MapReduce を用いたグラフアプリケーションにおける重複メッセージの排除による高速化

黒 松 信 行† 置 田 真 生† 萩 原 兼 一†

本研究は，MapReduce を用いて実装されたメッセージパッシング方式のグラフアプリケーションを対象に，MapReduce のボトルネックである Map タスクと Reduce タスク間の通信を削減する手法を提案する．まず，1 つの MapReduce ジョブ内におけるメッセージの重複を排除することで，メッセージの数を削減する．さらに，MapReduce ジョブの繰り返しにおけるメッセージパターンの重複に着目し，1 度目のパターンを保存して再利用することでメッセージ量を削減する．PageRank に対して提案手法を適用した結果，既存の高速化手法である in-mapper combining と比べ最大 1.57 倍の高速化を実現した．

Acceleration for Graph Application in MapReduce with Eliminating Redundant Messages

Nobuyuki Kuromatsu,† Masao Okita† and Kenichi Hagihara†

For MapReduce graph applications based on message passing, this papar proposes a new method to reduce communications between Map tasks and Reduce tasks, which are a bottleneck of a MapReduce job. The proposed method is a combination of the following two techniques. The first technique reduces the number of messages by removing redundant messages in a MapReduce job. The second technique reduces the size of a message by reusing message patterns of the first job in iterative jobs after the second. Experimental results show that the proposed method is up to 1.57 times faster than an existing *in-mapper combining* method for PageRank apllications with MapReduce.

1. はじめに

近年，並列分散処理によるグラフ処理が注目を集めている[1]．例えば，2012 年時点でドメインを持つウェブページは 9 億以上存在する[2]．ウェブ上の情報を解析し役立てるために，ウェブページを頂点，リンクを辺とみなすグラフ問題が用いられる．しかし，ウェブページの増加に伴って計算機単体によるグラフ解析が不可能になりつつある．

クラスタ上で大規模なデータを並列分散処理する手法として，Google が提唱した MapReduce プログラミングモデル[3] がある．MapReduce はデータを key と value の組（key-value pair，以降 KVP）として扱い，複数のクラスタノード（以降，単にノード）上で並列に処理する．MapReduce はノードを追加することで扱えるデータ量および実行速度がスケーラブルに向上する．そのため，大規模なグラフ問題は MapReduce と親和性が高い．

Jimmy Lin らはメッセージパッシングに基づくグラフアルゴリズムの MapReduce 実装を高速化する手法を提案している[4]．このアルゴリズムは次の (i)〜(iii) の繰り返しである．

(i) 全頂点において頂点の内部状態と局所的なグラフ構造に基づく計算を行う
(ii) 全頂点における (i) の結果（以降，メッセージと呼ぶ）をそれぞれの隣接頂点に渡す
(iii) 全頂点において受け取ったメッセージをもとに次の内部状態を計算する

本論文では，このアルゴリズムを Jimmy パターンと表記する．PageRank[5] をはじめ，センサネットワーク分野などグラフ問題の多くは Jimmy パターンで実装できる[6]．

Jimmy パターンを実装する場合，1 回の繰り返しを 1 回の MapReduce ジョブで実現する．Map フェーズ，Shuffle フェーズ，および Reduce フェーズ（2 章で後述）がそれぞれ (i)，(ii)，および (iii) に相当する．

† 大阪大学 大学院情報科学研究科
The Graduate School of Information Science and Technology of Osaka University

© 2013 Information Processing Society of Japan

頂点間のメッセージは Shuffle フェーズにおける KVP（以降，中間 KVP）としてノード間で送受信される．一般に，MapReduce のボトルネックは中間 KVP の生成と転送であるため[7]，高速化のためにはメッセージの削減が重要である．Jimmy らの手法は PageRank の MapReduce 実装を 1.69 倍高速化した．

我々は，Jimmy パターンを対象に，中間 KVP の重複を排除することで高速化する新たな手法を提案する．Jimmy らの既存手法は key の重複に着目して KVP を集約する一方，提案手法は value の重複に着目して KVP を統合する．

本稿では 2 章で MapReduce の仕組みについて述べる．3 章では PageRank アプリケーションのアルゴリズムを，4 章では Jimmy パターンを高速化する既存手法を紹介する．続く 5 章で提案手法の詳細を示し，6 章で実験結果を示す．最後に 7 章で本稿をまとめる．

2. MapReduce

1 回の MapReduce 実行を MapReduce ジョブ（以降，単にジョブ）と呼ぶ．ジョブは複数の Map タスクと Reduce タスクから構成される．Map タスクの数は入力データ量に比例して自動的に決定され，Reduce タスクの数はユーザが指定する．ジョブの処理の流れを図 1 に示す．1 回のジョブの流れは Map フェーズ，Shuffle フェーズ，Reduce フェーズの 3 つのフェーズに分類できる．結果を得るためにジョブを繰り返し実行する場合もある．

