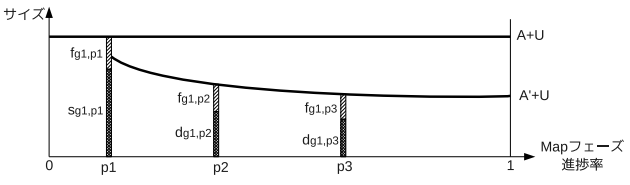


図 7 グループ分割の判断過程
Fig. 7 Group division deciding process.

ら Reducer の処理内容に依存するため，U の適切な値をここで一意に決定することはできない．よって提案方法では，従来の MapReduce における R と同様に，プログラマや Job の実行者が Job 開始時に U の値を指定する必要がある．

SD は**図 6** に従い，(α) および (β) の処理を行う．提案方法の (α) 処理には，2 つの判断方法がある．1 つは “分割開始判断” であり，未分割のグループを初めて分割するかどうかを判断する．判断の基準には A を用い，全グループの平均サイズを超過したグループに分割判断を下す．もう 1 つは “分割更新判断” であり，分割済みグループの分割係数を更新する必要があるかどうかを判断する．判断の基準には A' を用い，グループ分割後の各部分のサイズが，その時点での未分割グループの平均サイズ以下になるように判断を下す．

以上をふまえ，**図 7** に沿って説明する．未分割のグループ $g1$ および進捗率 $p1$ において，一般式 (3) が成立（ただし上述のとおり A_p に U を加える）した場合，SD はグループ $g1$ に分割開始判断を下す．

$$s_{g,p} + f_{g,p} > min\Big(A_p + U, \frac{S_{Out}}{R}\Big) \tag{3}$$

$\frac{S_{Out}}{R}$ と比較する理由は，グループ総数 N が異常に少ない状況（Map フェーズ序盤で特定グループのみ出力される，グループが本当に少数しか存在しない，等）への “特別対処” である．本対処がないと最悪の場合（たとえば $N = 1$），分割開始判断が 1 度も成立しないこともありうるが，本対処の導入により少なくとも，総中間データサイズの Partition あたりの平均値において分割開始判断が成立する．以降で出現する $\frac{S_{Out}}{R}$ はすべて同じ理由である．ここでは $A_p + U$ が選ばれたものとする．この時点でグループ $g1$ を除外し，A' を更新する．他グループより大きいグループを除外することで，A' は A より小さくなる（グループ総数 N が Partition 総数 R より本当に少なく，全グループのサイズがほぼ均一の場合，式 (3) の特別対処により，全グループが分割されうるが，その場合は特別に $A' = \infty$ として扱う）．

次に SD は，進捗率 $p1$ におけるグループ $g1$ の 1 回目の分割係数 $x_{g1,1}$ を一般式 (4) で決定（$ceil$ は天井関数）し，分割係数情報に組み込む．グループ分割後の各部分のサイズが，その時点の未分割グループの平均サイズ以下になるよう，分割の基準（分母）に A'_p（ただし上述のとおり U を加える）を用いる．ここでは $A'_p + U$ が選ばれたものとする．

$$x_{g,n} = ceil\Big(\frac{e_{g,p}}{min\big(A'_p + U, \frac{S_{Out}}{R}\big)}\Big) \tag{4}$$

$x_{g1,1}$ の決定時点では $d_{g1,p} + f_{g1,p} \leq e_{g1,p1}$ すなわち $d_{g1,p} + f_{g1,p} \leq x_{g1,1} \times (A'_{p1} + U)$ が Map フェーズ終了まで成立すると期待したが，$e_{g1,p1}$ の過小予測や，他グループの分割による A' の低下により，分割済みのグループ $g1$ および進捗率 $p2$ において，一般式 (5) が成立（ここでは $n = 1$）した場合，SD はグループ $g1$ に分割更新判断を下す．

$$d_{g,p} + f_{g,p} > x_{g,n} \times min\Big(A'_p + U, \frac{S_{Out}}{R}\Big) \tag{5}$$

すると SD は，進捗率 $p2$ におけるグループ $g1$ の 2 回目の分割係数 $x_{g1,2}$ を式 (4) で決定し，分割係数情報へ組み込む．

以降，Map フェーズ終了までグループ $g1$ の分割更新判断と分割係数決定を繰り返す．

なお提案システムでは，過少のカウント情報に基づく動

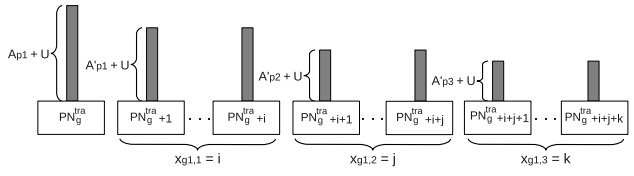


図 8 分割係数情報に基づくパーティショニング
Fig. 8 Partitioning based on division information.

