Vol.2014-HPC-145 No.41
2014/7/30

GPU向けの反復型グラフ処理フレームワークにおけるトポロジ変更の実現

三谷 康晃[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，GPU（Graphics Processing Unit）向けの反復型グラフ処理フレームワークにおける高速なトポロジ変更を実現するために，辺を効率よく追加および削除するためのメッセージ管理手法を提案する．提案手法は，各頂点がメッセージ受信用のリストを保持し，リストにメッセージを挿入することにより送信を実現する．このデータ構造はグラフの頂点間の接続関係に依存しないため，辺の追加および削除時にメッセージバッファの再構築は不要である．また，ダブルバッファリング技術を応用し，送信時のメッセージ複製を回避し高速化する．さらに，メッセージ挿入時に発生する，スレッド間の衝突を削減するために，挿入位置を動的に分散する．評価実験では，トポロジ変更を前提としない反復型グラフ処理フレームワーク Medusa と提案手法を性能に関して比較した．Bellman-Ford アルゴリズムに対して最大で1.3 倍の高速化を達成したが，PageRank アルゴリズムの性能は約 50〜90%低下した．

キーワード：CUDA，グラフ処理，高速化，バルク同期並列

Realizing Topology Mutation for an Iterative Graph Processing Framework on a GPU

Yasuaki Mitani[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we propose a message management method for efficient addition and deletion of edges, aiming at realizing fast topology mutation for an iterative graph processing framework on a graphics processing unit (GPU). Our method allows vertices to have own lists for receiving messages, which are sent by insertion into the lists. This data structure is independent from the connectivity of vertices of the graph, so that edge addition and deletion does not require reconstruction of message buffer. We also achieve acceleration by using a double buffering technique that avoids duplication of messages to be sent. Furthermore, our method dynamically distributes insertion positions to reduce the number of thread conflicts that can occur at insertion of a message. In experiments, we compare the performance of our method with that of Medusa, a GPU-accelerated framework that does not assume topology mutation. Our method achieves the best speedup of 1.3x for the Bellman-Ford algorithm, but decreases performance for the PageRank algorithm by 50%–90%.

Keywords: CUDA, graph processing, acceleration, bulk synchronous parallel

1. はじめに

我々の身の回りには，人間関係，ウェブページのリンク関係および道路網などの様々なネットワークが存在する．これらのネットワークをグラフとして表現し，その性質をグラフ上で分析すれば，強い影響力を持つネットワーク要素などの有益な情報が得られる．ただし，記憶装置の大容量化に起因してグラフは大規模化しているうえに，SNS（Social Networking Service）の普及に伴い，秒単位で更新されるグラフも出現している．したがって，大規模グラフを高速に処理する技術が必要である．

[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 School of Information Science and Technology, Osaka University

大規模グラフ処理の並列化を目的として，Pregel[1] などの反復型グラフ処理フレームワークが提唱されている．Pregel は，分散メモリ空間や排他制御などに起因する並列処理特有の記述を隠蔽しているため，開発者は数百台規模の PC を用いた並列処理を小さな負担で実現できる．このフレームワークは，頂点ごとの独立な処理を記述する方式を採用していて，頂点に関連付けられた重みの更新や他の頂点に対するメッセージの送受信が可能である．頂点ごとの処理はバルク同期並列モデル [2] に基づいて反復実行され，反復ごとにすべての頂点が同期することを前提としている．また，送信したメッセージは次の反復で受信側が参照できる．したがって，反復型グラフ処理フレームワークは，同期をまたいでメッセージを保持するためのメッセージバッファを必要とする．

近年, CPU の性能を凌駕するアクセラレータとして GPU（Graphics Processing Unit）[3] が注目されている．GPU は数千個ものコアを内蔵し，高い並列性を持つ演算器である．その高い性能により加速を果たす反復型グラフ処理フレームワークとして，Medusa[4] が提案されている．Medusa は，GPU における実効性能を最大化するために，グラフのトポロジに特化したデータ構造を用い，静的に確保した配列上でメッセージを管理している．したがって，グラフ上の辺や頂点を追加もしくは削除するためには，同期ののちにメッセージバッファを再構築し，再度同期する必要があり，性能に関して不利である．この点は，ポインタジャンピング [5] など，処理の過程でグラフのトポロジを変更する場合に問題となる．