入力データは一定量ごとにブロックに分割され，分散ファイルシステム（DFS）を構成するノード上に分散している．Map タスクは入力データの局所性に基づいてノードに割り当てられる[3]．一方，Reduce タスクは局所性に関係なく各ノードに割り当てられる．同一のプログラムを繰り返し実行しても，ジョブごとにノードに割り当てられるタスクは変化する．出力結果は DFS に保存される．

Reduce フェーズにおける負荷分散を実現するため，MapReduce はパーティションの概念を備える．パーティションとは，KVP の key 空間を特定の規則で分割した区分である．デフォルトの規則は key のハッシュ値であるが，ユーザが自由に変更できる．各 Reduce タスクが 1 つのパーティションに属する中間 KVP を処理するため，パーティション数が Reduce タスク数と等しく，負荷が均等に分散するような規則をユーザが決める必要がある．

Map フェーズでは，各 Map タスクが 1 つの入力ブロックに対して任意の Map 処理を実行する．得られた KVP はパーティション毎にまとめられ，一定サイズごとにディスクに書き込まれる．これをスピルファイルと呼ぶ．スピルファイルの書き込みは Map 処理とオーバラップして実行される．Map タスクは全入力を処理した後，全スピルファイルの内容を key でソートし，再びディスクに書き出す（図 1(1)）．

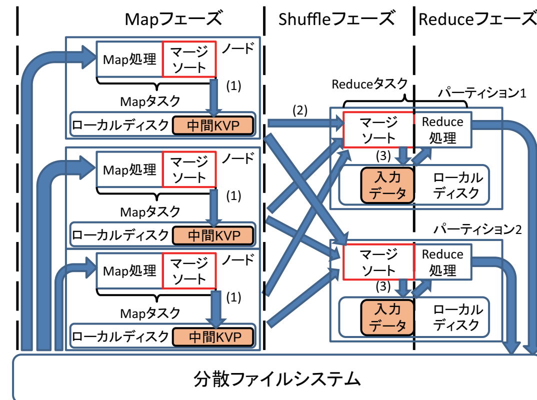

図 1　MapReduce ジョブの処理の流れ
Fig. 1　A flow of a MapReduce job

Shuffle フェーズでは Reduce タスクの入力データを作成する．まず，いずれかの Map タスクが完了すると，Reduce タスクは自身が担当するパーティションに属する中間 KVP をその Map タスクから転送する（図 1(2)）．次に，Reduce 処理の入力データを生成するため，集めた中間 KVP を key ごとにソートし，1 つのファイルにまとめディスクに書き出す（図 1(3)）．この転送およびソートは未完了の Map タスクとオーバラップして実行される．全 Map タスクの結果をソートするまで Shuffle フェーズは続く．

Reduce フェーズでは，key ごとに集められた中間 KVP を Reduce タスクが集約処理することで最終的な結果を求める．

3. PageRank

PageRank ではウェブを有向グラフ $G = (V, E)$ とみなす．V および E はそれぞれウェブページを要素とする頂点集合およびリンクを要素とする辺集合である．

PageRank の計算方法は以下の式 (1)〜(3) である[5]．式 (1) で各頂点に対し初期値のランクを与え，式 (3) を満たすまで式 (2) の計算を繰り返す．

$$P_R(v;1) = \frac{1}{|V|} \tag{1}$$

$$P_R(v;t) = \frac{(1-d)}{|V|} + d \sum_{u \in N_{in}(v)} \frac{P_R(u;t-1)}{|N_{out}(u)|} \tag{2}$$

```
1  struct Vertex{
2      double rank //現在のランク P_R(v;t)
3      int vertices // |V|
4      List<String> adjacency //N_out(v)
5  }
6  struct Message{
7      String from //メッセージを作成した頂点の識別子
8      double message //M_v
9  }
10
11 void map(String v, Vertex N){ // < v,N > を入力
12     Message M
13     M.from ← v
14     M.message ← N.rank / |N.adjacency|
15     //u ∈ N_out(v) に M_v を渡す
16     foreach (String u :N.adjacency)
17         emit(u, M) // 中間 KVP< u,M > として出力
18     //N_out(v) を次の繰り返しへ持ち越す
19     emit(v, N)
20 }
21
22 void reduce(String v, List<Object> [X_1, X_2, ...]){
23     Vertex N
24     double s ← 0
25     foreach(Object X :[X_1, X_2, ...])
26         if(isVertex(X)) //構造体 Vertex かどうか
27             N ← X
28         else
29             s ← s+X.message
30     N.rank ← (1 − d)/N.vertices+d · s //ランクの更新
31     emit(v, N) // 結果の KVP< v,N > として出力
32     // 式 (3) による終了判定（省略）
33 }
```