作を回避するため，Map フェーズ進捗率 p が 0.1%を超過してから (α) および (β) の処理を行うようにしたが，別の基準（たとえば，処理済み入力データサイズ s_{in} がブロックサイズ B を超過）を用いてもよい．

4.5 Map タスクによる分割係数情報に基づくパーティショニング

提案方法では，(α) 処理で分割判断が下った大きなグループを，(β) 処理で求めた分割係数に従って等分割することで，Data Skew を緩和する．

従来の MapReduce では，たとえば式 (6) を用いて，グループ g の中間 Key（Key_g）に対する Partition（$Partition_g$）を一意に決定する．したがって，同じ中間 Key を共有するすべての中間レコードが 1 つのグループを構成する．

$$Partition_g^{tra} = hash(Key_g) \bmod R \tag{6}$$

提案方法では，Mapper がグループ g に関する分割係数情報を未取得の間は $Partition_g^{tra}$ をそのまま用いるが，取得後は式 (7) を用い，同じ中間 Key に対して複数の Partition を割り当てることで，中間 Key を変更せずにグループ g を複数の Partition へ分割する．

$$Partition_g^{pro} = \Big(Partition_g^{tra} + \sum_{n}^{n_{max}-1} x_{g,n} + select(1, x_{g,n_{max}})\Big) \bmod R \tag{7}$$

ここで n_{max} は $x_{g,n}$ の最新の n を表す．$select(1, x_{g,n_{max}})$ は，閉区間 $[1,\ x_{g,n_{max}}]$ 内の整数値を，$Partition_g^{pro}$ を計算するたびにラウンドロビンで返す関数である．式 (7) によるパーティショニングの様子を**図 8** に示す．Reduce-Side Joins において必要となる主データの複製数は $min(\sum_n^{n_{max}} x_{g,n}, R-1)$ である（主データの必要最大数は R であるが，$Partition_g^{tra}$ へ出力済みの分を減じて，複製すべき最大数は $R-1$ となる）．

4.6 プログラミングモデル

プログラマにとって使いやすいものとするため，提案システムでは Hadoop [5] における MapReduce プログラミングモデルとの整合性を重視している．本節では，従来のプログラミングモデルと提案システムとの差異をまとめる．なお今回実装した提案システムでは，実装の容易性を考慮し，Hadoop のソースコードを直接改変することは避けた．改変による実装を行えば，以下の独自クラスやメソッド名を従来のものと統合することは可能である．

Job を起動するドライバクラスは，従来の Configured クラスを継承する代わりに独自の DivConfigured クラスを継承する．このクラスには register メソッドが用意されており，提案方法による Data Skew 緩和を行う場合はこれをコールする必要がある．コールしなかった場合，本節に従って実装した MapReduce プログラムであっても，従来どおりの動作となる．また，Job 起動時のパラメータとして格納レート Q と閾値 U が指定可能となっており，省略した場合はデフォルト値（1%および 1 MB）が適用される．今日の計算機にとって，1 MB のデータの処理は，多くの場合それほど長時間を要するものではないと我々は考える．したがって，5 章では U はデフォルト値に固定して評価を行う．

従データを処理する Mapper は，従来の Mapper インタフェースを継承して map メソッドを実装する代わりに，独自の DivMapper クラスを継承して doMap メソッドを実装する．主データを処理する Mapper は，独自の DivMapperMaster クラスを継承し，doMap メソッドを実装する．DivMapperMaster クラスを継承した Mapper は，入力データの処理完了後も分割係数情報を取得し続け，グループ分割が発生するたびに該当する主データを複製出力する（なお，2 つ以上の入力データに対してそれぞれ異なる Mapper を使用するには，Hadoop の MultipleInputs 機構を利用すればよい）．また，パーティショニングをカスタマイズする場合は，従来の Partitioner インタフェースを継承して getPartition メソッドを実装する代わりに，独自の DivPartitioner クラスを継承して calcPartition メソッドを実装する（提案システムでは，従来の map メソッドや getPartition メソッドを，カウント情報の収集や ST とのやりとりのために利用している）．ただし，これらのメソッドの実装内容は従来とまったく同じものでよい．なお，Reducer については従来のままである．

このように，従来の MapReduce プログラムのわずかな修正で，提案方法を適用できる．

5. 評価と考察

5.1 準備

5.1.1 入力データ

本章では，3.2 節の処理を用いた評価を行う．入力データの様子を**図 9** に示す．座標 (0,0) から (20,20) の範囲を定義域とし，縦横 20 等分割した 400 個のグリッドを用いる．主データは，グリッドごとの領域数がちょうど 5 個になるように作成した．従データは，グループ間のサイズ差異が異なる 3 種類を用意した．図 9 (b) では，グリッドごとの点数が一定になるように各点の X および Y 座標を決定し，グループ間のサイズ差異をなくした．図 9 (c)，(d)

