MapReduce を用いた近似 K 近傍グラフの並列構築法

和良品友大† 青山 一生†† 澤田 宏††

† 奈良先端科学技術大学院大学
†† NTT コミュニケーション科学基礎研究所
E-mail: †tomohiro-w@is.naist.jp, ††{aoyama.kazuo,sawada.hiroshi}@lab.ntt.co.jp

あらまし 全 K 近傍問題を解くことによって得られる K 近傍リスト (有向 K 近傍グラフ) は，幅広い技術分野で利用されているため，この問題を効率良く解くことは重要である．本稿では，データの種類に対する制約がなく，大規模データに適用可能な発見的方法である *NN-descent* 法 に着目し，この方法の MapReduce を用いた並列実装法を提案する．提案法は，データサイズと処理を行う計算機の数とに対してスケーラブルであり，大規模データを対象とする全 K 近傍問題を効率良く解くことが可能になる．人工データを用いた実験により，提案法がこのスケーラブルな性質を有することを示す.

キーワード 全 K 近傍問題, K 近傍グラフ，Hadoop, MapReduce, 並列実装

1. はじめに

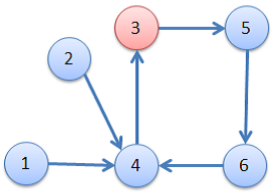
全 K 近傍問題 (All K-nearest neighbor problem; All K-NN problem) とは，与えられたオブジェクト集合の各要素について，最も類似する (近い) K 個のオブジェクトをその集合から見つける問題である．この問題の解が，各オブジェクトについて最類似の K 個のオブジェクトを列挙した K 近傍リスト (K-NN list) である．このオブジェクトを頂点とし，各オブジェクトに最も類似する K 個のオブジェクトに有向辺を張る場合，この問題は有向 K 近傍グラフ (directed K-NN graph) を作成する問題と一致する．K 近傍リスト (有向 K 近傍グラフ) は，コンピュータグラフィックス [1,2]，データクラスタリング [3,4]，次元削減 [5,6]，音声認識 [7]，類似探索 [8,9]，などの幅広い技術分野で利用されている．また，これらの多くの分野では，取り扱うデータが大規模な高次元データであることが多々ある．このため，大規模な高次元データを対象とした全 K 近傍問題を効率的に解く方法は種々の分野で必要とされている．

全 K 近傍問題は総当り法 (brute force，linear scan) で厳密に解くことができる．この方法は，対象とするオブジェクト集合の要素数を n とした場合，$O(n^2)$ の距離（又は非類似度）計算回数 (以降，計算コストと呼ぶ) を要するため，大規模データを対象とする場合には適していない．他の厳密解を求める方法に，距離空間の三角不等式等から得られる距離の下限値を用いて，探索空間を削減する効率的な方法がある [10,11]．この方法は，低次元データには有効であるが，高次元データを対象とする場合，距離の下限値が小さくなり，探索空間を効率的に削減することが困難になる．

一方，大規模な高次元データに対しては，この困難を回避するため，厳密解ではなく近似解を求めるアプローチが提案されている．このアプローチには，分割統治法 [12–14] や locality-sensitive hashing (LSH) [15,16] に代表されるハッシングを基本原理とする方法 [17] がある．これらの方法は対象とするデータや非類似度 (又は距離) に強い制約を課す．例えば，オブジェクトがユークリッド空間上の点として表現される，オブジェクト間の非類似度がユークリッド距離により定義される，などである．このため，多様な種類のデータやそのデータに固有の非類似度定義を直接扱えないという問題がある．また，近似 K 近傍リストの厳密解に対する再現率を向上させるためには，高い計算コストが必要になるという問題もある．

データや非類似度定義に対する制約がなく，大規模な高次元データを対象とした場合であっても，近似 K 近傍リストを効率的かつ高再現率で作成する発見的方法 (heuristics) に，*NN-descent* 法がある [18]．*NN-descent* 法は，K 近傍リストを有向 K 近傍グラフと捉えたときに，オブジェクト（頂点）間の関係が「近傍の近傍は近傍になりやすい」という性質を持つことを前提としている．この性質は，作成する K-NN グラフのクラスタリング係数 [19,20] が大きいという性質に相当する．*NN-descent* 法をより大規模なデータに適用する方法として，Hadoop MapReduce フレームワークで並列実装することが示唆されている [18]．しかしながら，その具体的な並列実装方法は示されていない．また，あるアルゴリズムを並列実装する場合，その性能は実装方法に大きく依存する [21]．このため，実際に *NN-descent* 法を並列実装し，その性能を評価し，有用性を示すことは重要な課題である．

本稿では，Hadoop MapReduce を用いた，*NN-descent* 法の効率的な並列実装法を提案する．提案法は，MapReduce 処理を実行する各計算機における少メモリ使用量と計算機間での少データ転送量とを同時に実現するように設計された Map 関数と Reduce 関数，及び，それらの入出力である key-value ペアを用いることを特徴とする．我々は，大規模で多様なデータに提案法を適用するための第一歩として，人工データを用いた実験で提案法を評価し，データサイズと処理を行う計算機の数



頂点v	$B(v)$	$R(v)$	$A(v)$
1	{4}	∅	{4}
2	{4}	∅	{4}
3	{5}	{4}	{4,5}
4	{3}	{1,2,6}	{1,2,3,6}
5	{6}	{3}	{3,6}
6	{4}	{5}	{4,5}

図 1　記 号 の 例

(以降，この計算機をノード，ノードの数をノード数と呼ぶ) とに対してスケーラブルであることを確認した．提案法は，与えられた大規模な高次元データから，近似 K 近傍リストを効率的に作成することを可能にする．

