Vol.2011-HPC-130 No.62
2011/7/29

系列パターン抽出のMapReduce 実装におけるタスク分割方式の検討

井 上 佑 希[†1] 置 田 真 生[†1] 萩 原 兼 一[†1]

並列分散を容易にするプログラミングモデルの一つとして，Google 社が提唱する MapReduce がある．高性能な MapReduce プログラムを記述するためにはアルゴリズムに応じて適切なタスク分割を選択する必要がある．本稿では，系列パターン抽出アルゴリズム PrefixSpan を対象に，MapReduce 実装におけるタスク分割方式を検討する．複数のタスク分割方針を比較する実験の結果，データ参照と計算時間の削減に重点をおいた実装が最も高速であることを確認した．

A study of MapReduce task design guidelines for implementing sequential pattern mining

Yuki Inoue , Masao Okita and Kenichi Hagihara

MapReduce, which is originally developed by Google, is a progamming model for supporting development of parallel and distributed programs. A suitable MapReduce task design for an algorithm enhances the performance of a program. In this paper, we investigate suitable designs for a sequential pattern mining algorithm known as PrefixSpan. Results from comparison of three different designs show that a design aiming to reduce data access and computation is the most efficient in our proposed designs.

1. はじめに

近年，記憶装置の大容量化およびインターネットの普及に伴い，Web ページ，アクセスログ，画像など多種多様なデータが生成されている．その容量は加速度的に増加しており，データを利用するための解析時間の増大が問題となっている．現実的な時間内で解析するためには，並列分散処理が必須である．

一般に，並列分散処理の開発における開発者の負担は大きい．その理由は，プログラム本来の処理に加えて，並列分散特有の処理を実装する必要があるためである．並列分散特有の処理の例として，タスク分散，データ通信，同期，耐故障性などがある．

大規模な並列分散処理の実装を容易にする技術として，Google 社が提唱する MapReduce プログラミングモデル[1] がある．MapReduce を用いる利点は，並列分散特有の機能を開発者自身が実装する必要がなく，解決したい問題のアルゴリズムのみを実装すれば良い点にある．開発者は，解決したい問題を分解し，抽出処理（Map タスク）および集約処理（Reduce タスク）の組合せで表現する（以降，この作業をタスク分割と記述する）．その他の処理は MapReduce のフレームワークが自動的に実行するため，開発者の負担が軽減される．

ただし，高性能な MapReduce プログラムを記述するためには，アルゴリズムに応じて適切なタスク分割を選択する必要がある．ログの集計のような単純なアルゴリズムであれば，Map タスクと Reduce タスクは自明である．一方で，複雑なアルゴリズムにおいては複数のタスク分割方式があり得る．選択するタスク分割方式によっては，十分な性能を発揮できない[2]．この主な要因は，Map タスクと Reduce タスク間のデータ（以降，中間データ）の入出力および通信である．中間データは MapReduce フレームワークの特性上必ず発生する．

複雑なアルゴリズムの 1 つに，PrefixSpan[3] がある．これはパターン抽出アルゴリズムの一種であり，文字列やタンパク質構造の解析に利用される．PrefixSpan は計算量が大きく，多大な計算時間を必要とする．例えば，PrefixSpan を用いてソースコードを解析するツール Fung[4] では，ソースコード集合 50MB の解析に，計算機 1 台を用いて 24 時間を要する．さらなる大規模化のために並列分散処理による高速化が求められており，大規模データ処理に適した MapReduce による実装が期待されている．しかし，高性能な PrefixSpan を実現するための MapReduce のタスク分割方式は明らかでない．

そこで本研究では，広く普及している MapReduce フレームである Hadoop[5] を用いて，PrefixSpan プログラムを実装する．本研究の目的は，PrefixSpan に適した MapReduce のタスク分割方式を究明することである．そのため，3 つの異なる方針のタスク分割方式に基づいて PrefixSpan を実装する．各実装を比較し，もっとも高性能な設計方針を提案する．

以降，2 章では関連研究について述べる．次に 3 章では前提となる PrefixSpan および MapReduce について説明し，4 章では 3 つのタスク分割方式について説明する．その後，