図 2　MapReduce における PageRank の単純実装
Fig. 2　A naive implementation of PageRank

$$\sum_{v \in V} |P_R(v;t-1) - P_R(v;t)| < \epsilon \qquad (3)$$

$P_R(v;t)$ は繰り返し $t(t>0)$ 回目の PageRank における $v \in V$ のランクを表す，$N_{out}(v) = \{u|v \to u \in E\}$ および $N_{in}(v) = \{u|u \to v \in E\}$ である（$u \to v$ は u から v への有向辺を表す）．d は減衰係数[8] を表し，リンクを持たないページのランクが増加し続けることを防ぐ．ϵ は収束の閾値であり，パラメータとして指定する．なお G は変化しないと仮定とする．

3.1 PageRank の MapReduce 実装

PageRank は Jimmy パターン（1 章）で実装できる．入力は G，出力は全ての $v \in V$ に対する $P_R(v;t)$ である．全ての v は内部状態として $P_R(v;t)$ をもち，以下の処理を繰り返す（$t > 1$）．

(i) $M_v = P_R(v;t)/|N_{out}(v)|$
(ii) 全ての $u \in N_{out}(v)$ に対して M_v をメッセージとして渡す
(iii) 受け取った全ての $M_w(w \in N_{in}(v))$ をもとに，式 (2) および式 (3) を計算

これを MapReduce を用いて単純実装した疑似コードを図 2 に示す．ジョブの入出力形式は，頂点を key とし，その頂点の内部状態および局所的グラフ構造を value とする KVP である．以降では，a および b をそれぞれ key および value とする KVP を $< a,b >$ と表記する．

ブロック数を m とすると，V は m 個の部分集合に分割され，任意の Map タスク p が部分集合 $V(p)$ を担当する．p は各 $v \in V(p)$ について図 2 の関数 map を実行する．

関数 map が生成する中間 KVP の種類は，$< u, M_v >$ および $< v,N >$ の 2 種である（図 2：17 行目および 19 行目）．v の内部情報および局所的グラフ構造（N）は Reduce の計算には不要な情報であるが，ジョブの出力 KVP を加工せずそのまま次のジョブの入力 KVP とするために Reduce タスクへ送信する必要がある．

中間 KVP の key は，メッセージを受け取る頂点を表す．パーティション数を r とすると，V は Map フェーズとは異なる基準で r 個の部分集合に分割され，任意の Reduce タスク q が部分集合 $V(q)$ を担当する．q は各 $v \in V(q)$ について図 2 の関数 reduce を実行し，更新されたランクを含む KVP を出力する．

4. 既存の高速化手法

Jimmy パターンを高速化する既存手法[4] のうち，in-mapper combining（以降，IMC）と Schimmy について示す．IMC は中間 KVP 数を削減し，Schimmy は中間 KVP のデータ量を削減する．なお，両手法は適用条件を満たせば Jimmy パターン以外に対しても適用可能である．

4.1 In-Mapper Combining

IMC は 1 つの Map タスクが生成する中間 KVP の key の重複に着目し，主記憶上で KVP を集約してからディスクへ出力する．IMC を適用できる条件は，中間 KVP をあらかじめ集約しても Reduce の計算内容が変化しないことである．

IMC は Map タスクにおいて次のように実装する．

(I) 関数 map が生成する中間 KVP を主記憶上に保存
(II) 中間 KVP を key ごとに分類し，同一 key を持つ KVP 群を集約して 1 つの KVP に変換
(III) 全入力 KVP を処理した後，集約した KVP をディスクへ出力

3.1 節の PageRank に IMC を適用すると，同一頂

点に渡すメッセージ群を集約できる．ある Map タスク p が生成する中間 KVP において，任意の頂点 v を key とする KVP の集合を $K(p|v)$，その各要素の value の集合を $U(p|v)$ とする（ただし構造体 Vertex を表す value を除く）．$U(p|v)$ は，全ての $u \in N_{in}(v) \cap V(p)$ から v が受け取るメッセージ集合に等しい．したがって，$U(p|v)$ の全要素を加算してから v に渡しても計算結果は変化しない．そこで $K(p|v)$ を 1 つの $< v, \Sigma_{M \in U(p|v)} M >$ に集約できる．

ただし，IMC を適用すると関数 map の計算と出力をオーバラップできない．IMC では Map タスクの全入力データを処理するまで (III) のディスク出力を開始できないためである．集約効率が低い入力データを処理する場合，出力が Map タスクにおけるボトルネックになりうる．

4.2 Schimmy

Schimmy はジョブの実行中に変化しない静的なデータに着目し，全ノードが静的なデータをあらかじめローカルディスクに保持することで Shuffle フェーズのデータ転送量を削減する．Schimmy を適用できる条件は，静的なデータが存在することである．