2. 準　　備

初めに，記号と用語を定義し，次に関連技術である *NN-descent* 法 [18] と Hadoop MapReduce [22] とを説明する．

2.1 記号と用語

本稿では，オブジェクトとそれらの関係性を頂点と有向辺を用いて有向グラフで表現する．そのため，オブジェクトと頂点とを同一視し，同じ記号を用いて表す．全 K 近傍問題の対象であるオブジェクト (頂点) 集合を V とし，そのオブジェクト (頂点) 数を $n=|V|$ とする．頂点集合 V のうち，$v\in V$ からの非類似度が最も小さい K 個の頂点を v の K 近傍頂点集合と呼び，V の全ての頂点の K 近傍頂点集合をリスト形式にまとめたものを K 近傍リストと呼ぶ．頂点 v の K 頂点集合を $B(v)$，v のリバース K 頂点集合を $R(v)=\{u\in V|\ v\in B(u)\}$，$v$ の隣接頂点集合を $A(v)=B(v)\cup R(v)$ とし，v の隣接頂点部分集合を $A_S(v)$ とする $(A_S(v)\subseteqq A(v))$．また，V の全ての頂点の K 頂点集合及び隣接頂点部分集合をリスト形式にまとめたものをそれぞれ B, A_S とする．図 1 に $V=\{1,2,3,4,5,6\}$ を用いた場合の $B(v)$，$R(v)$，$A(v)$ を示す．頂点 3 に着目すると，リバース K 頂点集合 $R(3)$ は，$R(3)=\{u\in V|\ \{3\}\in B(u)\}$ であるため，$R(3)=\{4\}$ となる．これは，グラフでは頂点 3 へ有向辺を張っている頂点集合に該当する．頂点 3 の隣接頂点集合 $A(3)$ は，$A(3)=B(3)\cup R(3)=\{5\}\cup\{4\}=\{4,5\}$ である．表 1 に本稿で用いる記号をまとめる．

2.2 *NN-descent* 法

初めに，アルゴリズムの概要把握のため，*NN-descent* 法の基本的な考え方と処理手順について簡潔に述べ，次に，効率的な処理を説明し，最後に処理の流れをまとめる．

NN-descent 法は，式 (1) の目的関数 $F(V)$ を最小にする $B(v)$ を求める発見的方法と捉えることができる．

$$F(V)=\sum_{v\in V}\sum_{u\in B(v)}\sigma(v,u) \qquad (1)$$

但し，$\sigma:\ V\times V\to\mathbb{R}$ は近傍性を評価する非類似度関数である．目的関数 $F(V)$ を最小にする $B(v)$ を各 v について総当たり法で求めるには，$\sigma(v,p)$, $p\in V\setminus\{v\}$ の計算を行う必要がある．一方，*NN-descent* 法は「近傍の近傍は近傍になりやすい」という性質を利用し，頂点 v と非類似度計算を行う頂点集合を，次のように絞り込む．まず，各頂点 v について，全頂点から無作為に選択した K 個の頂点を K 頂点集合 $B(v)$ とする．次に，頂点 v の全ての隣接頂点集合の隣接頂点集合 $\bigcup_{u\in A(v)}A(u)$ の頂点と，v との非類似度計算を行う．そして，現在の $B(v)$ の頂点より近い頂点が存在する場合，その頂点と v から最も遠い頂点 z $(z\in B(v))$ とを置換する．以上の操作を全頂点について実行し，更に $B(v)$ に変更がなくなるまで，操作を繰り返し行う．最終的に得られた各頂点の $B(v)$ をリスト形式にしたものを，近似 K 近傍リストとする．基本的な考え方と処理手順は以上である．

この処理手順を素朴に実行すると，各頂点について「隣接頂点集合の隣接頂点集合」という空間的に広域な頂点との非類似度計算を必要とする．局所的かつ効率的な処理を行うために，*NN-descent* 法は，local join，sampling, early termination を用いている．Local join とは，頂点 p の隣接頂点部分集合 $A_S(p)$ に対し，全ての異なる頂点の組み合わせ $v,u\in A_S(p)$ について非類似度を計算し，$B(v)$ または $B(u)$ の更新を行うことである．例えば，図 2 のグラフが与えられたとき (簡単のため $A_S(p)=A(p)$ とする)，p の隣接頂点 $v,u\in A(p)$ の非類似度計算と更新とを行うことは，v とその隣接頂点の隣接頂点 u との非類似度計算と更新とを行うことに相当する．提案法で利用するため，$T(v)=\{u\mid v\in A_S(p)\ \wedge\ u\in A_S(p),\ p\in V\}$，即ち，$v\in A_S(p)$ となる p を中継点とした v の隣接頂点 (p) の隣接頂点を要素として持つ集合を定義する．Local join と $T(v)$ の関係を図 2 に示す．Local join は，処理対象を局所的な頂点のみに限定するため，頂点集合 V を分割して異なる計算機に保持する Hadoop 環境（詳述は 2.3 節）では，頂点間の非類似度

表 1　記号の説明

記号	説明
V	頂点集合
n	V の頂点数
$B(v)$	v の K 頂点集合 $(v\in V)$
$R(v)$	v のリバース K 頂点集合．$R(v)=\{u\in V\mid v\in B(u)\}$
$A(v)$	v の隣接頂点集合．$A(v)=B(v)\cup R(v)$
$A_S(v)$	v の隣接頂点部分集合 $(A_S(v)\subseteqq A(v))$
$T(v)$	$T(v)=\{u\mid v\in A_S(p)\ \wedge\ u\in A_S(p),\ p\in V\}$
$C(v)$	$C(v)=B(v)\cup T(v)$
A_S	$A_S(v)$ $(\forall v\in V)$ をまとめたリスト
B	$B(v)$ $(\forall v\in V)$ をまとめたリスト
$\sigma(v,u)$	v と u の非類似度関数値
ρ	Sampling パラメータ
δ	Early termination パラメータ