†1 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻
Department of Computer Science, Graduate Schoolof Information Science and Technology, Osaka University

© 2011 Information Processing Society of Japan

5 章では比較実験の結果について述べ，最後に 6 章でまとめと今後の課題について述べる．

2. 関連研究

Richard Mccreadie ら[2) は，MapReduce を用いて分散索引生成のアルゴリズムを実装している．しかし，単純な実装では従来のシステムより低い性能しか達成できない．そこでタスク分割方針（中間出力の Key-value ペア）を改善することで中間入出力を削減し，性能を向上している．

また，高性能な MapReduce プログラムを実現するための異なるアプローチとして，高速なハードウェアを利用する方法がある．小川ら[6) は，読み書き性能の高速な SSD(Solid State Drive) を利用したクラスタにおける MapReduce の実現を目指している．一方で，Becerra ら[7) は，ヘテロジニアスマルチコアプロセッサの Cell BE(Cell Broadband Engine) クラスタにおける MapReduce を実現している．これらの手法を本研究で提案する設計方針と組み合せて利用することで，より高性能な MapReduce プログラムの実現を期待できる．

本研究で利用する Hadoop は，大規模クラスタ環境での動作を前提としている．その他，様々な環境に対応した MapReduce フレームワークが存在する．マルチコア環境を対象とした Phoenix[8)，GPU 環境を対象とした実装[9) や Mars[10)，および小規模クラスタ環境を対象とした実装[11) がある．動作環境ごとに性能ボトルネックが異なるため，適切なタスク分割方針も異なる．本研究の対象は，Hadoop を用いたクラスタ上で動作する MapReduce プログラムのみに限定する．

3. 背景

3.1 PrefixSpan アルゴリズム

PrefixSpan は，系列の集合内に頻出するパターンの列挙問題を解くアルゴリズムである．系列の集合 S を入力とし，出現回数が閾値 f を上回る系列パターンの集合 P を出力する．ここで，系列 $s \in S$ および系列パターン $p \in P$ は，ある集合 E の要素 e の順列として表現される．s および p は要素の重複を許す一方，1 つの s 中には p の重複は許さない．なお，今回実装するアルゴリズムでは p の長さ $|p|$ にも閾値 l を設定し，$|p| \geq l$ となる p のみを出力する．

PrefixSpan では，長さの短い p の列挙から始め，頻出する p のみを拡張してより長い p を列挙することで，効率的に P を抽出する．アルゴリズムの概要を以下に示す．初期値として $P = \emptyset$，$q = \epsilon$，$T = S$ を与える．ここで，$C(s)$ は s を構成する要素の集合を表す．

(1) **列挙**：$\forall s \in T(e \in C(s))$ を満たす e を列挙する
(2) **限定**：列挙した e を集計し，個数が f 以上の e を要素とする集合を E' とする
(3) $|q| + 1 \geq l$ の場合，$\forall e \in E'$ について，q の末尾に e を連結したパターン q^+ を P に追加する
(4) $\forall e \in E' \forall s \in T(e \in C(s))$ を満たす (s, e) の全ての組について，
  (a) **射影**：s のうち e に後続する系列 s'（以降，後続列）を取り出し，s' を要素とする集合を T' とする
  (b) $T = T'$，$q = q^+$ として 1 から処理を繰り返す

このアルゴリズムは繰り返しを含んでおり，終了条件は $T = \emptyset$ である．以降，このアルゴリズムを A と記述する．

3.2 MapReduce

本章では，Hadoop を基に MapReduce の概要について説明する．Hadoop は，分散共有ファイルシステム HDFS 上で動作するマスタ・ワーカ型のフレームワークであり，現在 Apache Software Foundation によりオープンソースで開発が進められている．

1 つの MapReduce プログラムの実行単位（ジョブ）は，複数の Map タスクおよび複数の Reduce タスクから構成される．Map タスクの並列実行を可能にするため，各 Map タスクは完全独立である必要がある．Reduce タスクも同様である．一方，Map タスクの出力結果が Reduce タスクの入力となるため，これらの間には依存関係が存在する．フレームワークのマスタがジョブの実行を管理し，ワーカが実際にこれらのタスクを実行する．