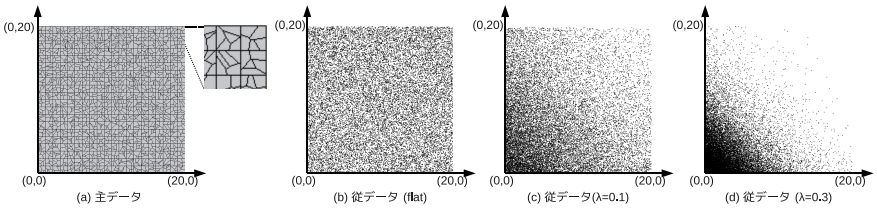


図 9 評価の入力データ
Fig. 9 Input data for estimation.

では，各点の X および Y 座標を指数分布 $f(x,\lambda)=\lambda e^{-\frac{x}{\lambda}}$ に則り決定し，グループ間のサイズ差異を設けた．いずれの従データも，レコード数は $1{,}024^3$，レコード長はすべて 100 Byte，総入力データサイズは 100 GByte である．

提案システムを実用に供することを考えた場合，入力データ内でのレコードの並び順について，何も前提を置くことはできない．そこで図 9 (b)，(c)，(d) に関し，それぞれ両極端な 2 種類のデータを用意した．一方は，同じグループに属することになる入力レコードどうしを一カ所に集約したもの（以下，“Sorted 版”）であり，大きいグループほど Map フェーズの早期に処理されるように配置した．もう一方は，同じグループに属することになる入力レコードどうしを一カ所に集約せず，入力データ全体に無作為に分散したもの（以下，“Spread 版”）であり，Map フェーズの全期間にわたって全グループのレコードが満遍なく出現する．

5.1.2 MapReduce プログラム

3.2 節の処理を行う 2 つの MapReduce プログラムを用意した．“従来版” は従来方法（文献 [6] の p.235）に従って実装したものである．Partition は式 (6) ではなく，400 個の各グリッドに，左下から始めて Y 軸方向優先で右上方向へ 0 から 399 までの通番 $Number_{grid}$ を振り，$Partition_g^{tra} = Number_{grid} \bmod R$ で決定するようにした．これは，各 Partition に割り当てられるグループ数を均一化（1 章の (I) は本論文の対象外）するためである．“提案版” は 4.6 節に従って実装したものであり，式 (7) における $Partition_g^{tra}$ は，やはり本節のものを用いる．

5.1.3 実行環境

1 台の JobTracker ノードと 100 台の TaskTracker ノードを用いてクラスタを構成し，その上で Hadoop v0.19.1 を動作させた．各 TaskTracker ノードは，HDFS の DataNode の役割も兼ねる．各 TaskTracker ノードで同時実行される Map タスク数は 2（つまり，$M_{pool}=200$），Reduce タスク数は 1 とした．Job で使用する Reduce タスク数（Partition 数）R は 100 とした．さらにバックアップタスクは実行しないように設定した．HDFS のブロックサイズ B はデフォルトの 64 MByte としたため，入力データは約 64 MByte の split に分割され，各 Map タスクに割り当てられる．提案版の実行時に限り，JobTracker ノードおよび全 TaskTracker ノードでそれぞれ SD と CD を実行させた．また，全ノードのハードウェア構成は統一されており，CPU は Intel 3 GHz Dual コア，メモリ容量は 4 GByte，ネットワーク接続は 1 Gbps Ethernet である．

5.2 評価結果と考察

提案方法の目的は，Reduce フェーズの Data Skew を緩和し，Job 完了時間を短縮することである．したがって評価結果では主に，Job 完了時間（Map フェーズと Reduce フェーズの所要時間の合計）と，各 Reduce タスクに割り当てられた中間データサイズ（GByte 単位）の標準偏差に注目するが，必要に応じて他の結果も示す．以降では，Job 開始から Map フェーズ進捗率が 100%に達するまでの時間を Map フェーズ所要時間，そこから Job 完了までの時間を Reduce フェーズ所要時間としている．したがって Reduce フェーズ所要時間には，Reducer の実行時間だけでなく，Map タスクから Reduce タスクへの中間データ送達やそのソート処理に要する時間の一部も含まれる．Job 完了時間の評価では，各 10 回ずつ試行して成績の良い（Job 完了時間が短い）上位 5 回の平均値を用いた．

ところで，MapReduce プログラムでは Mapper や Reducer の実装の自由度が高く，実行環境や入力データも千差万別である．したがって，本節で示す Job 完了時間の数値自体は絶対的なものではなく，あくまで比較対象間の関係性やそれぞれの傾向に注目することとする．