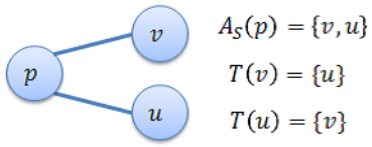


図 2　p を中継点とする local join と $T(v)$ の例

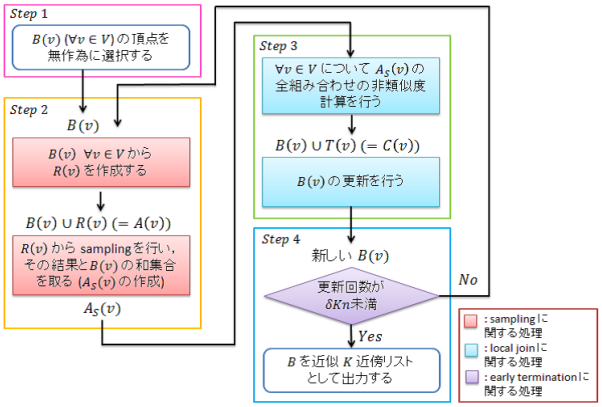


図 3　*NN-descent* 法の処理とデータの流れ

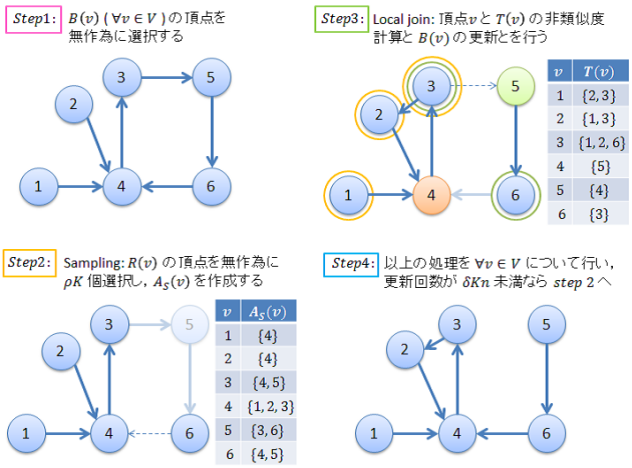


図 4　*NN-descent* 法の処理例

計算の際の計算機間のデータ転送量を低減する．Sampling は，local join を行う際に，予め設定されたパラメータ ρ を用いて，対象頂点から ρK 個を無作為に選択し，それらの頂点についてのみ非類似度計算を行う方法である．これは，処理の効率化と，Hadoop で処理を行う際に中間出力されるデータ量の大幅な削減につながる．Early termination は，予め設定されたパラメータ δ を用いて，K 近傍リストの更新回数が δKn 未満になったとき，アルゴリズムを終了させる．MapReduce ジョブは，ファイル IO とデータ転送とを行うため，処理の繰り返し回数を減らすことは重要である．このように，*NN-descent* 法は，効率的で並列実装に適したアルゴリズムである．

NN-descent 法の全体の流れを例を用いて説明する．図 3 に *NN-descent* 法の処理とデータの流れを，図 4 に図 1 のグラフを処理した際の $B(v)$ の更新の流れを示す．なお，図 4 では，パラメータ $K = 1$, $\rho = 2.0$ とした．Step 1 で全頂点 $v \in V$ について，$K(= 1)$ 頂点集合 $B(v)$ を頂点集合 V から無作為に選択し作成する (図 1 のグラフに相当する)．Step 2 で $R(v)$ から sampling を行い，その結果を用いて $A_S(v)$ を作成する．ここでは，$A_S(4)$ の作成を例に説明する．まず，頂点 4 を K 頂点集合に含む $B(1), B(2), B(6)$ から $R(4) = \{1, 2, 6\}$ を作成する．次に，$R(4)$ から $\rho K(= 2.0 \times 1 = 2)$ 個の頂点を無作為に選択する．ここでは，頂点 1, 2 が選択されたものとする．最後に，$R(4)$ の sampling 結果と $B(4)$ との和集合を $A_S(4) = \{1, 2, 3\}$ とする．以上の処理を全頂点 $v \in V$ について行う． 尚，頂点 4 以外の頂点では，$|R(v)| \leqq 2$ であるため，$A_S(v) = A(v)$ とする．Step 3 では，各 $v \in V$ で local join を行う．ここでは，頂点 3 を $A_S(p)$ に含む頂点 p で local join を行い，$B(3)$ を更新することを例に述べる．頂点 3 を $A_S(p)$ に含む頂点 p は，$p = 4, 5$ である．頂点 4 で local join を行ったときは，$A_S(4) = \{1, 2, 3\}$ の全組み合わせの非類似度 $\sigma(1, 2)$, $\sigma(1, 3)$, $\sigma(2, 3)$ を計算し，$B(1)$, $B(2)$, $B(3)$ の更新を，頂点 5 で local join を行ったときは，$A_S(5) = \{3, 6\}$ の全組み合わせの非類似度 $\sigma(3, 6)$ を計算し，$B(3)$ と $B(6)$ との更新を行う．即ち，頂点 3 と $T(3) = \{1, 2, 6\}$ との非類似度を計算し，$B(3)$ の更新することと同じ操作が行われる．ここでは，頂点 3 に最も近い頂点を頂点 2 とし，$B(3) = \{2\}$ に更新されたものとした．以上の操作を全ての頂点について行った後，Step 4 では，$B(v)$ $(\forall v \in V)$ の更新回数が δKn 未満であるという条件が満たされれば，B を近似 K 頂点リストとして出力する．前記以外の場合は，Step 2 から Step 4 を再度行う．

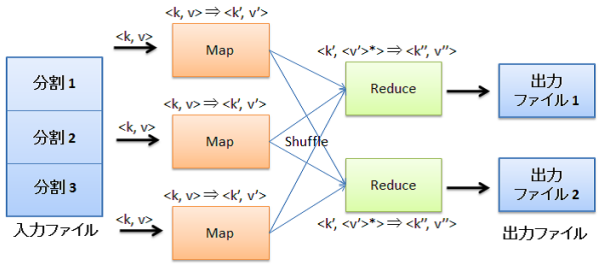


図 5　MapReduce の処理の流れ

2.3　Hadoop と MapReduce

Hadoop は Hadoop Distributed File System (以降，HDFS と呼ぶ) と Hadoop MapReduce (以降，MapReduce と呼ぶ) とを主構成要素とする分散処理フレームワークである．HDFS は，大きなファイルを「チャンク」という単位に分割して，複数の計算機に格納する分散ファイルシステムである．ユーザは，このチャンクの大きさ (以降，チャンクサイズと呼ぶ) を指定することができ，デフォルトのチャンクサイズは 64 MB である．ファイルの分割及び計算機への割り当ては，Hadoop が自動的に行うため，ユーザは 1 つのファイルシステムを利用するのと同じ感覚で HDFS を利用できる．MapReduce は，大規模データに対するスケーラブルな分散処理を表現するためのプログラミングモデルであり，計算機クラスタ上でデータ処理を行うための実行フレームワークでもある [22]．ここで，スケーラビリティとはデータサイズの増加に適応できる能力及び度合いのことを指す．