3.1 節の PageRank に Schimmy を適用すると，関数 map が出力する構造体 Vertex のデータ量を削減できる．G が静的であるため，隣接リスト $N_{out}(v)$ はジョブの繰り返しにおいて変化しない．したがって，$N_{out}(v)$ をあらかじめノードが保持していれば，中間 KVP として $N_{out}(v)$ を出力する必要はない．

しかし，あるノードが実行する Map タスクは実行のたびに変化するため，各ノードが全ての $v \in V$ に関する $N_{out}(v)$ を保持する必要がある．したがって，グラフが大規模化するにつれ各ノードが保持するデータ量が増大する．その結果，事前にデータを配布するコストが増大し，またディスク容量を超えて保持できない可能性も生じる．

5. 提案手法

我々は Jimmy パターンに対して，重複情報を排除して中間 KVP の数とデータ量を削減する手法を提案する．提案手法は 2 つの手法 RRSP（Removing Redundant messages to the Same Partition）および IRS（In-Reducer Schimmy）の組み合わせである．5.2 節で後述する IRS の特徴により，提案手法はジョブの繰り返しにおける 2 回目以降にしか適用できない．

提案手法を PageRank の MapReduce 実装（3.1 節）に適用する場合，対象となる中間 KVP の種類は $< u, M_v >$ のみである．本章では，簡単のために

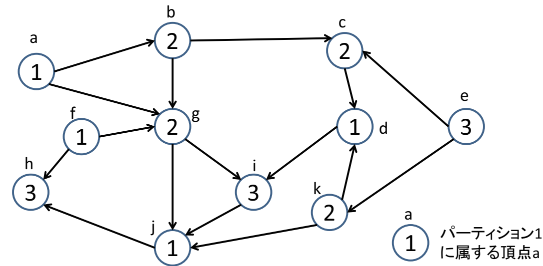

図 3 PageRank を求めるグラフの例
Fig. 3 An example of graph in PagerRank

もう一方の $< u, N >$ を無視して記述する．

5.1 RRSP

RRSP はある Map タスク p が生成する中間 KVP の value の重複に着目する．頂点 v が頂点 u と頂点 w それぞれにメッセージ M_v を渡すと仮定する．このとき，2 つの中間 KVP$< u, M_v >$ と $< w, M_v >$ が生成される．これらの KVP のパーティションが同じとき，M_v は p からある 1 つの Reduce タスクに対して 2 重に送信される．図 3 のグラフを例に PageRank の場合を考えると，頂点 b と頂点 g が属するパーティション 2 を担当する Reduce タスクは，$< b, M_a >$ と $< g, M_a >$ の 2 つを受け取る．しかし，b と g のランクを計算する上で M_a は 1 つで十分である．

そこで RRSP は，ある Map タスクが生成する中間 KVP をパーティションおよび value で分類し，同一のパーティションおよび value をもつ KVP 群を 1 つの KVP に統合する．統合後の KVP の key および value は，それぞれ統合前の KVP 群の key を要素とする集合および統合前の KVP の value とする．

PageRank は RRSP を効果的に適用できるアルゴリズムである．その理由は，頂点 v を入力とする関数 map が出力する中間 KVP の value が全て同一の M_v となるためである．ある Map タスク p が生成する統合前の中間 KVP の集合 $K(p)$ は $K(p) = \{< u, M_v > | v \in V(p), u \in N_{out}(v)\}$ となる．r 個の Reduce タスク $Q = \{q_1, q_2, \cdots, q_r\}$ がある場合，統合後の中間 KVP の集合 $K(p)'$ は $K(p)' = \{< V(q) \cap N_{out}(v), M_v > | v \in V(p), q \in Q\}$ となる．

なお，統合によって key を変更すると，パーティションが変化する可能性がある．その結果，統合後の中間 KVP が統合前と異なる Reduce タスクに送信され，正しい実行結果が得られない．

統合前のパーティションを保持するため，統合後の KVP の value に int 型の付加情報を追加する．さらに，パーティションの規則を key のハッシュ値ではなくこの付加情報に変更する．

5.1.1 IMC と RRSP の比較

RageRank を対象に IMC および RRSP の中間 KVP の削減率を比較する．入力グラフ G における頂点の出次数の平均を e，Map タスクの数を m，Reduce タスクの数を r とする．1 つの Map タスクが担当する頂点数の平均は $|V|/m$ である．各頂点は平均 e 個のメッセージを生成するため，単純実装における 1 つの Map タスクが出力する中間 KVP 数の平均は $e|V|/m$ となる．