最初に，指数分布（$\lambda=0.1$ および 0.3）のデータを用いた，Map フェーズ進捗に応じた予測最終平均サイズ A，A'，および出現済みグループ数 N の結果を**図 10** に示す．格納レート Q は 1%とした．図 10 (a)，(b) は Spread 版の結果であり，Map フェーズを通じて A，N はほぼ一定である．特に N は，序盤から正しい値（$=400$）を検出できている．この場合，提案方法は設計どおりの効果を発揮する．一方，図 10 (c)，(d) は Sorted 版の結果である．最終的には Spread 版と同じ値に近づくが，Map フェーズを通じて値が大きく変化する．これは，序盤に特定グループだけが集中して出現し，N が正しい値に達するのが遅れるためである．N と反比例の関係にある A，A' は，序盤では最終的な値よりも大きくなる．この影響で，分割開始判断が下るタイミングが遅れたり，分割係数が過小評価されたり

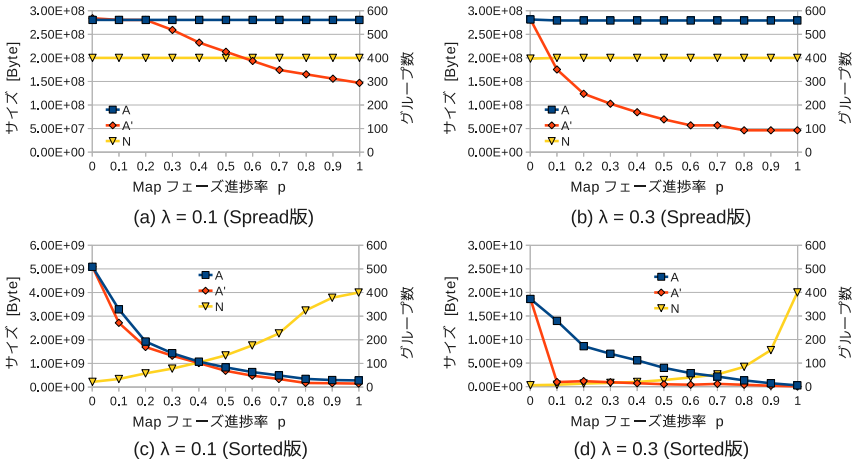


図 10 予測最終平均サイズとグループ出現数

Fig. 10 Estimated average size and detected group number.

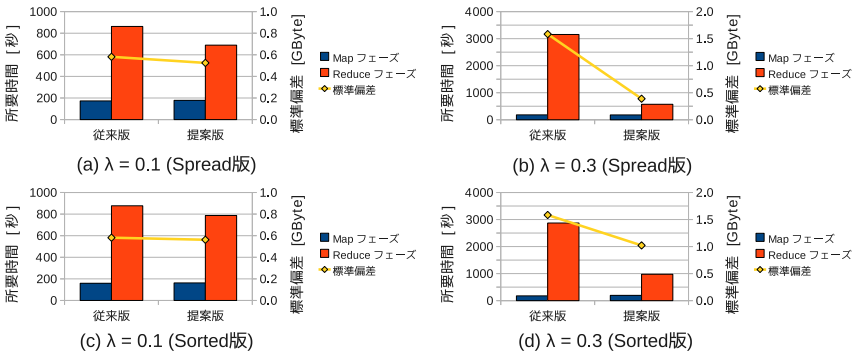


図 11 従来方法と提案方法の比較

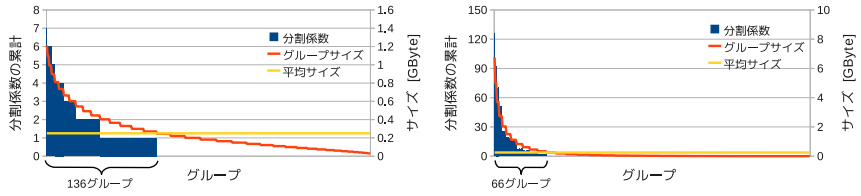
Fig. 11 comparison between normal maprecude and proposal.