図 5 に MapReduce の処理の流れを示す．MapReduce の処理は Map, Shuffle, Reduce の 3 つのフェーズからなり，各々の入出力データは key-value ペアのデータ構造で表される．まず，Map フェーズでは，key-value ペア表現の入力データ $\langle k, v \rangle$ を受け取り，ユーザ定義の Map 処理を実行し，中間 key-value ペア $\langle k', v' \rangle$ を生成する．この Map 処理は，主に HDFS 上の

ファイルのチャンクを所持している計算機上で非同期に並列実行される．その際，1 つのチャンクにつき，1 つの Map タスクが起動するため，Map 処理全体では，チャンク数に相当する Map タスクが実行される (以降，この Map タスクの総数を起動 Map タスク数と呼ぶ)．一方，計算機クラスタ内で同時に並列実行される Map タスクの最大数 (以降，最大 Map タスク数と呼ぶ) はユーザによって決められる．起動 Map タスク数が最大 Map タスク数より大きい場合，同時処理できない Map タスクが発生する．これらの Map タスクは，現在実行中の並列処理が終了した後，同時並列処理される．次に，Shuffle フェーズでは，中間データ $\langle k', v' \rangle$ を受け取り，同じ key k' に対して value v' のリスト $\langle v' \rangle^*$ を生成し，ソートし，Reduce フェーズに渡す．最後に Reduce フェーズでは，k' と対応する v' のリスト $\langle v' \rangle^*$ を受け取り，ユーザ定義の Reduce 処理を行い，最終出力となる key-value データ $\langle k'', v'' \rangle$ を生成する．Reduce 処理は，計算機の数以下の Reduce タスク (通常，ユーザがクラスタ内の計算機数かそれより少し小さい値を指定する．以降，この数を最大 Reduce タスク数と呼ぶ) が起動し，非同期に並列実行される．このように，MapReduce ジョブは，共有メモリ領域を必要としないため，潜在的に高いスケーラビリティを持つ．

MapReduce は，煩雑な分散処理を容易に行うことを可能とする一方で，実装面では次のような制約がある．入出力データ構造は key-value ペアである．アルゴリズムは Map 処理と Reduce 処理のみで記述される．Map 処理と Reduce 処理とは非同期並列で実行されるため，計算機間でのデータの同期処理は Shuffle フェーズで実行される．これらの制約のため，アルゴリズムの性能は，key-value ペアのデータ構造の設計と MapReduce の各処理の実装方法に大きく依存する．

3. 提 案 法

本節では，提案法の概要について簡潔に述べ，2 つの 提案 MapReduce 処理である，create Adjacency Vertices (以降 *createAdjVertices*) 及び update K Vertices (以降 *updateKVertices*) について詳細に述べる．

3.1 提案法の概要

Hadoop MapReduce を用いた *NN-descent* 法の効率的な実装法を Algorithm 1 に示す．初めに，各頂点 $v \in V$ の K 頂点集合 $B(v)$ を，SAMPLE(V, K) を用いて初期化する．SAMPLE(V, K) は，頂点集合 V から K 個の頂点を無作為に選択し，選択した各頂点に v との非類似度 (設定可能な最大値，例えば，MAX_VALE) を付与する (頂点が持つ情報については，この節の最後に述べる)．次に，B の更新回数が δKn 未満になるまで次の 2 つの MapReduce 処理を繰り返す．1 つ目の MapReduce 処理である *createAdjVertices* は，入力として B を受け取り，A_S を生成し出力する．2 つ目の MapReduce 処理である *updateKVerteices* は，*createAdjVertices* の出力結果 (A_S) を受け取り，新たな B を求め，更新し出力する．最後に B の更新回数が δKn 未満であるならば，B を近似 K 近傍リストとし，出力する．

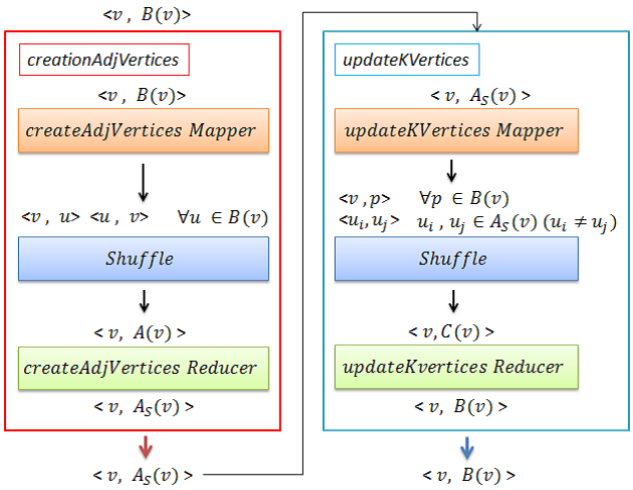


図 6　提案法の処理の流れ

Algorithm 1 NN-Descent with MapReduce

Input: V, K, ρ, δ

Output: B (approximate K-NN list)

1: $B(v) \leftarrow$ SAMPLE(V, K)　$\forall v \in V$
2:
3: **repeat**
4:　　$A_S \leftarrow createAdjVertices(B)$
5:　　$B \leftarrow updateKVerteices(A_S)$
6: **until** the number of B updates $< \delta Kn$
7:
8: **return** B

図 6 に 2 つの MapReduce 処理の流れを示す．*createAdjVertices* と *updateKVerteices* とは，各々 Map 関数と Reduce 関数とから成り，両関数の間に Shuffle フェーズを持つ．*createAdjVertices* では，まず，Map 関数が頂点 v を key，v の K 頂点集合 $B(v)$ を value とした key-value ペアを入力として受け取り，v を key, $u \in B(v)$ を value とした key-value ペアと，$u \in B(v)$ を key, v を value とした key-value ペアとをそれぞれ生成し出力する．次に，Shuffle フェーズを利用し，Map 関数の全出力から隣接頂点集合 $A(v)$ を作成する．最後に Reduce 関数が頂点 v を key, v の隣接頂点集合 $A(v)$ を value とした key-value ペアを入力として受け取り，$A(v)$ から $A_S(v)$ を生成し，v を key, $A_S(v)$ を value とした key-value ペアを生成し出力する．*updateKVerteices* では，まず Map 関数が頂点 v を key, v の $A_S(v)$ を value とした key-value ペアを受け取り，v を key, $B(v)$ を value とした key-value ペアと，$A_S(v)$ から重複を許さず構成可能な全 key-value ペア ($\langle u_i, u_j \rangle$, $u_i, u_j \in A_S(v)$ $(u_i \neq u_j)$) とをそれぞれ生成し出力する．次に，Shuffle フェーズを利用し，Map 関数の全出力から $C(v) = B(v) \cup T(v)$ を作成する．最後に Reduce 関数が頂点 v を key, $C(v)$ を value とした key-value ペアを入力として受け取り，$C(v)$ から v と最も近い K 個の頂点を抽出し新たな