そこで本研究では，GPU 向けの反復型グラフ処理フレームワークにおける高速なトポロジ変更の実現を目的として，そのためのメッセージ管理手法および GPU 実装を提案する．提案手法のデータ構造はグラフの頂点間の接続関係に依存しないため，メッセージバッファを再構築することなく，辺を追加もしくは削除でき，追加の同期も不要である．ただし，各頂点はメッセージを送信するたびにメモリ領域を動的に確保する必要がある．また，送信を高速化するために，ダブルバッファリング技術を応用しメッセージの複製を回避する．さらに，数千個もの頂点が同時に同一の頂点にメッセージを送信しうるため，それらの書き込み領域が可能な限り重複（衝突）しないように工夫する．

以降では，2 章で関連研究を紹介し，本研究の位置づけを示す．3 章で，Pregel を例として，反復型グラフ処理フレームワークの典型的な仕様について説明する．4 章で，提案するメッセージ管理手法を説明し，5 章で提案手法を評価する．6 章で本論文をまとめ，今後の課題について検討する．

2. 関連研究

Medusa[4] は，反復型グラフ処理をマルチ GPU 環境で高速化する．Medusa のメッセージバッファは，辺を介したメッセージ送受信を前提としていて，静的に確保された配列である．各配列要素は 1 つの辺に対応する．これによりトポロジを維持するグラフ処理を効率よく実行できる．しかし，グラフ上の辺を追加もしくは削除するためには，配列要素に対する辺の割り当てを変更するために，メッセージバッファを再構築する必要がある．なお，完全グラフを仮定してメッセージバッファを構築しておけば，再構築することなく辺を追加もしくは削除できる．しかし，完全グラフは $O(|V|^2)$ のメモリ領域を必要とし，扱えるグラフの規模が限られる．一方，提案手法は実際に送信されるメッセージ量だけのメモリ領域を用い，再構築することなく辺を追加もしくは削除できるため，高速なトポロジ変更を実現できる．ただし，現在の提案手法は単一 GPU を対象している．

Pregel[1] は，CPU クラスタなどの分散環境において耐故障性を持つ反復型フレームワークである．頂点間のやりとりはメッセージ通信により実現し，大規模グラフに対する並列アルゴリズムを容易に実装できる．Pregel はトポロジ変更を実現しているが，GPU による加速は提供していない．同様の実装として，Apache Giraph[6] や GPS（Graph Processing System）[7] などが知られている．

Harish ら [8] や Okuyama ら [9] は，GPU 向けの開発環境 CUDA（Compute Unified Device Architecture）[10] を用いて，グラフ処理を高速化している．CUDA による記述は，並列処理の詳細な制御を可能とする．したがって，GPU の性能を引き出せる利点を持つ．しかし，プログラムが GPU アーキテクチャに強く依存してしまう．一方，反復型グラフ処理フレームワークは，GPU アーキテクチャや計算ノード数などの詳細を隠蔽しているため，開発者の負担は小さい．

3. 反復型グラフ処理フレームワーク

Pregel[1] は，重み付き有向グラフ $G = (V, E, w)$ に対する反復処理をバルク同期並列モデル [2] に基づいて並列化する．ここで，V および $E \subseteq V \times V$ はそれぞれ G の頂点集合および辺集合を表し，関数 $w : V \cup E \to \mathbb{R}$ は頂点 $v \in V$ および辺 $e \in E$ に対する重みを定義する．バルク同期並列モデルでは，各々の反復をスーパーステップと呼び，反復ごとのバルク同期を前提としている（**図 1**）．以降では，i $(i \geq 1)$ 番目のスーパーステップを S_i と表す．

開発者は，頂点 $v \in V$ に対する 1 スーパーステップあたりの処理を関数 $P(v)$ として記述するだけで，Pregel はすべての $v \in V$ に対して $P(v)$ を適用することを反復する．$P(v)$ 内では頂点間のデータのやりとりを記述でき，次のスーパーステップに向けて他の頂点にメッセージを送信したり，直前のスーパーステップにおいて v に送信されたメッセージを参照できる．さらに，トポロジ変更操作とし