するため，提案方法の効果が低下することが予想される．

次に，指数分布（$\lambda = 0.1$ および 0.3）のデータを用い，5.1.2 項の 2 つのプログラムを実行した結果を**図 11** に示す．格納レート Q は 1%とした．図 11 (a)，(b) は Spread 版の結果である．従来版と比較して提案版の Reduce フェーズ所要時間が短縮されており，標準偏差も低下していることから，提案方法による Data Skew 緩和効果が発揮されていることが分かる．なお，$\lambda = 0.3$ のほうがグループ間のサイズ差異が大きいため，提案方法の効果も高い．一方，図 11 (c)，(d) は Sorted 版の結果である．従来版と比較すると提案方法の効果が発揮されているが，Spread 版の結果と比較すると Reduce フェーズ所要時間，標準偏差ともに効果が低下していることが分かる．これは前述の原因によるものである．Sorted 版のようなデータが与えられた場合に，提案方法の効果の低下を防ぐ手段については，今後の課題である．

次に，指数分布（$\lambda = 0.1$ および 0.3）の Spread 版におけるグループ分割の様子を**図 12**，**図 13** に示す．格納レート Q は 1% とした．図 12 では，横軸に 400 グループをサイズ順（降順）に並べ，各グループの分割係数の累計（$\sum_n^{n_{max}} x_{g,n}$）を示す．$\lambda = 0.1$ では 136 グループ，$\lambda = 0.3$ では 66 グループにおいて分割が行われ，グループ間のサイズ差異が大きいほど，より少数のグループが集中的に分割されている様子が分かる．図 12 から，大きいグループほど多く分割するという，提案方法の効果を確認できる．図 13 では，図 9 の入力データで左下座標が (0,0)〜(4,0) である 5 つのグリッドに対応するグループについて，分割判断（開始および更新）が下ったタイミングを示す．最初の分割開始判断が下った後も，分割更新判断が繰り返し下っている様子が分かる．これは主に，図 10 のように Map フェーズ中に予測最終平均サイズ A' が低下する影響によるものである．図 13 から，状況に応じて分割係数を更新するという，提案方法の効果を確認できる．

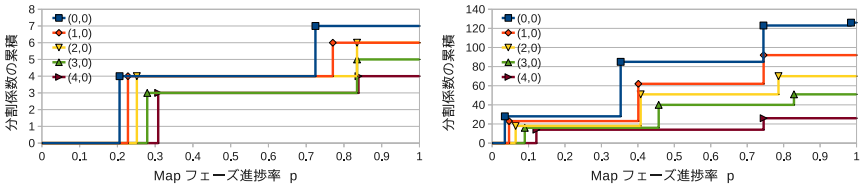
次に，式 (2) で算出される回転比 Z の結果を**図 14** に示す．ここでは格納レート Q を 0.05% から 50%まで変動させた．入力データは指数分布（$\lambda = 0.1$）の Spread 版を用いた．図 14 (a) はすべての結果を含んでおり，格納レートが小さいほど Z が大きくなる傾向が分かる．つまり，図 2 の報告ループが 1 回転する間の格納ループ回転数に応じて Z が決定される．図 14 (b) は (a) の一部を拡大したものである．$Q = 1\%$ の場合，Z が 1 に近くなっている．これは今回の評価環境において，2 つのループの回転周期がほぼ



(a) λ = 0.1　(b) λ = 0.3

図 12　全グループの分割結果

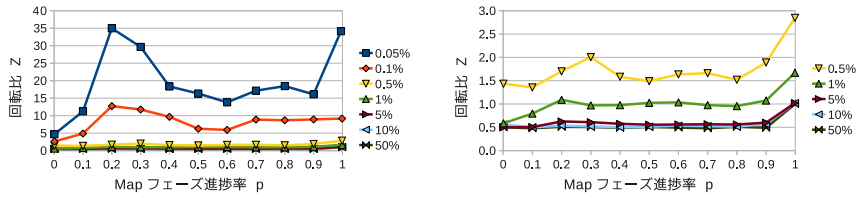
Fig. 12　Division result of all groups.



(a) λ = 0.1　(b) λ = 0.3

図 13　5 グループの分割の詳細

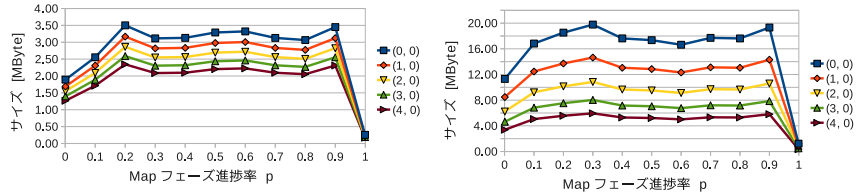
Fig. 13　Detail of division result on 5 groups.



(a) 全ての結果　(b) 一部に注目

図 14　Z の算出結果

Fig. 14　Result of Z calculation.



(a) λ = 0.1　(b) λ = 0.3

図 15　5 グループの f_g の算出結果

Fig. 15　Result of f_g calculation on 5 groups.