各タスクの入出力に利用するデータ構造は，Key-Value ペア（以降，KVP）に統一されている．KVP を用いることで，Key を基にした効率的なタスク配置およびデータ転送をフレームワークが自動的に実行できる．KVP では，全てのデータを文字列の組 $(key, value)$ として表現する．したがって，リストやオブジェクトを扱うプログラムでは，データをシリアライズ・デシリアライズする必要がある．なお，タスクの内部で利用するデータ構造は KVP に限らず自由である．

KVP の格納場所は，HDFS とワーカのローカルディスクの 2 箇所である．Map タスクの入力および Reduce タスクの出力，すなわちジョブの入出力は HDFS 上に格納する．一方，Map タスクの出力および Reduce タスクの入力，すなわち中間データはローカルディスク上に格納する．

MapReduce ジョブの実行は次の 3 つのフェーズからなる．これを図 1 に示す．各フェーズ内の処理は，複数のワーカ上で並列実行される．

© 2011 Information Processing Society of Japan

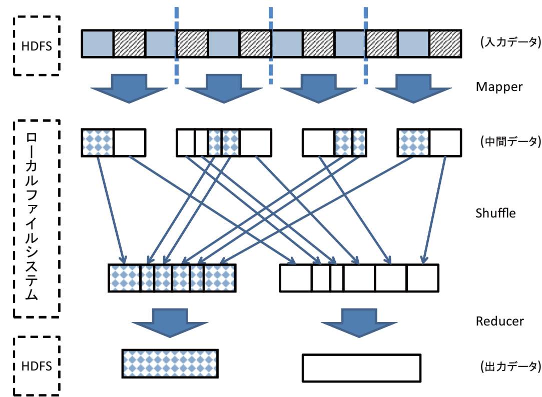

図 1 MapReduce のジョブ実行 (各ブロックが KVP を，ブロックのパターンが Key を示している)

- *Mapper*：マスタは入力である KVP を予め指定されたサイズに分割し，ワーカへ Map タスクを割り当てる．ワーカはユーザ定義の Map タスクを実行し，中間データの KVP を出力する．
- *Shuffle*：ワーカ間で通信を行い，中間データの KVP を *key* ごとに分類する．同じ *key* ごとに *value* のリストを作成し，ソートする．
- *Reducer*：ワーカは *key* と対応する *value* のリストを受け取り，ユーザ定義の Reduce タスクを実行する．データの転送量を削減するため，ワーカは必要なデータに近いノードに Reduce タスクを割り当てる．

なお，Mapper と Shuffle の間に Combiner を実行する場合もある．Combiner では，1 つのワーカ上に存在する中間データの KVP に対して，予め Reduce タスクを実行する．Shuffle における転送量を削減するために有用であるが，Reduce タスクの演算が結合的かつ可換でなければ適用できない．さらに，ジョブの進捗状況によって実行の可否が決まるため，実行の保障はない．

4. MapReduce を用いた PrefixSpan の実装

3.1 章のアルゴリズム A を変換し，Map タスクと Reduce タスクに分割する．実装するタスク分割方針は，列挙におけるタスク分割の基準および限定と射影の順序によって次の 3 つに分類できる．

- *s-EB*：s を基準に列挙を分割し，射影の後に限定を実行する
- *p-BE*：p を基準に列挙を分割し，限定の後に射影を実行する
- *s-BE*：s を基準に列挙を分割し，限定の後に射影を実行する

4.1 *s*-EB 実装の設計

この実装では，入力データの参照および計算量の削減に主眼を置いた．列挙および射影はそれぞれ S のデータ全体を参照する必要がある．A では，列挙と射影が独立しておりデータ全体への参照が 2 回発生する．そこで列挙と射影を同時に処理することで，重複を省く．