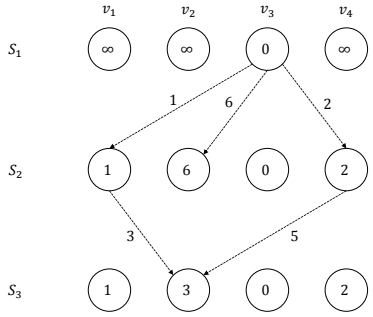


図 1 Pregel における並列実行の様子

```
1  void Compute(MessageIterator *msgs) {
2    double mindist = IsSource(vertex_id()) ? 0 : INF;
3    for ( ; !msgs->Done(); msgs->Next()){
4      mindist = min(mindist, msgs->Value());
5    }
6    if (mindist < GetValue()) {
7      *MutableValue() = mindist;
8      OutEdgeIterator iter = GetOutEdgeIterator();
9      for( ; !iter.Done(); iter.Next()) {
10       SendMessageTo(iter.Target(),
11                     mindist + iter.GetValue());
12     }
13   }
14   VoteToHalt();
15 }
```

図 2 Pregel の記述例（単一始点最短経路問題を解く Bellman-Ford アルゴリズム）

て以下を記述できる．

- 次のスーパーステップ開始時に反映される，頂点や辺の追加および削除
- 同一スーパーステップ内で即座に反映される，v に対する出辺の追加や削除

図 2 は，単一始点最短経路（SSSP：Single Source Shortest Path）問題を解く Bellman-Ford アルゴリズム [11] を Pregel で記述した例である．この問題では，重み付き有向グラフ $G = (V, E, w)$ 上のある始点 $s \in V$ から残りすべての頂点 $v \in V - \{s\}$ までの最短距離 d_v を求める．Bellman-Ford アルゴリズムは，頂点 $v \in V$ ごとにその時点での最短距離 d_v を格納し，それらを初期値 ∞ から更新していく．各反復では，v のすべての隣接頂点（$\forall u \in V \mid (v, u) \in E$）に対して d_v をメッセージとして送信する．

Pregel はオブジェクト指向型のプログラミングモデルを採用している（図 2）．各頂点 v をクラス `Vertex` のインスタンスとして表現し，関数 $P(v)$ をクラス `Vertex` のメソッド `compute()` として実装する．v が受信したメッセージは `compute()` の引数として参照でき（1 行目），v の出辺 $(v, u) \in E$ および重み $w(v)$ はそれぞれメソッド `GetOutEdgeIterator()` および `GetValue()` で参照できる．さらに，メソッド `SendMessageTo()` によりメッセージを送信できる．14 行目の `VoteToHalt()` は，v がメッセージを受信するまで自身の処理を休止するメソッドである．

3.1 Medusa のメッセージ管理手法

Medusa のメッセージバッファは配列である．頂点 v の受信バッファ B_v は v の入辺に対応する配列要素を持ち，B_v は同一配列内で隙間なく割り当てられている（**図 3**(b)）．各頂点 v は，B_v の先頭番地から $i(v)$ 個の配列要素を参照し，メッセージを受信できる．ここで，$i(v)$ は v の入次数を表す．このように，同一頂点に対する送信メッセージを配列上の連続領域に格納することにより，GPU の得意とするメモリ参照パターンを活用できる．また，Medusa のメッセージバッファは辺ごとに用意されているため，メッセージ送信時の書き込みは競合しえない．

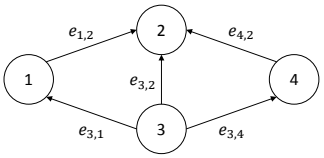
しかし，辺を追加もしくは削除するたびにメッセージバッファの割り当てを更新する必要がある．現在の Medusa では，この更新は CPU 上で処理せざるを得ない．したがって，メッセージバッファの再構築は CPU・GPU 間のデータ転送および，頂点 v ごとの $i(v)$ の計算，および v の入辺に対する順序付けを必要とし，全体性能を低下させる恐れがある．

4. 提案するメッセージ管理手法

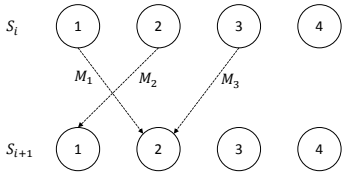
提案手法は，辺の追加や削除を迅速に終えるために，メッセージバッファの再構築が不要なデータ構造を採用する．Medusa のように辺ごとに受信バッファを静的に設けるのではなく，実際にやりとりされるメッセージごとに受信バッファを動的に確保する．これにより頂点間の接続関係に依存しないデータ構造を実現し，メッセージバッファの再構築を不要とする．この方針のもと，提案手法が解決すべき課題は以下の 3 点である．