IMC は 1 つの Map タスクが生成する中間 KVP のうち，同一の key をもつ KVP 群を 1 つの KVP に変換する．PageRank では中間 KVP の key は頂点を表すため，Map タスクが生成する中間 KVP 数は最大 $|V|$ となる．したがって IMC の削減率 D_I は式 (4) と予測できる．

$$D_I = \frac{|V|}{\frac{e|V|}{m}} = \frac{m}{e} \quad (4)$$

一方，RRSP は 1 つの Map タスクが生成する中間 KVP のうち，value が同一かつパーティションが同一の KVP 群を 1 つの KVP に統合する．PageRank では任意の頂点 v から生成するメッセージは同一であるため，v をもとに生成する中間 KVP 数は最大 r 個となる．したがって RRSP の削減率 D_R は式 (5) と予測できる．

$$D_R = \frac{r}{e} \quad (5)$$

PageRank の MapReduce 実装では，一般に $r < m$ が成り立つ．よって，IMC と比較して RRSP は中間 KVP 数をより削減すると期待できる．

なお，RRSP の処理は map 関数ごとに閉じているため，MapReduce の標準実装と同様に計算と出力のオーバラップが可能である．

5.2 IRS

IRS は，RRSP 適用後の実装に Schimmy を応用することで，中間 KVP のデータ量を削減する．Jimmy らの利用法（4.2 節）との相違点は，Reduce タスクで Schimmy を利用する点である．

PageRank を対象にした場合，統合後の中間 KVP はパーティション f ごとに $< V(q) \cap N_{out}(v), M_v >$ である（q は f を担当する Reduce タスク）．グラフ G とパーティション規則が固定であれば，$L(q|v) = V(q) \cap N_{out}(v)$ はジョブの繰り返しにおいて変化しない．あらかじめ q が $L(q|v)$ を保持していれば，$L(q|v)$ を中間 KVP に含める必要はない．この場合，q へ送信する中間 KVP は $< \emptyset, M_v >$ に変更できる．

そこで，PageRank の 1 回目の繰り返しでは，$L(q|v)$ の配布を兼ねて単純実装を実行する．任意の Reduce タスク q は，入力された KVP から v と $L(q|v)$ の対応表を作成し，ローカルディスクに保存する．2 回目以降では，提案手法を実行する．q は $< \emptyset, M_v >$ を受け取り M_v の送信元情報（v）と対応表から $L(q|v)$ を復元し，ランクを計算する．

ただし，IRS では任意の q をジョブの繰り返しにおいて同一のノードに割り当てる必要がある．しかし，MapReduce の標準スケジューラはその保証がない．そこで q に対するノードを固定するジョブスケジューラを実装した．このジョブスケジューラのデメリットとして，本来の MapReduce が備える Reduce タスクの耐故障性は失われる．性能面でのデメリットはない．

表 1 実験環境
Table 1 The experimenatal environment

CPU	Intel Xeon 3.2GHz 4 cores
主記憶	DDR3-SDRAM 1333MHz ECC 8 GB
ハードディスク	SATA 6 Gb/s 7200 rpm 1TB
ネットワーク	Gigabit Ethernet
OS	CentOS 5.6
Hadoop	バージョン 1.0.1
HDFS	上記 Hadoop に同梱版

6. 評　　価

PageRank を対象に提案手法と IMC の比較実験を行った．実行には MapReduce を実装したオープンソースソフトウェアである Hadoop[9] を用いた．実験環境を表 1 に示す．Map タスクのブロックの大きさは 64 MB とし，Reduce タスクの数はノード数と同数にした．パーティションの規則はデフォルトである．PageRank の閾値 $\epsilon = 0.005$ とした．

なお，5 章で示したように，提案手法は 1 回目のジョブには適用できない．提案手法を用いる場合でも，1 回目のジョブは図 2 の単純実装と同様の実装を用いる．2 回目以降のジョブに RRSP と IRS を適用する．

6.1 ウェブの一部分を用いた実験

実際のウェブの一部分から作成したグラフデータを用いて提案手法を評価する．英語版 Wikipedia[10] の “Portals” ページを始点とし幅優先探索でクロールした結果をもとに，$|V| = 999,991$，$|E| = 46,174,713$ のグラフデータを作成した．

このデータ量は 295 MB であり，Map タスク 5 個分の入力データに相当する．そこで，ノードを 5 台用いて PageRank を実行した．ランクが収束するまでに繰り返したジョブの回数は 10 回である．

6.1.1 実行時間の比較

各手法を用いた場合の総実行時間を図 4 に示す．単

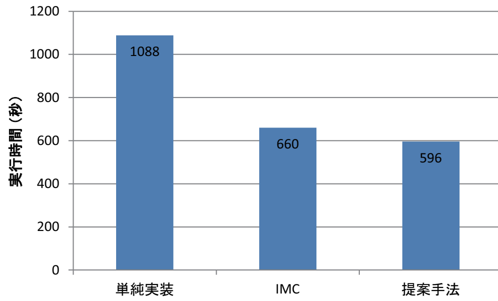

図 4 Wikipedia をもとに生成したグラフに対する PageRank の総実行時間（ジョブ 10 回の合計）

Fig. 4 Total execution time of 10 MapReduce jobs to calculate PageRank for a part of the Web