A における 1 回の繰り返しを 1 つの MapReduce ジョブとして実装する．入力データを KVP_i，中間データを KVP_m，出力データを KVP_o とすると，それぞれ次のように表現できる．

$$KVP_i = (q, s), \text{ただし } s \in T$$
$$KVP_m = (q^+, s')$$
$$KVP_o = (q^+, s')$$

Map タスクは列挙と射影を処理する．与えられた $KVP_i = (q, s)$ から $q + e(\forall e \in C(s)) = q^+$ を生成し，全て出力する．このように与えられた s に依存して q^+ を生成することを，s を基準とした列挙とする．これは，KVP_i から e の出現回数によらず全ての射影可能な s' を列挙することに等しい．

Reduce タスクは限定を処理する．Reduce タスクへの入力は，同一の q^+ に対応する s' の集合 V である．$|V| \geq f$ であれば，次の繰り返しの KVP_i として $\forall s' \in V(q^+, s')$ を出力する．さらに，$|q^+| \geq l$ であれば，P の要素として q^+ を出力する．

この実装の問題点は，中間データ量と Map タスクサイズの不均等である．まず，限定の前に射影を処理するため，A と比較して KVP_m の個数が増大する．したがって KVP_m の出力時間および Shuffle における転送時間がボトルネックとなる可能性が高い．次に，Map タスクの処理時間は $|s|$ に依存して増大する．$|s|$ に極端な偏りがある場合，Mapper における負荷分散に失敗し実行性能が低下する可能性がある．

4.2 *p*-BE 実装の設計

この実装では，中間データ量の削減に主眼を置いた．*s*-EB 実装において中間データ量が増大する原因は，射影の後に限定を処理するためである．したがって，A と同様に射影の前に限定を処理する．さらに，限定のための集計を効率よく処理するため，列挙において系列パターンごとにタスクを分割する．

A における 1 回の繰り返しを，2 つの MapReduce ジョブで実装する．この実装では，予

© 2011 Information Processing Society of Japan

め S を走査して E を取り出す必要がある．これはプログラムの初期時に一度だけ処理する．

まず，1 つ目のジョブは列挙と限定を処理する．このジョブの KVP は以下のようになる．ここで n，k は整数，U_k は T を d 個に分割した部分集合のうち k 番目の集合とする．

$KVP_i^1 = (q + e, U_k)$, ただし $e \in E$

$KVP_m^1 = (q^+, n)$

$KVP_o^1 = (q^+, \epsilon)$

Map タスクは列挙を処理する．$\forall s \in U_k$ に対して，e を含む s の個数 n を数える．q の末尾に e を連結して q^+ とし，(q^+, n) を出力する．ある q^+ を key に持つ KVM_m の個数は高々d 個であるため，d を小さくすることで中間データ量を抑制できる．

Reduce タスクは限定を処理する．同一の q^+ に対応する n を合算し，値が f 以上である場合のみ (q^+, ϵ) を出力する．

1 つ目のジョブの出力結果は，サイドデータを利用して 2 つ目のジョブに与える．サイドデータの詳細は 4.4 で後述する．

次に，2 つ目のジョブは射影を処理する．KVP は以下のようになる．

$KVP_i^2 = (q + e, U_k)$, ただし $e \in E$

$KVP_m^2 = (q^+, s')$

$KVP_o^2 = (q^+ + e, T')$, ただし $\forall e \in E$

Map タスクはサイドデータを参照し，q^+ がサイドデータに含まれていれば U から後続列を取り出す．KVP_m^2 の形式は s-EB 実装の KVP_m と同様であるが，限定処理によりその個数は減少している．

Reduce タスクは次の繰り返しのための KVP を作成する．さらに，$|q^+| \geq l$ であれば，P の要素として q^+ を出力する．

この設計の問題点は，HDFS への入出力量とワーカの主記憶容量である．q を基に派生しうる全ての系列パターン（$\forall e \in E(q + e)$）をジョブの入出力とするため，HDFS への入出力量が増大する．また，各 Map タスクが U_k を主記憶上に保持するため，d の値はワーカの主記憶容量で制限される．なお，A の 1 回の繰り返しを 2 つ MapReduce のジョブで実装するため，ジョブの開始および終了に伴うオーバヘッドも 2 回ずつ発生する．