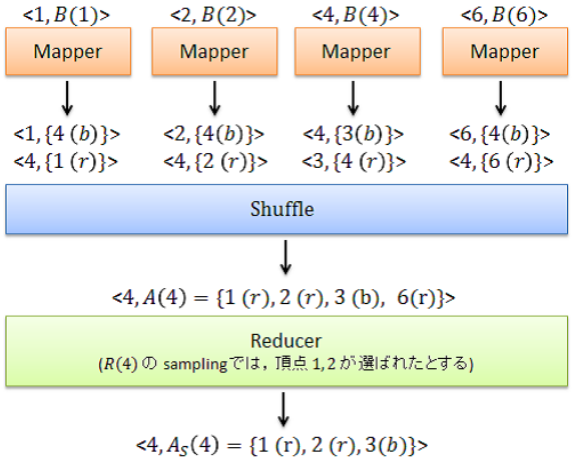


図 7　*createAdjVertices* の処理例

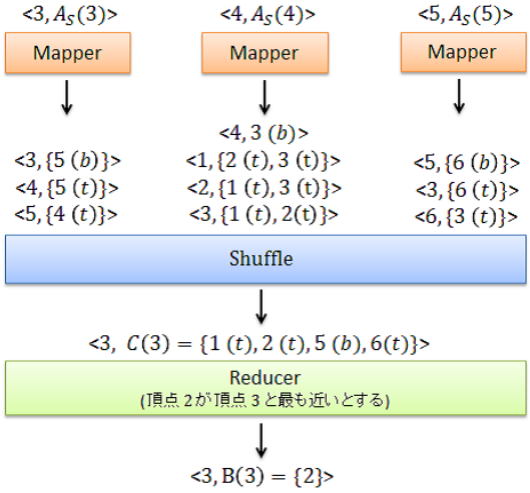


図 8　*updateKVerteices* の処理例

$B(v)$ を作成して，v を key, $B(v)$ を value とした key-value ペアを生成し出力する．

提案法は，頂点 u を key-value ペアの value として扱う時，$u \in B(v)$ と $u \in R(v)$，$u \in T(v)$ (頂点 v は，頂点 u に対応する key の頂点) を区別するため (詳細は 3.2, 3.3 節)，頂点 u に次の 3 つのシンボルのうちのいずれか 1 つを属性として与え区別する．

（1） ‘b’: u は $B(v)$ の頂点 ($u \in B(v)$)
（2） ‘r’: u は $R(v)$ の頂点 ($u \in R(v)$)
（3） ‘t’: u は $T(v)$ の頂点 ($u \in T(v)$)

更に，頂点 u には，key の頂点 v との非類似度も付与する．この時，value の各頂点は，頂点識別子 (ID)， シンボル (b,r,t), key 頂点に対する非類似度で表される．次の 2 つの節で *createAdjVertices* 及び *updateKVerteices* の詳細な実装内容を述べる．

3.2 *createAdjVertices*

Algorithm 2 に *createAdjVertices* の擬似コードを，例として図 7 に 図 1 のグラフが与えられたときの処理の流れを示す．2.2 節の例と同様に，$A_S(4)$ の作成を例に説明する．また，頂点の属性を括弧内に示す．Map 関数は，頂点 v を key，v の K 頂点集合 $B(v)$ を value とした key-value ペアを入力として受け取り，$B(v)$ の各頂点 u に属性 ‘b’ を付与し，key を v, value を u とした key-value ペアを出力する．また，v に属性 ‘ r ’，非類似度（設定可能な最大値）を付与し，$u \in B(v)$ を key, v を value とした key-value ペアを出力する．

Shuffle フェーズは，Map 関数の全出力から隣接頂点集合 $A(v)$ を作成する．$A(v)$ 作成の過程を図 7 を用いて述べる．Map 関数の全出力から頂点 4 の隣接頂点集合 $A(4)$ を作成する．頂点 4 を Map 関数で処理すると，$\langle 4, 3(b)\rangle = \langle 4, B(4)\rangle$ が出力される．また，頂点 4 を $B(u)$ に持つ頂点 u $(= 1, 2, 6)$ を Map 関数で処理すると，$\langle 4, 1(r)\rangle, \langle 4, 2(r)\rangle, \langle 4, 6(r)\rangle$ が出力される，即ち $\langle 4, R(4)\rangle$ が出力されたことになる．そして，頂点 4 を key に持つ key-value ペアを全て集めると，$\langle 4, \langle 1(r), 2(r), 3(b), 6(r)\rangle\rangle = \langle 4, B(4) \cup R(4)\rangle = \langle 4, A(4)\rangle$ になる．このようにして，Map 関数の全出力から $A(v)$ を作成する．

Reduce 関数は，頂点 v を key, $A(v)$ を value とした key-value ペアを入力として受け取り，属性を用いて $A(v)$ から $B(v)$ と $R(v)$ を作成する．次に，$R(v)$ から ρK 個を無作為に選択した結果の集合と $B(v)$ との和集合を $A_S(v)$ とする．最後に v を key, $A_S(v)$ を value にして出力する．以上のようにして B から A_S を生成する．

Algorithm 2 *createAdjVertices*

Input: B (list of K-vertex sets)
Output: A_S (list of adjacency vertex subsets)

```
1: function MAP(v, B(v))
2:     Provide attribute 'b' to all vertices in B(v)
3:     EMIT(v, B(v))
4:
5:     Provide attribute 'r' to v
6:     for all u ∈ B(v) do
7:         EMIT(u, {v(r)})
8:     end for
9: end function
10:
11: function REDUCE(v, A(v))
12:     B(v) ← all vertices with 'b' in A(v)
13:     R(v) ← all vertices with 'r' in A(v)
14:
15:     A_S(v) ← B(v) ∪ SAMPLE(R(v), ρK)
16:     EMIT(v, A_S(v))
17: end function
```

3.3 *updateKVerteices*

Algorithm 3 に *updateKVerteices* の擬似コードを，図 8 に図 1 のグラフが与えられたときの処理の流れ (図 7 の例の続き) を，2.2 節の例と同様に $B(3)$ の更新を例に示す．図 8 では，$A_S(3)$, $A_S(5)$ を，頂点 3 と 5 とに対して，図 7 の例と

同様の処理を行った後の出力結果とし，$A_S(3) = \{4(r), 5(b)\}$, $A_S(5) = \{3(r), 6(b)\}$ とする (図 4 参照).