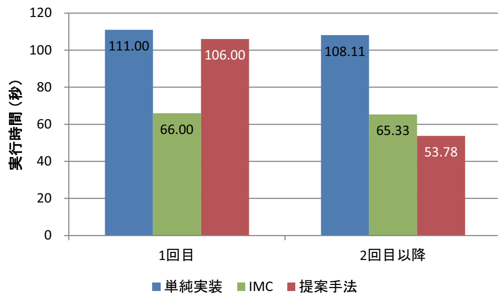

図 5 ジョブの平均実行時間

Fig. 5 Average execution time of a of job

純実装と比較して IMC が 1.65 倍高速であることに対し，提案手法は 1.83 倍高速である．

1 回目のジョブの実行時間と，2 回目以降のジョブの平均実行時間を図 5 に示す．2 回目以降のジョブにおいて，提案手法は単純実装と比べ約 2.0 倍高速である．したがって，ジョブの繰り返し回数が十分大きければ，PageRank の総実行時間も最大約 2 倍の高速化が期待できる．また，2 回目以降の提案手法は IMC と比較して 1.21 倍高速である．したがって，今回の実験では，ジョブの繰り返し回数が 6 回以上のとき提案手法の方が高速である．

6.1.2 実行の詳細比較

本節における比較の対象は，全て 2 回目以降のジョブの実験結果とする．

まず，Shuffle フェーズにおける中間 KVP のデータ転送量および中間 KVP 数を図 6 に示す．図 6 が示すとおり，提案手法は IMC と比較して中間 KVP をより削減する．提案手法のデータ転送量および中間 KVP 数は，IMC の場合のそれぞれ 16.8%および 31.6%に減少する．

次に，Map タスクおよび Reduce タスクの処理時

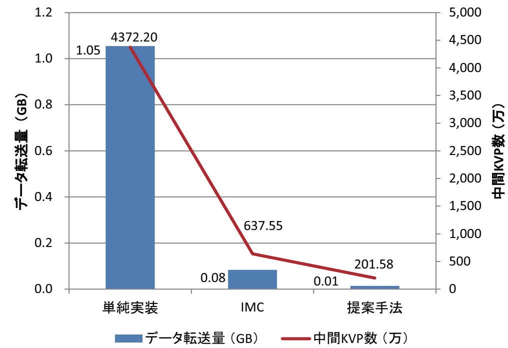

図 6 2 回目以降のジョブにおけるデータ転送量および中間 KVP 数

Fig. 6 Transfferd data size and the number of intermediate KVP in a job after the second iteration

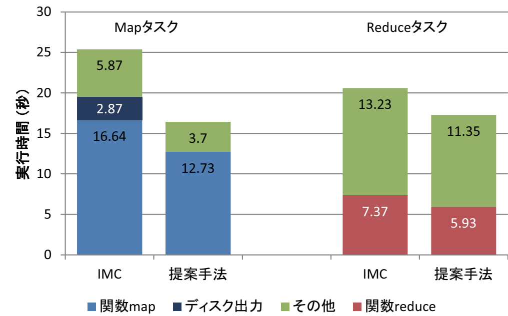

図 7 2 回目以降のジョブにおける Map タスクおよび Reduce タスクの平均処理時間の内訳

Fig. 7 Average breakdowns of a Map task and a Reduce task in a job after the second iteration

間の内訳を図 7 に示す．図のディスク出力は，4.1 節 (III) の処理に要する時間である．生成する KVP 数の減少によって，提案手法における関数 map の処理時間は IMC と比較して平均 3.9 秒短縮した．さらに，提案手法はディスク出力を関数 map とオーバラップするため，ディスク出力時間は見かけ上存在しない．Map タスクのその他は主に中間 KVP のマージソートであり，これも KVP 数の削減によって平均 1.9 秒短縮した．

提案手法における関数 reduce の処理時間は IMC と比較して平均 1.4 秒減少した．この理由は，IRS のオーバヘッドより入力 KVP 数の減少による関数 reduce の時間短縮が大きいためと考えられる．Reduce タスクのその他は，主にデータ転送とマージソートである．

6.2 ランダム生成データを用いた実験

提案手法のスケーラビリティを評価するため，ランダムに生成したグラフデータに対する PageRank の

ⓒ 2013 Information Processing Society of Japan

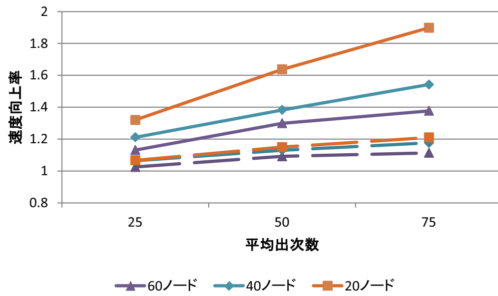

図 8 DS-L に対する提案手法と IMC の単純実装に対する速度向上率（実線：提案手法，破線：IMC）
Fig. 8 Speedup with proposed method and IMC for DS-L