4.3 s-BE 実装の設計

この実装では，MapReduce ジョブの繰り返しにおける KVP 入出力の削減に主眼を置いた．p-BE 実装では，先に挙げたように HDFS への入出力量が増大する．この原因は，q を基に派生しうる全ての系列パターンをジョブの入出力としているためである．したがって，この実装では s-EB 実装と同様に，s を基準とする列挙を Map タスク内で処理する．

A における 1 回の繰り返しを，2 つの MapReduce ジョブで実装する．

まず，1 つ目のジョブは列挙と限定を処理する．このジョブの KVP は以下のようになる．ここで n は整数とする．

$KVP_i^1 = (q, s)$, ただし $s \in T$

$KVP_m^1 = (q^+, n)$

$KVP_o^1 = (q^+, \epsilon)$

Map タスクは列挙を処理する．与えられた s において $q + e(e \in s) = q^+$ をすべて出力する．このとき Key は q^+,Value は 1 である．また In-Mapper(4.4) の手法を利用し，ノード内で q^+ が同一のものがあれば Value を合算し,n とする．

Reduce タスクでは限定を処理する．同一の q^+ に対応する n を合算し，値が f 以上である場合のみ (q^+, ϵ) を出力する．

1 つ目のジョブの出力結果は，p-BE 実装と同様，サイドデータを利用して 2 つ目のジョブに与える．

2 つ目のジョブは射影を処理する．KVP は以下のようになる．

$KVP_i^2 = (q, s)$, ただし $s \in T$

$KVP_m^2 = (q+, s')$

$KVP_o^2 = (q+, s')$

Map タスクは再度 s を基準とする列挙を処理する．s-EB 実装と同様に q^+ を列挙し，サイドデータに含まれている q^+ について後続列を取り出す．KVP_m^2 の個数は，p-BE 実装と同じく，s-EB 実装と比較して少ない．

Reduce タスクは，入力をそのまま出力すると共に，$|q^+| \geq l$ であれば，P の要素として q^+ を出力する．

この設計の問題点は，s-EB 実装と同様に，Map タスクサイズの不均等である．さらに p-BE 実装と同様，A の 1 回の繰り返しを 2 つの MapReduce のジョブで実装するため，ジョブの開始および終了に伴うオーバヘッドも 2 回ずつ発生する．また，In-Mapper の手法を利用するため，メモリ制約を受ける (4.4).

4.4 実装の補足

本研究での実装において，Hadoop の既存であるサイドデータと，In-Mapper Combiner の手法を利用した．

© 2011 Information Processing Society of Japan

表 1 実験環境

CPU	Xeon 2.8 GHz × 2
主記憶	2GB
ハードディスク	292GB
接続方式	Ultra320 SCSI
ネットワーク	Gigabit Ethernet
Hadoop バージョン	0.21.0
ノード	マスタ 1 台，ワーカ 29 台

4.4.1 サイドデータ

サイドデータは，Map，または Reduce タスクが入力データを処理する際に必要となる，追加の読み込み専用データである．一般にデータ参照が必要となる処理の際に利用される．本研究では，p-BE，および s-BE 実装において，値が f 以上となる q^+ の照合のために 2 度目のジョブの入力として利用する．

ジョブ実行開始時，マスターノードから全てのワーカノードに対して配布され，ワーカノード上で実行されるタスクのいずれからも参照できる．1 つのワーカノード上で複数回タスクが実行される場合，タスクはサイドデータの再配布をマスタに依頼することなく，自ノード上のコピーを参照すれば良い．通信負荷の減少を期待できる．

ただし，サイドデータのデータは全ワーカノードにコピーされるため，大規模なサイドデータの配布はオーバヘッドが大きい．

4.4.2 In-Mapper Combiner

In-Mapper Combiner は，中間 KVP の総数の減少を目的とした手法である．処理内容は Combiner 処理と同様である．ただし，通常の Combiner とは異なり，Mapper のあとに MapReduce フレームワークによって任意に実行されるのではなく，開発者の意図で必ず実行される.,