Map 関数は，まず，頂点 v を key, $A_S(v)$ を value とした key-value ペアを受け取り，属性を用いて $A_S(v)$ から $B(v)$ を作成し，key を v, value を $B(v)$ とした key-value ペアを生成し出力する．次に $A_S(v)$ の全ての頂点に，属性 't' を付与する．最後に，$A_S(v)$ の全ての頂点 $u \in A_S(v)$ について key-value ペア $\langle u, A_S(v) \backslash \{u\} \rangle$ を生成し出力する．$A_S(v) \setminus \{u\}$ の全ての頂点 $p \in A_S(v) \backslash \{u\}$ には，非類似度 l $(= \sigma(u, p))$ を付与する．ここで出力される key-value ペアは，頂点 v を中継点とする，隣接頂点の隣接頂点の組みである．

Algorithm 3 *updateKVerteices*

Input: A_S (list of adjacency vertex subsets)
Output: B (list of K-vertex sets)

```
 1: function MAP(v, A_S(v))
 2:     B(v) ← all vertices with 'b' in A_S(v)
 3:     EMIT(v, B(v))
 4:
 5:     Provide attribute 't' to all vertices in A_S(v)
 6:     for all u ∈ A_S(v) do
 7:         for all p∈A_S(v) \ {u} do
 8:             l ← σ(u, p)
 9:             Replace p's current dissimilarity with l
10:         end for
11:         EMIT(u, A_S(v) \ {u})
12:     end for
13: end function
14:
15: function REDUCE(v, C(v))
16:     B(v) ← all vertices with 'b' in C(v)
17:     T(v) ← all vertices with 't' in C(v)
18:
19:     for all u ∈ T(v) do
20:         UPDATE(u, B(v))
21:     end for
22:     EMIT(v, B(v))
23: end function
```

Shuffle フェーズは，Map 関数の全出力から $C(v)$ を作成する．$C(v)$ 作成の過程を図 8 を用いて述べる．この例では，Map 関数の全出力から $C(3)$ を作成する．頂点 3 を Map 関数で処理すると，$\langle 3, 5(b) \rangle = \langle 3, B(3) \rangle$ が出力される．また，頂点 3 を $A_S(p)$ に持つ頂点 p $(= 4, 5)$ を Map 関数で処理すると，$\langle 3, 1(t) \rangle, \langle 3, 2(t) \rangle, \langle 3, 6(t) \rangle$ が出力される，即ち $\langle 3, T(3) \rangle$ が出力される．そして，頂点 3 を key に持つ key-value ペアを全て集めると，$\langle 3, \langle 1(t), 2(t), 5(b), 6(t) \rangle \rangle = \langle 3, B(3) \cup T(3) \rangle = \langle 3, C(3) \rangle$ になる．このようにして，Map 関数の全出力から $C(v)$ を作成する．

Reduce 関数は，頂点 v を key, $C(v)$ を value とした key-value ペアを入力として受け取り，$C(v)$ から属性を利用して $B(v)$ 及び $T(v)$ を作成する．次に，全ての $u \in T(v)$ について，$B(v)$ を UPDATE$(u, B(v))$ 関数を利用して更新する．UPDATE$(u, B(v))$ 関数は，v からの非類似度に関して，u の非類似度よりも大きい非類似度を有する頂点が $B(v)$ の中に存在した場合，u とその頂点とを置換し，B の更新回数を 1 増加する．最後に，v を key, 更新を行った新しい $B(v)$ を value とした key-value ペアを生成し出力する．以上のようにして A_S から新しい B を生成する．

表 2　実 験 環 境

計算機環境	
CPU	Intel Xeon E3-1290 v2 4 Core, 3.70-GHz
メモリ	32-GB RAM
OS	CentOS 5.8
Java	Oracle Java 1.6.0 _ 26
Hadoop 環境	
Distribution	Cloudera's Distribution, including Apache Hadoop 3 (CDH3) (注1)
最大 Map タスク数	3 × ノード数
最大 Reduce タスク数	ノード数
チャンクサイズ	64 MB
NN-descent 法のパラメータ	
K	20
ρ	0.5
δ	0.001

4. 評 価 実 験

提案法のスケーラビリティを文献 [23, 24] に記載の方法と同様に，sizeup, scaleup, speedup を用いて評価実験を行った．初めに，実験環境について述べる．次に，評価実験の詳細及びその結果について述べる．最後に，実験結果の考察を述べる．

4.1 実 験 設 定

初めに，計算機環境について述べる．計算機環境に，1 台のマスターサーバと 3 台のスレーブサーバとで構成された，4 台構成の計算機クラスタを用いた．実際に処理を行う計算機はスレイブサーバである．そのため，ノード数は 3 である．

次に，データと非類似度について述べる．データには，16 次元ユークリッド空間の単位超球上から一様ランダムに選択された 1,000,000 個 (以降，1000K と略す) の点からなる集合を用いた．非類似度には，ユークリッド距離を用いた．

最後に，実験で利用したプログラムついて述べる．*NN-descent* 法は，非類似度計算の重複を避けるために，2.2 節で述べた 3 つの手法の他に incremental search を用いている [18]．3 章では説明の簡単化のために省略したが，実際には incremental search を取り入れた実装を行なっており，評価実験ではこの実装を用いた．また，非類似度計算のために，頂点の属性には 'b', 'r', 't' のシンボル及び非類似度の他に座標情報も付与している．その他の実験設定及び環境については，表 2 にまとめた．