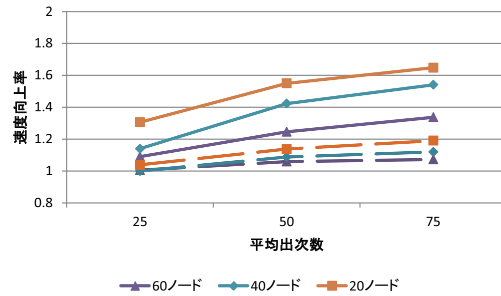

図 9 DS-S に対する提案手法と IMC の単純実装に対する速度向上率（実線：提案手法，破線：IMC）
Fig. 9 Speedup with proposed method and IMC for DS-S

表 2 ランダムに生成したグラフに対する PageRank の総実行時間（単位：秒，n：実行ノード数，e：平均出次数）
Table 2 Total execution time of PageRank for random generated graphs

	n	e	単純実装	IMC	提案手法
DS-L	60	25	1,376	1,342	1,216
		50	2,543	2,329	1,957
		75	3,865	3,471	2,807
	40	25	2,026	1,903	1,672
		50	3,966	3,509	2,868
		75	6,299	5,354	4,084
	20	25	4,076	3,820	3,088
		50	8,696	7,563	5,311
		75	1,4684	12,132	7,736
DS-S	60	25	844	839	774
		50	1,267	1,196	1,017
		75	2,031	1,894	1,519
	40	25	1,125	1,125	987
		50	1,942	1,786	1,365
		75	3,106	2,773	2,016
	20	25	2,163	2,082	1,656
		50	3,880	3,411	2,504
		75	6,450	5,418	3,914

性能を評価する．

入力として $|V|$ =1 億のグラフデータ DS-L と $|V|$ =5 千万のグラフデータ DS-S を生成した．各頂点の隣接頂点はランダムに決定している．乱数には Java の Random クラス（一様分布）を用いた．DS-L と DS-S のそれぞれで平均出次数 e を 25，50，および 75 とする計 6 種類のデータを用意した．DS-L のデータ量は $e = 25, 50, 75$ の場合でそれぞれ 24 GB，44 GB，および 66 GB であり，DS-S のデータ量はそれぞれ 12 GB，22 GB，および 33 GB である．

実行ノード数 n を 20，40，および 60 と変更しながら，6 種類のデータに対して PageRank を実行する．全ての場合においてランクが収束するまでに 5 回ジョブを繰り返した．

提案手法および IMC の総実行時間を表 2 に，単純実装に対する提案手法と IMC の速度向上率を図 8 および図 9 に示す．表 2 より，全条件において提案手法は IMC より速く，最大約 1.57 倍高速であった（DS-L に対して $e = 75$ かつ $n = 20$）．

表 3 提案手法および IMC の中間 KVP 数の削減率
Table 3 Reduction rate of the number of intermediate KVP with proposed method and IMC

	ノード数	削減率	平均出次数		
			25	50	75
DS-L	60	D_I	1.00	0.96	0.96
		D_R	0.78	0.62	0.51
	40	D_I	0.96	0.96	0.96
		D_R	0.70	0.52	0.40
	20	D_I	0.96	0.96	0.96
		D_R	0.52	0.33	0.24
DS-S	60	D_I	0.96	0.93	0.97
		D_R	0.79	0.62	0.52
	40	D_I	0.97	0.93	0.97
		D_R	0.71	0.52	0.41
	20	D_I	0.97	0.93	0.97
		D_R	0.54	0.33	0.25

6.2.1 頂点数による性能変化

図 8 および図 9 より，$n \geq 40$ かつ e が一定のとき $|V|$ を増大しても提案手法の速度向上率はほぼ変化しない．この理由は，表 3 が示すように，D_R が変化しないためである．これは D_R の予測値が $|V|$ に依存しないこと（式 (5) 参照）からも妥当である．

しかし，$n = 20$ かつ $e \geq 50$ のとき，$|V|$ が増大すると提案手法の速度向上率は増大する．この理由は Shuffle フェーズのオーバラップにある．DS-L に対して既存実装および IMC を実行した場合，全てのデータ転送を Map フェーズとオーバラップできず，関数 reduce の開始までに待ち時間が発生していた．一方で，DS-L に対して提案手法を実行した場合は，データ転送量が十分少なく，データ転送を完全にオーバラップできた．このように，中間 KVP の削減は転送待ち時間短縮の観点からも有用である．