In-Mapper Combiner は，Mapper ワーカノード 1 つに対してそれぞれ 1 度だけ実行される初期化 (setup) メソッドと終了 (cleanup) メソッドを利用して実装する．Map タスク単体では KVP の出力を行わず，主記憶上に保持する．ワーカノードに指定された全 Map タスクが終了時に，cleanup メソッドにて Combiner 処理をし，KVP を出力する．

これにより，中間 KVP の総数の削減に加え，次の 2 つの利点を得る．1 つ目は個々の Map タスクによるディスクの書き出しや Combiner 処理時のディスクからの読み込みが行われない点，2 つ目は In-Mapper Combiner は必ず実行されるという点である．一方で欠点はスケーラビリティの低下である．1 つのノード上での大量の出力，または多数のタスクが実行された場合に主記憶に出力結果を保持できない．

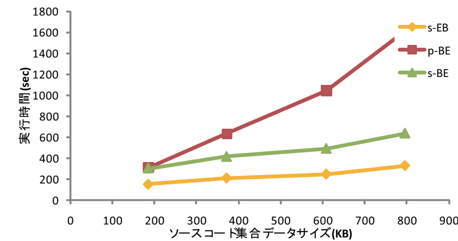

図 2 実装別実行時間

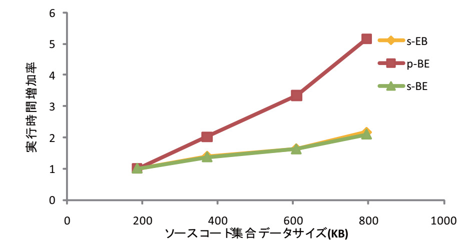

図 3 実装別実行時間増加率

5. 評価実験

5.1 評価内容

本章では，4 章で示した 3 つの MapReduce 実装の性能測定と評価を行う．それぞれの MapReduce 実装の設計の重点を基準とし，実行時間，入力データ量および中間データ量，出力データ量を測定，評価した．本実験の実験環境は３０台構成の PC クラスタである．各ノードのスペックは表 1 に示す．

実験データとして，Hadoop のソースコード集合の一部を用いた．ソースコード中の各メソッド内のメソッド呼び出し，制御構文をそれぞれ要素として抽出したデータ (特徴列データ) を入力として与える．

特徴列データのデータサイズを変化させ，実行時間，中間データ量および入出力データ量を測定した．実行時間は，各ジョブの実行時間の総和を指す．各測定値は Hadoop の WebUI と呼ばれる解析ツールを利用して測定した．ただし，ソースコード集合のデータサイズと PrefixSpan アルゴリズムの計算時間は基本的に比例しない．これは実行時間が各特徴列データの内容に依存するためである．そのため，解析規模の変化における各実装の計測要素の変化を測定するため，本研究ではデータサイズの変更の際，実験データのソースコード集合がより大きなサイズのソースコード集合の部分集合となるようにする．本稿では，実験時のソースコード集合のデータサイズは，それぞれ 186KB，372KB，609KB，および 796KB である．