一致していることを表す．Q を大きくするほど格納ループの回転周期は長くなり，報告ループの回転周期よりも長くなると，Z は次第に $\frac{T}{M_{pool}}$（各 TaskTracker ノードで同時実行される Map タスクの最大数の逆数）に近づく．以上から，Z は 2 つのループの回転周期のバランスによって決定されることが分かる．既述のとおり，クラスタ規模が変化すると報告ループの回転周期が変化する一方，格納ループの回転周期はクラスタ規模には影響されない．したがって，クラスタ規模が大きくなるほど Z も大きくなり，式 (1) の遅延カウント量の算出に反映される．なお，Map フェーズ終盤で値が上昇する傾向にある．正確な原因は不明だが，中間データの Reduce タスクへの送達量が次第に増加し，ネットワークが混雑するためと推測される．

次に，図 9 の入力データで左下座標が (0,0)〜(4,0) である 5 つのグリッドに対応するグループについて，式 (1) で算出される遅延カウント量 f_g の結果を**図 15** に示す．入力データは指数分布（$\lambda = 0.1$ および 0.3）の Spread 版を用い，格納レート Q は 1% とした．今回の評価環境では，遅延カウント量は最大で 18 MB 程度となる．クラスタ規模が増大するほど実行中 Map タスク数 M_{cur} および回転比 Z が増加し，遅延カウント量 f_g はそれらの積に比例する．したがって，クラスタ規模が増大（4,000 台 × 8 コア [8] という事例も報告されている）するほど，遅延カウント量は無視できないものとなる．提案方法は遅延カウント量を考慮しているため，クラスタ規模によらず適用可能である．

次に，格納レート Q を 0.05%から 50%まで変動させた

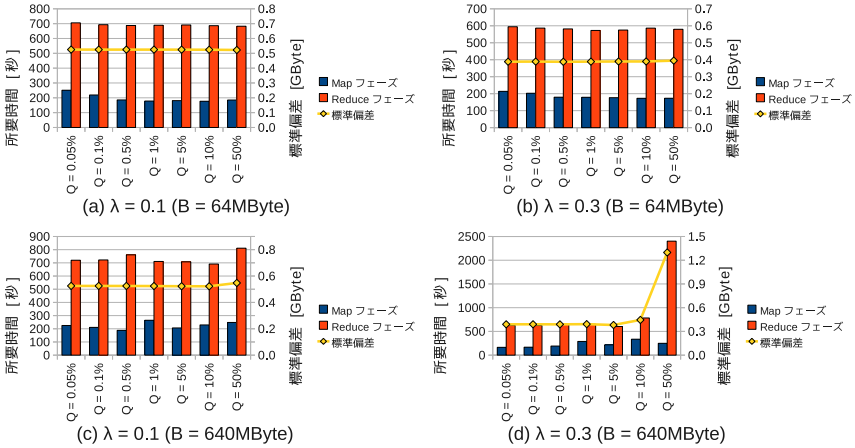


図 16 格納レート Q の変動による影響

Fig. 16 Influence of storing-rate Q variation.

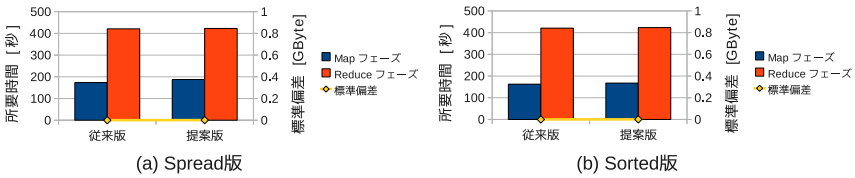


図 17 Data Skew が存在しない場合の結果の比較

Fig. 17 Comparison between results for non-skewed data.

場合の，Job 完了時間の結果を図 16 に示す．入力データは指数分布（$\lambda = 0.1$ および 0.3）の Spread 版を用いた．図 16 (a)，(b) から，Q を小さくすると Map フェーズ所要時間が若干伸びることが分かる．これは Mapper から ST への格納回数が増し，その分のオーバヘッドが増加するためである．一方で，図 16 (a)，(b) では Q の変動による Reduce フェーズ所要時間や標準偏差への大きな影響は見られない．これは 5.1.3 項の条件下で実行したためであり，入力データ 100 GByte のうち，一時に Map タスクの処理対象となるのはたかだか 12.5 GByte（$= B \times M_{pool}$）である．Map フェーズの途中から起動される残りの Map タスクは，起動時点における分割係数情報（図 3 の 1 行目）を最初から利用できる．もしも 5.1.3 項の 10 倍規模のクラスタを用いたならば，全入力データが一時に Map タスクの処理対象となり，全 Map タスクの実行がいっせいに開始される．つまり全 Map タスクが，分割係数情報をまったく利用できない状態から実行され始めることになる．今回はそのようなクラスタを用意できないが，代わりにブロックサイズ B を 10 倍の 640 MByte とし，全入力データが一時に処理対象となるようにして同様の評価を行ったのが図 16 (c)，(d) である（ブロックサイズを増やすと Map タスクあたりの中間データも増え，それが内部バッファのサイズを超過すると，Hadoop による中間データのソート処理がメモリ上ではなくディスク上で行われるようになり，処理効率が落ちる．図 16 (c)，(d) の所要時間はその影響を含んでいるため，ここでは標準偏差に注目する）．図 16 (c)，(d) から，Q を大きくすると標準偏差が伸び，提案方法の効果が著しく低下することが分かる．以上から，クラスタ規模によって提案方法の効果を著しく低下させないために，Q を大きくしすぎることは避けるべきである．また，Map フェーズ所要時間の伸びを抑えるために，Q を小さくしすぎないことが望ましい．最適な Q を一意に決定することは困難であるが，0. 数%から数%の間であれば，提案方法の効果をおおむね良好に引き出せる．