(注1)：
http://www.cloudera.com/content/cloudera/en/products/cdh.html

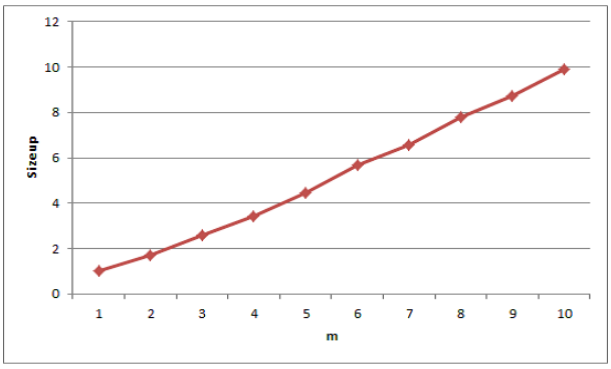


図 9　Sizeup による評価結果

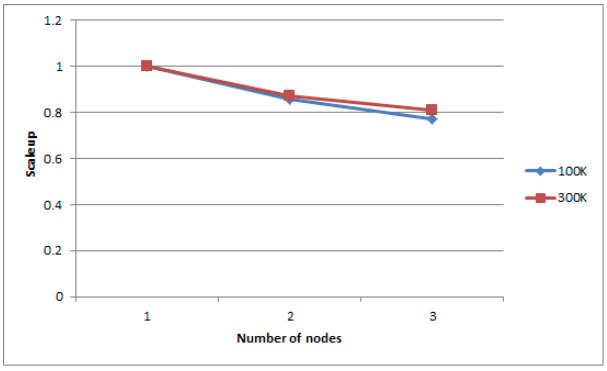


図 10　Scaleup による評価結果

4.2　スケーラビリティ性能評価

提案法のスケーラビリティ評価結果を示す．初めに，提案法のデータサイズに対するスケーラビリティを評価するために，実行時間のデータサイズ依存性を調べた．sizeup を以下に定義し，100K のデータを処理させたときの実行時間と m×100K のデータを処理させたときの実行時間とを比較し評価した．m×100K のデータを処理させたときの sizeup の値が m に近い程，スケーラビリティが高いと言える．

$$Sizeup(m) = \frac{\text{データサイズ } (m \times 100K) \text{ 時の実行時間}}{\text{データサイズ } 100K \text{ 時の実行時間}}$$

ノード数を 3 にし，m を 1 から 10 に変化させたときの sizeup を調べた．評価結果を図 9 に示す．m を増加させると，それに比例して sizeup の値も増加しており，m と近い値になっていることが確認できる．

次に，提案法のデータサイズ及びノード数の両方に対するスケーラビリティ評価を行った．scaleup を以下に定義し，ノード数を 1，データサイズを D にして処理を行わせたときの実行時間と，ノード数を m，データサイズを $m \times D$ にして処理を行わせたときの実行時間とを比較し評価した．ノード数 m，データサイズ $m \times D$ のときの実行時間が，ノード数 1，データサイズ D のときの実行時間に近ければ近い程 (scaleup が 1 に近い程)，スケーラビリティが高いと言える．

$$Scaleup(D, m) = \frac{\text{データサイズ} D, 1 \text{ノード時の実行時間}}{\text{データサイズ} m \times D, m \text{ノード時の実行時間}}$$

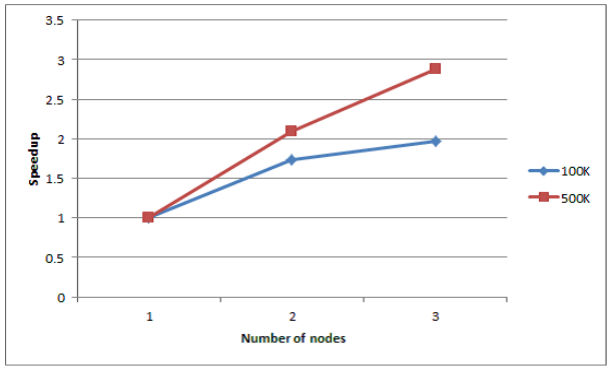


図 11　Speedup による評価結果

ノード数 m を 1 から 3，データサイズ D を 100K，300K にしたときの scaleup を調べた．評価結果を図 10 に示す．ノード数を増加させると，ノード間の通信コスト及びデータ転送コストなどのオーバーヘッドが増加するため，2 ノード以上の実行時間は 1 ノードのものとは同じにはならないが，scaleup の減少具合は緩やかでスケーラブルと言える．

最後に，提案法のノード数に対するスケーラビリティを評価するために，実行時間のノード数依存性を調べた．speedup に定義し，ノード数を 1 から m まで増加させたときの実行時間を調べた．m 台のノードで処理を行わせたときの speedup の値が m に近いほど，スケーラビリティが高いと言える．

$$Speedup(m) = \frac{1 \text{ ノード時の実行時間}}{m \text{ ノード時の実行時間}}$$

m を 1 から 3 に変化させ，それぞれの場合についての speedup を調べた．データサイズは 100K に固定したときと 500K に固定したときとでそれぞれ評価実験を行った．評価結果を図 11 に示す．500K のデータで評価を行った時，m を増加させると，それに比例して speedup の値も増加しており，m と近い値になっていることが確認できる．一方，100K のデータで評価を行った時は，$m = 3$ の speedup の値は 1.97 となり，3 よりやや低い値になった．次の節では，この原因についての考察を述べる．

4.3　考　　察

Sizeup，scaleup，500K のデータを用いた場合の speedup の評価実験により，提案法がデータサイズとノード数 m に対して共にスケールすることが確認できた．一方，100K のデータを用いて speedup 評価実験を行った時には，$m = 3$ の speedup は，理想の値である 3 より低い値になった．本節では，この原因を考察する．

我々は，ノード数に対してデータサイズが小さかったことが原因であると考える．データサイズが小さい場合に，$m = 3$ の speedup が 理想の値である 3 よりも小さくなることを，Map 処理と Reduce 処理の処理法に基づき説明する．2.3 節で述べたように，起動 Map タスク数は，チャンク数，即ち，処理対象のデータの大きさによって決められ，最大 Map タスク数はユーザによって決められる．最大 Map タスク数が起動 Map タスク数より小さい場合は，初めに最大 Map タスク数分並列