ⓒ 2013 Information Processing Society of Japan

また，提案手法と同様に，他の条件を固定して $|V|$ を増大しても，IMC の速度向上率および D_I の実測値は変化しない．なお，m は入力データ量に比例し，入力データは隣接リストであるためそのデータ量は $|V|$ および e に比例する．式 (4) より D_I の予測値は次の式 (6) となり，$|V|$ に比例する（k は定数）．

$$D_I = k|V| \quad (6)$$

この実験では，DS-S に対してすでに D_I が上限値 (1) に近いため，DS-L に対する D_I は増加しなかった．

6.2.2 辺密度による性能変化

図 8 より，他の条件を固定して e を増大すると速度向上率は増大する．提案手法と IMC を比較すると，提案手法の速度向上率はより大きく増大する．図 9 も同様の傾向を示す．

この理由は，e が増大すると D_R が減少するためである（表 3）．式 (5) より，予測値も e に反比例する．これは e が大きいほど同一メッセージを value とする中間 KVP 数が増加するためである．なお D_I は実測値および予測値ともに e によらず一定である（式 (6)）．

以上より，入力グラフの辺密度が高いほど提案手法の性能は向上する．

6.2.3 ノード数による性能変化

図 8 より，他の条件を固定して n を増大すると速度向上率は減少する．提案手法と IMC を比較すると，提案手法の速度向上率はより大きく減少する．図 9 も同様の傾向を示す．

この理由は，n が増大すると D_R が増大するためである（表 3）．本実験では $r = n$ としたため，式 (5) より予測値も n に比例する．n が大きいほど 1 つのパーティションに送信されるメッセージが減少するためである．なお，D_I は実測値および予測値ともに n によらず一定である．

式 (5) より，n に関わらず r を一定に設定すれば，D_R の増大を回避し速度向上を期待できる．しかし，その場合 Reduce フェーズの負荷が r 個のノードに集中し，$n - r$ 個のノードが遊休状態になる．n に対して r が少ない場合は実行効率が低下し，速度向上率は減少する．したがって，n が増大した場合には適切な r を設定する必要がある．

7. ま と め

本論文は Jimmy パターンを高速化する手法を提案した．具体的には，まず，1 つの MapReduce ジョブにおいて，2 つのノード間で転送される中間 KVP の value が重複することに着目し，同一の value をもつ KVP を統合することで KVP 数を削減した．次に，MapReduce ジョブの繰り返しにおける静的なデータをあらかじめローカルディスクに保持することで KVP のデータ量を削減した．

PageRank の MapReduce 実装に提案手法を適用し，既存の高速化手法である IMC と比較して最大 1.57 倍の高速化を達成した．さらに，PageRank に対しては，一般に提案手法が IMC より高速であることおよび入力グラフの辺密度が高いほど提案手法の性能が向上することを示した．

今後の課題は，提案手法を用いる場合の適切な Reduce タスク数の決定，および Jimmy パターン以外のアプリケーションへの提案手法の応用である．

謝辞 本研究の一部は科学研究費補助金（基盤研究 (B)23300007，若手研究 (B)23700036）の支援による．

参 考 文 献

1) Lumsdaine, A., Gregor, D., Hendrickson, B. and Berry, J. W.: Challenges in Parallel Graph Processing, *Parallel Processing Letters*, Vol. 17, No. 1, pp. 5–20 (2007).
2) https://www.isc.org/solutions/survey/: The ISC Domain Survey (2012).
3) Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *CACM*, Vol. 51, No. 1, pp. 107–113 (2008).
4) Lin, J. and Schatz, M.: Design patterns for efficient graph algorithms in MapReduce, *8th Workshop on Mining and Learning with Graphs*, MLG '10, ACM, pp. 78–85 (2010).
5) Page, L., Brin, S., Motwani, R. and Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web., Technical Report 1999-66, Stanford InfoLab (1999).
6) Cudre-Mauroux, P., Aberer, K. and Feher, A.: Probabilistic Message Passing in Peer Data Management Systems, *22nd Int'l Conf. Data Engineering, ICDE '06*, IEEE, p. 41 (2006).
7) Wang, G., Butt, A., Pandey, P. and Gupta, K.: A simulation approach to evaluating design decisions in MapReduce setups, *Int'l Sympo. Modeling, Analysis Simulation of Computer and Telecommunication Systems, MASCOTS '09*, IEEE, pp. 1 –11 (2009).
8) Boldi, P., Santini, M. and Vigna, S.: PageRank as a function of the damping factor, *14th Int'l Conf. World Wide Web*, WWW '05, ACM, pp. 557–566 (2005).
9) White, T.: *Hadoop: The Definitive Guide*, Yahoo! Press, USA (2010).
10) http://en.wikipedia.org/wiki/: Wikipedia. the free encyclopedia (2012).

ⓒ 2013 Information Processing Society of Japan