また本実験において p-BE 実装の d の値は 1 とする．

© 2011 Information Processing Society of Japan

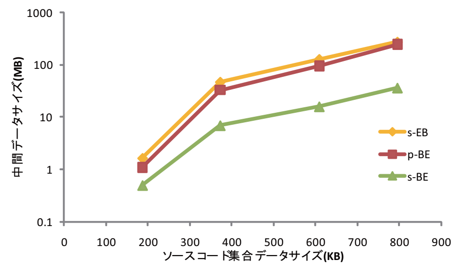

図 4　実装別中間データサイズ結果

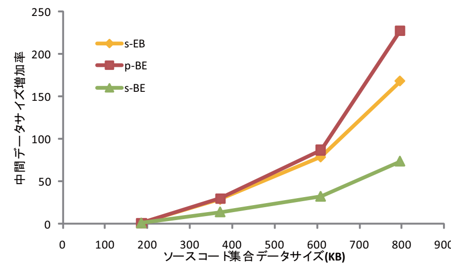

図 5　実装別中間データサイズ増加率

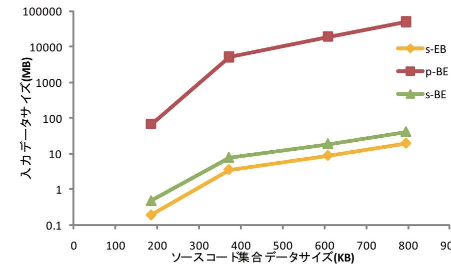

図 6　実装別入力データサイズ結果

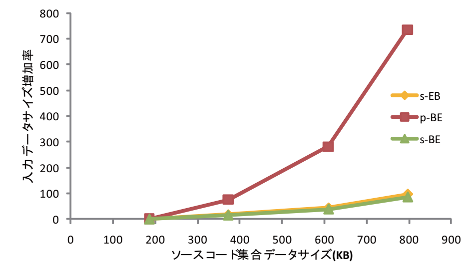

図 7　実装別入力データサイズ増加率

5.2 評価結果

各実装における実行時間を図 2 に，加えて図 3 に示す．図 3 より，3 つの実装のうち，s-EB 実装の実行時間が最も短い．s-EB 実装と比較して，p-BE，s-BE 実装はそれぞれ最大で 488%,199% の実行時間である．さらに図 3 より実行時間の増加率は s-EB 実装，s-BE 実装がほぼ同じであることに対し，p-BE 実装のみ大きいことが分かる．

要因の一つは，A を実現するためのジョブの数である．データ参照と計算時間を削減のために s-EB 実装は A を 1 つのジョブで実現した．一方で，p-BE 実装，および s-BE 実装は A を 2 つのジョブで実現した．本実験で解析をするソースコード集合のデータ量は小さい．よって，各ジョブの実行時間は計算時間ではなく，ジョブ実行の際のオーバヘッドに依存する．したがって，図 2 に示されるように s-BE 実装の実行時間は，s-EB 実装の実行時間と比較して，約 2 倍になった．

p-BE 実装の実行時間の増大には他の要因が存在すると考えられる．そこで，KVP のデータ量について比較する．

まず，中間データ量を分析した．中間データ量の増大により，Shuffle フェーズでのデータソートの計算が影響を与えていると予測する．中間データ量総和とその増加割合を図 4, 5 にそれぞれ示す．図 4 より，s-EB 実装が最も中間データが多い．s-EB 実装は実行時間が最速であるため，中間データ量は実行時間に影響を与えていないと考えられる．

また，p-BE 実装では，中間データ量の削減に重点を置いて実装したが，図 6 より，中間データ量が十分に削減できていない．この原因は中間データ (KVM_m) の Key である q^+ の派生の違いによるものである．

s-EB 実装において，入力 KVM および中間 KVM は以下の通りである．

$KVP_i(KVP_i^1) = (q, s)$, ただし $s \in T$

$KVP_m(KVP_m^1) = (q^+, n)$

このとき q^+ の種類は $|C(s)|$ に依存する．$|s|$ が大きいほど $|C(s)|$ も大きくなる傾向が高い．したがって KVP_m の数は $|s|$ に依存する..

一方で，p-BE 実装では，入力 KVM および中間 KVM は以下の通りである．

$KVP_i^1 = (q+e, U_k)$, ただし $e \in E$

$KVP_m^1 = (q^+, n)$

このとき q^+ は $|U_k|$ に依存した数の KVP_m が発生する．$|U_k| \times d = |E|$ であるため，q^+ は $|E|$ に依存するとも言える．本研究での入力データは，ソースコード集合を解析した特徴列データであるため，メソッド呼び出しの種類ごとに新たな $e \in E$ が発生する．その結果，$|s| \ll |E|$ の状態となるため，p-BE 実装における中間データ量が増大した．