に処理を行い，処理終了後に，再び最大 Map タスク数分の並列処理を，全ての起動 Map タスクを処理し終えるまで繰り返し行う．このように，最大 Map タスク数が，起動 Map タスク数より小さい場合は，いくつかの段に分けて Map 処理を行う (以降，この段数を Map タスクの段数と呼ぶ)．100K のデータを処理させた時，Map タスクの段数は，$m = 1$ のときは *createAdjVertices* 実行時に 2 段， *updateKVerteices* 実行時に 3 段，$m = 2$ のときはそれぞれ 1 段と 2 段，$m = 3$ のときは共に 1 段になったことを確認した．以上から，ノード数を $m = 1$ から $m = 2$ に増加させた時は，両処理の Map タスクの段数が 1 段減ったことに対し，$m = 2$ から $m = 3$ に増加させた時は，*createAdjVertices* の Map タスクの段数が減らなかったことが分かる．このことが速度向上に影響を与えたと考える．

また，Reduce タスク数は，クラスタのノード数分，並列に処理を行うことが可能であるが，ノード数が多いとオーバーヘッドが増加する．そのため，ノード数が多く，かつ，データ量が小さい場合は Reduce 処理のうちオーバーヘッドの占める割合が大きくなる．従って，$m = 3$ のときは，Reduce 処理もあまり速度が向上しなかったと考える．以上から，ノード数 3，データサイズ 100K の時は，ノード数に対してデータが小さかったため， Map 処理，Reduce 処理の両方で速度が向上せず，speedup の値が低くなったと考える．前節の実験結果及び上記の考察より，データサイズが十分大きい場合には，提案法は，データサイズ及びノード数に対してスケーラブルになると考える．

5. おわりに

本稿では，MapReduce を用いた，*NN-descent* 法の効率的な並列実装法を提案した．提案法は，データサイズ及びノード数に対してスケーラブルであり，大規模データを対象とする全 K 近傍問題を効率良く解くことが可能にする．提案法のスケーラビリティを評価するため，人工データを用いて，sizeup，scaleup，speedup の 3 つの指標に関して評価実験を行い，提案法がデータサイズとノード数とに対してスケーラブルであることを確認した．今後の課題として，次の 2 つが挙げられる．1 つは，データサイズ及びノード数を増加した場合の前記評価実験の実施である．他の 1 つは，画像や文書などに代表される実データを用い，提案法を評価することである．

文　献

[1] J. Sankaranarayanan, H. Samet, and A. Varshney, “A fast all nearest neighbor algorithm for applications involving large point-clouds,” Comput. Graphics, vol. 31, pp. 157-174, 2007.

[2] M. Connor and P. Kumar, “Fast construction of k-nearest neighor graphs for point clouds,” IEEE Trans. Vis. Comput. Graphics, vol. 16, no. 4, pp. 599-608, 2010.

[3] A. K. Jain, “Data clustering: 50 years beyond K-means,” Pattern Recogn. Lett., vol. 31, pp. 651-666, 2010.

[4] C.-T. Chang, J. Z. C. Lai, and M. D. Jeng, “Fast agglomerative clustering using information of k-nearest neighbors,” Pattern Recogn., vol. 43, pp. 3958-3968, 2010.

[5] J. B. Tenenbaum, V. de Silva, and J. C. Langford, “A global geometric framework for nonlinear dimensionality reduction,” Science, vol. 290, 22 Dec., pp. 2319-2323, 2000.

[6] P. Indyk and A. Naor, “Nearst-neighbor-preserving embeddings,” ACM Trans. Algorithms, vol. 3, no. 3, article 31, August 2007.

[7] H. Lee, T.H. Wen, and L.-S. Lee, “Improved semantic retrieval of spoken content by language models enhanced with acoustic similarity graph,” Proc. IEEE SLT, pp. 182-187, December, 2012.

[8] K. Aoyama, K. Saito, H. Sawada, and N. Ueda, “Fast approximate similarity search based on degree-reduced neighborhood graphs,” Proc. ACM KDD, pp. 1055-1063, August, 2011.

[9] J. Wang and S. Li, “Query-driven iterated neighborhood graph search for large scale indexing,” Proc. ACM Multimedia, October, 2012.

[10] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, “Practical construction of k-nearest neighbor graphs in metric spaces,” Proc. Int. Workshop Experimental Algorithms, pp. 85-97, May, 2006.

[11] Y. Chen and J. M. Patel, “Efficient evaluation of all-nearest neighbor queries,” Proc. IEEE ICDE, pp. 1056-1065, April, 2007.

[12] J. Chen, H. Fang, and Y. Saad, “Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection,” JMLR, vol. 10, pp. 1989-2012, 2009.

[13] P. W. Jones, A. Osipov, and V. Rokhlin, “Randomized approximate nearest neighbors algorithm,” Proc. Natl. Acad. Sci., vol. 108, no. 38, pp. 15679-15686, September, 2011.

[14] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, “Scalable k-NN graph construction for visual descriptors,” Proc. CVPR, pp. 1106-1113, June, 2012.

[15] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” Proc. ACM STOC, pp. 380-388, May, 2002.

[16] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” Proc. ACM SCG, pp. 253-262, June, 2004.

[17] M. R. Trad, A. Joly, and N. Boujemaa, “Distributed kNN-graph approximation via hashing,” Proc. ACM ICMR, June, 2012.

[18] W. Dong, M. Charikar, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” Proc. WWW Conf., pp. 577-586, March-April, 2011.

[19] D. J. Watts, and S. Strogatz, “Collective dynamics of ‘small-world’ networks,” Nature, vol. 393, pp. 440-442, 4 June, 1998.

[20] M. E. J. Newman, “The structure and function of complex networks,” SIAM Review, vol. 45, no. 2, pp. 167-256, 2003.

[21] J. Lin, “Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-occurrence Matrices with MapReduce,” Proc. EMNLP, pp. 419–428, October 2008.

[22] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Comm. ACM, vol. 5, no. 1, pp. 107-113, 2008.

[23] X. Xu, J. Jäger, and H.-P. Kriegel, “A fast parallel clustering algorithm for large spatial databases,” Data Min. Knowl. Discov., vol. 3, no. 3, pp. 263-290, 1999.

[24] W. Zhao, H. Ma, and Q. He, “Parallel k-means clustering based on MapReduce,” Proc. CloudCom, pp. 674-679, 2009.