- メッセージバッファの動的なメモリ領域確保
- メモリ領域の高速な確保および解放
- メッセージ送信時の衝突削減

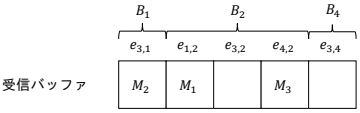
まず，メッセージは反復処理の過程で動的に構築されるため，受信バッファのためのメモリ領域を動的に確保する必要がある．次に，数万個ものスレッドを並列処理する GPU では，各スレッドが独立にメッセージを送信するため，メモリ領域の確保および解放が性能ボトルネックになりえる．最後に，複数のスレッドが同一頂点に対してメッセージを送信しうるため，受信バッファへの書き込みは排他的な制御を必要とする．この際の衝突を削減することが，並列性の高い GPU において実効性能を高めるために



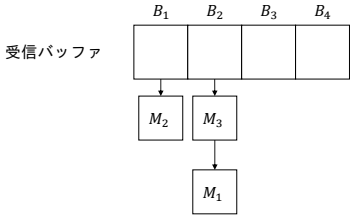
(a) グラフの例



(b) メッセージの送信例



(c) Medusa



(d) 提案手法

図 3 提案手法および Medusa のメッセージバッファ

必要である．以降，各々について述べる．

4.1 メッセージバッファの動的なメモリ領域確保

図 3(c) に，提案手法のデータ構造を示す．提案手法は，頂点ごとの受信バッファとしてリストを採用し，各リストの先頭を指すポインタを静的に確保した配列に格納する．メッセージ送信時には，送信側の頂点を担当するスレッドが送信先のリストに対してメッセージを追加する．この際，複数のスレッドが同一頂点に対してメッセージを送信しうるため，アトミック演算を用いて排他的な追加操作を実現する．提案手法は，CUDA の `atomicCAS()` 関数を用い，ロック機構が不要なリストを実装している [12]．

4.2 メモリ領域の高速な確保および解放

提案手法は，Hong ら [13] のメモリアロケータを拡張することにより，リストに対する高速な操作を実現する．このメモリアロケータは，MapReduce[14] の GPU 実装のために設計されている．メモリアロケータとしての機能の一部を制限することにより，メモリ領域の高速な確保を実現している．例えば，確保済みのメモリ領域を個別に開放することは許可されておらず，すべてのメモリ領域をまとめて開放する必要がある．その替わりに，メモリ領域の確保および開放は，それぞれ時間計算量 $O(p)$ および $O(1)$ で処理できる．ここで，p はスレッド数を表す．

提案手法は，メッセージの読み出しを終えれば，すべてのリストを開放できることに着目し，2 個のメモリアロケータをダブルバッファリングのために使う．つまり，$i\ (\geq 1)$ を非負整数として，奇数番目の S_{2i-1}（偶数番目の S_{2i}）で用いた受信バッファを S_{2i}（S_{2i+1}）の最後に開放し，S_{2i+1}（S_{2i+2}）で使用していく．このように，偶数番目および奇数番目の反復それぞれについて受信バッファを区別することにより，簡潔な開放操作を実現できる．

なお，Medusa のように，各反復処理の最後に受信バッファの内容を別のメモリ領域に退避すれば，メモリアロケータは 1 個で済む．しかし，この方針はリスト内のメッセージを計数したうえで，それらを複製する必要があり，実行効率が低下する可能性があるため採用しない．

```
1  void insert(Element **cursor, Element *target) {
2    Element **current_cursor = cursor;
3    for (;;) {
4      Element *next_element = *current_cursor;
5      if (try_insert(current_cursor, target) == SUCCESS) {
6        return;
7      }
8      next_element = *current_cursor;
9      if (truth_with_half_probability() == true) {
10       if (next_element == NULL) {
11         current_cursor = cursor;
12       } else {
13         current_cursor = &(next_element->next);
14       }
15     }
16   }
17 }
```