最後に，図 9 (b) の flat なデータを用い，5.1.2 項の 2 つのプログラムを実行した結果を図 17 に示す．格納レート Q は 1%とした．グループ間のサイズ差異がなく，そもそもグループ分割が不要な入力データであるため，提案方法の効果は特に発揮されない．一方で Mapper から ST へのカウント情報の格納によるオーバヘッドの分，提案版の Map フェーズ所要時間が若干増加するが，図 17 から分かるとおりその影響は小さい．初期の提案システムでは図 2 の 2 つのループを同期させていたため，Map フェーズのオーバヘッドが大きかったが，非同期化したことでこの程度に抑えられた．したがって，Data Skew の程度の多寡によらず提案方法を適用しても問題はない．

6. 関連研究

Data Skew の研究は，分散 DB システムの Join 操作を対象として活発に行われてきた．

Wolf ら [9], [10]，Kitsuregawa ら [11] では，Data Skew の検出およびデータのノードへの割当て方法を決定するた

めに，従来方法に対して新たなフェーズを導入している．MapReduce はそもそも大量のデータを対象としており，大量の入力データを Map フェーズで処理している間に，大量の中間データを Reduce タスクへ送達する．しかしこれらの方法を MapReduce へ適用した場合，Map フェーズと Reduce フェーズの間に新たなフェーズを追加し，その前後で同期をとる必要が生じる．この場合，中間データの Reduce タスクへの送達を Map フェーズとオーバラップさせることができなくなり，Job の処理効率が著しく低下する．さらに，新たなフェーズを導入することで，当該フェーズの所要時間を新たに要することにもなる．これらの方法を適用した効果が，そのような処理効率低下を補えるケースでは問題はないが，そもそも Data Skew が存在しない入力データが与えられた場合は，処理効率低下をいっさい補えない．我々の提案方法では，Data Skew の検出やグループ分割の決定を Map フェーズ中に行い，新たなフェーズの導入を回避している．それでも，Data Skew が存在しない入力データが与えられた場合，従来方法と比較して Map フェーズの性能が若干低下するが，図 17 のとおりその程度は非常に小さく，問題はない．Reduce フェーズの Data Skew の発生有無を，Job の実行者が事前に判断することは困難であるため，発生有無を気にせずに提案方法を適用できることは重要である．

Hua ら [12] では，全グループ（文献中の “bucket”）のサイズを把握した後，ノードごとの割当てデータサイズが均等になるよう，グループのノードへの割当て（つまり，グループの組合せ）を決定し，一部のノードにデータが多く集まることを回避する．しかしこの方法では，ノードあたりの平均サイズを超える巨大なグループが存在した場合，当該グループを割り当てられたノードには，必然的に他より多くのデータが集まることになる．またこの方法では，グループの組合せを調整するため，ノード数に対して十分に多いグループ数が存在することを前提としている．しかし MapReduce では，入力データや Mapper による中間 Key の選び方，クラスタの規模によっては，Partition 数よりグループ数が少なくなりうる．一方，我々の提案方法は大きなグループを分割するアプローチであり，どんなに大きいグループでも，式 (3) により，最悪でもノードあたりの平均サイズ（$\frac{S_{Out}}{R}$）に分割される．また，Partition 数よりグループ数が少ない状況でも，分割対象となるグループが存在すればその効果を発揮する．そしてそのような状況では，最終的なグループ平均サイズは必ず $\frac{S_{Out}}{R}$ を超過するため，式 (3) の特別対処により，必ずいずれかのグループが分割対象となる．

Xu ら [13] では，入力データ内におけるグループ間のサイズ差異があらかじめ判明していることを前提としている．また DeWitt ら [14] では，Data Skew を検出するために，入力データに対するサンプリング抽出と事前分析を用いる．しかし MapReduce には，中間データにおけるグループ間のサイズ差異を Job 開始前にあらかじめ把握する方法は存在しない．また，中間 Key として何を用いるかは Mapper の実装に依存するため，Reduce フェーズの Data Skew の検出に対し，入力データのサンプリング抽出と事前解析という手段がつねに簡単かつ有効なわけではない．また MapReduce では，入力データのサンプリング抽出をどのように行うのかも，課題となる．そのため我々の提案方法では，Map フェーズ中に実際に出力された中間データに基づいて，グループ間のサイズ差異を動的に検出するようにした．