次に入出力データ量を分析した．各実装の入力データ量の総和と増加率を図 6，7 にそれぞれ示す．図 6，図 7 より，p-BE 実装は他の実装と比較して大量の入出力データ量が発生することが分かる．大量の入出力データによりデータ参照が増え，計算時間が増加したことが原因であると推測する．

入力データ量が増加した原因は，入力データの生成である．p-BE 実装の入力データは，

$KVP_i^1 = (q+e, U_k)$, ただし $e \in E$

である．先の中間データと同様に $q+e$ の種類もまた $|E|$ に依存して増加する．

6. おわりに

本稿では，MapReduce を用いて 3 つの異なる方針のタスク分割方式に基づいて PrefixSpan を実装した．

- 各タスクにおけるデータ参照と計算時間を削減し，タスク実行時間を削減を図る．

© 2011 Information Processing Society of Japan

- 中間データの KVP を削減し，中間データの KVP の出力時間と，Shuffle フェーズにおける転送時間の削減を図る．
- ジョブの繰り返しにおける KVP 入出力を削減し，ジョブの入出力時間の削減を図る．

上記の方針に基づき，データ参照と計算時間に重点を置いた *s*-EB 実装，中間データに重点を置いた *p*-BE 実装，および KVP 入出力に重点を置いた *s*-BE 実装をそれぞれ設計，実装した．

評価実験の結果，*s*-EB 実装の実行時間が最速であった．要因として重点を置いたデータ参照と計算時間の削減だけでなく，KVP 入出力が他の実装と比較しても小さいことが挙げられる．

今後の研究方針は，さらに異なる着目点のタスク分割による実行時間の比較，および大規模データに対するスケーラビリティの観点からのタスク分割方針の究明である．

謝辞 本研究の一部は科学研究費補助金（基盤研究 (B)2330007 および若手 (B)23700036）の支援を受けた．

参 考 文 献

1) Dean, J. and Ghemawat, S.: MapReduce : Simplified Data Processing on Large Clusters, *Communications of the ACM*, Vol.51, No.1, pp.107–113 (2008).
2) Mccreadie, R., Mcdonald, C. and Ounis, I.: Comparing distributed indexing: to MapReduce or not?, *Proc LSDSIR*, Vol.i, No.July, pp.73–1613 (2009).
3) Pei, J., Han, J., Asl, M.B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.C.: PrefixSpan Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth, *Proc.17th Int'l Conf. on Data Eng.*, pp.215–226 (2001).
4) 伊達浩典，石尾 隆，井上克郎：オープンソースソフトウェアに対するコーディングパターン分析の適用，ソフトウェアエンジニアリング最前線 2009 (2009).
5) Apache Hadoop Project: Apache Hadoop, http://hadoop.apache.org/.
6) 小川宏高，中田秀基，広渕崇宏，高野了成，工藤知宏：高速フラッシュメモリ向け MapReduce フレームワークの実現に向けて，情報処理学会研究報告. [ハイパフォーマンスコンピューティング]， Vol.2009, No.42, pp.1–8 (2009-07-28).
7) Becerra, Y., Beltran, V., Carrera, D., Gonzalez, M., Torres, J. and Ayguade, E.: Speeding Up Distributed MapReduce Applications Using Hardware Accelerators, *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, Washington, DC, USA, IEEE Computer Society, pp.42–49 (2009).
8) Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems, *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Vol.0, Washington, DC, USA, IEEE Computer Society, pp. 13–24 (2007).
9) Catanzaro, B., Sundaram, N. and Keutzer, K.: A map reduce framework for programming graphics processors, *In Workshop on Software Tools for MultiCore Systems* (2008).
10) He, B., Fang, W., Luo, Q., Govindaraju, N.K. and Wang, T.: Mars: a MapReduce framework on graphics processors, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, New York, NY, USA, ACM, pp.260–269 (2008).
11) 高田祐輔，Heien, E.，置田真生，萩原兼一：複数の並列計算環境に対応した MapReduce の Python による実装，情報処理学会研究報告. 計算機アーキテクチャ研究会報告， Vol.2009, No.1, pp.1–7 (2009-10-19).

© 2011 Information Processing Society of Japan