7. おわりに

本論文では，一部のグループのサイズが極端に大きいことに起因する，MapReduce における Reduce フェーズの Data Skew に関して，その影響を緩和するための方法とその適用対象を提案し，100 台規模のクラスタを用いた定量評価により，提案方法の効果を示した．提案方法では，グループの分割判断を行う責務が単一ノードにあるため，MapReduce クラスタの規模が増大するほど当該ノードへの負荷も増すが，負荷の程度とクラスタ規模に応じてグループ分割を予測的に行うことを特徴とする．ところで提案方法は，各グループに属することになる入力レコードが，入力データ全体にわたって満遍なく分散しているほど，その効果を発揮する．一方で，そうでない入力データを用いると提案方法の効果が低下するという問題があり，これは今後の検討課題である．

謝辞 評価環境の構築にご協力いただいた NTT ドコモ先進技術研究所の趙晩熙氏，甲本健氏のご両名に深く感謝いたします．

参考文献

[1] Dean, J. and Ghemawat, S.: MapRedue: Simplified Data Processing on Large Clusters, *Proc. OSDI '04*, Vol.6 (2004).
[2] Kavulya, S., Tan, J., Gandhi, R. and Narasimhan, P.: An Analysis of Traces from a Production MapReduce Cluster, *Proc. CCGRID '10*, pp.94–103 (2010).
[3] Lin, J.: The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce, *Proc. LSDS-IR '09*, pp.57–62 (2009).
[4] Kwon, Y., Balazinska, M., Howe, B. and Rolia, J.: Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions, *Proc. SoCC '10* (2010).
[5] Apache.org: Hadoop, available from ⟨http://hadoop.apache.org/⟩ （参照 2011-07-29）.
[6] White, T.: *Hadoop The Definitive Guide*, O'REILLY (2009).
[7] Apache.org: HDFS, available from ⟨http://hadoop.apache.org/hdfs/⟩ （参照 2011-07-29）.
[8] Yahoo: Scaling Hadoop to 4000 nodes at Yahoo!, available from ⟨http://developer.yahoo.com/blogs/hadoop/posts/2008/09/scaling_hadoop_to_4000_nodes_a/⟩

（参照 2011-07-29）.
[9] Wolf, J.L., Dias, D.M., Yu, P.S. and Turek, J.: An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew, *Proc. ICDE '91*, pp.200–209 (1991).
[10] Wolf, J.L., Dias, D.M. and Yu, P.S.: An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew, *Proc. DPDS '90*, pp.103–115 (1990).
[11] Kitsuregawa, M. and Ogawa, Y.: Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC), *Proc. VLDB '90*, pp.210–221 (1990).
[12] Hua, K.A. and Lee, C.: Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning, *Proc. VLDB '91*, pp.525–535 (1992).
[13] Xu, Y., Kostamaa, P., Zhou, X. and Chen, L.: Handling Data Skew in Parallel Joins in Shared-Nothing Systems, *Proc. ACM SIGMOD '08*, pp.1043–1052 (2008).
[14] DeWitt, D.J., Naughton, J.F., Schneider, D.A. and Seshadri, S.: Practical Skew Handling in Parallel Joins, *Proc. VLDB '92*, pp.27–40 (1992).

中山 誠 （正会員）

平成 11 年早稲田大学理工学部電気工学科卒業．平成 13 年同大学大学院修士課程修了．同年（株）NTT ドコモ入社．主に携帯電話端末搭載アプリケーションの開発に従事した後，平成 19 年より Web，ネットワークシステム，大規模分散処理の研究に従事．現在，NTT ドコモ先進技術研究所研究主任．

山崎 憲一 （正会員）

昭和 59 年東北大学工学部卒業．昭和 61 年同大学大学院修士課程修了．同年日本電信電話（株）入社．平成 12 年より（株）NTT ドコモ．平成 22 年より芝浦工業大学．プログラミング言語，OS，記号処理計算機，形態素解析，ユビキタスコンピューティングの研究に従事．現在，芝浦工業大学デザイン工学科教授．博士（工学）．ACM，電子情報通信学会各会員．

田中 聡 （正会員）

昭和 60 年慶應義塾大学工学部管理工学科卒業．昭和 62 年同大学大学院修士課程修了．同年日本電信電話（株）入社．以来専用マシン，ユビキタスコンピューティング，大規模分散処理研究に従事．現在，NTT ドコモ先進技術研究所主幹研究員．電子情報通信学会会員．