図 4 提案手法における挿入手順の記述例

4.3 メッセージ送信時の衝突削減

提案手法は，受信バッファへの排他的な書き込みを実現するためにアトミック関数 `atomicCAS()` を用いる．CUDA では，複数のスレッドが同一番地に対して `atomicCAS()` を試みた場合，1 個のスレッドのみが書き込みに成功し，残りは失敗する．したがって，失敗したスレッドは書き込みに成功するまで `atomicCAS()` を反復する必要がある．この反復回数を削減するために，提案手法は書き込み先を分散させることを狙う．すなわち，各スレッドの書き込み先が可能な限り重複しないように，リストへの挿入位置をスレッド間で分散させる．

図 4 に，提案手法におけるリストの挿入関数 `insert()`

を示す．この関数は，メッセージを送信するスレッドから同時に処理されうる．呼び出したスレッドは，5 行目の `try_insert()` 関数内でアトミック演算を用い，リストの先頭へのメッセージ挿入を試みる．挿入に成功したスレッドは呼び出し元に戻り，残りは引き続き挿入を試みる．この際，半数のスレッドは直前の位置のまま挿入を試みる．残りはリストを辿り，挿入位置 `current_cursor` をずらす（8〜14 行目）．なお，挿入位置がリストの末尾に達した場合，挿入位置を先頭に戻すことにより衝突回避を図る（12 行目）．

現在のスーパーステップにおいて同一頂点に対して送信されたメッセージ数を M とする．提案手法は平均的には二分木状に挿入位置を分散できるため，リストへのメッセージ挿入は，平均的には $O(\log M)$ で処理できる．一方，最悪時の時間計算量は $O(M)$ である．また，リストの走査は $O(M)$，一括開放は $O(1)$ で処理できる．

5. 評価実験

実験では，グラフ処理アルゴリズムとしてトポロジ変更が必要なものおよび不要なものを用い，性能に関して Medusa[4] と比較した．トポロジ変更が必要なものは最小全域木（MST：Minium Spanning Tree）問題を解く Borůvka アルゴリズム [15] の並列版 [16] である．一方，トポロジを変更しないものは，SSSP 問題を解く Bellman-Ford アルゴリズム [11] およびウェブページを順位付けする PageRank アルゴリズム [17] である．

表 1 に，実験で用いたグラフの特徴をまとめる．表中の i および o はそれぞれ辺の入次数および出次数を表し，$\sigma(i)$ は i の分散を表す．なお，グラフにおける辺の重み w は $1 \leq w \leq 1000$ の乱数値として与えた．RoadNet-CA はカリフォルニア州の道路網を表す無向グラフであり，WikiTalk は Wikipedia におけるユーザ間の会話を表す有向グラフである [18]．なお，WikiTalk の有向辺を逆向きにしたものが rev-WikiTalk である．R-MAT[19] および Random は，グラフ生成ツール GTGraph[20] を用いて生成した有向グラフである．前者は次数の分布が冪乗則に従うスケールフリーネットワークを表している．各グラフにつき，10 種類の重みを生成し，それらに対する実行時間の平均を計測結果とした．なお，CPU および GPU 間のデータ転送に要する時間は計測対象に含めていない．

表 2 に，実験に用いた計算機環境を示す．

5.1 トポロジを変更する場合の性能

Borůvka アルゴリズム [15] は，無向グラフに対する MST 問題を解く．各頂点は最小の重みを持つ辺で接続している頂点を自身のグループに加え，部分グラフに対する MST を得る．同様の手順にしたがい，得られた一連の MST をマージしていけば，グラフ全体の MST が得られる．各反復において，各頂点は自身が所属する MST の根を探す必要がある．この特定のために，Chung ら [16] は，トポロジ変更を伴うポインタジャンピング [5] を用いている．Borůvka アルゴリズムの特徴は，メッセージが根に集中することである．

表 1 実験で用いたグラフ

データ名	$\|V\|$ ($\times 10^6$)	$\|E\|$ ($\times 10^6$)	$\max(i)$	$\sigma(i)$	$\max(o)$	$\sigma(o)$
RoadNet-CA	2.0	5.5	12	1.0	12	1.0
R-MAT	1.0	16.0	555	23.3	1,742	32.9
Random	1.0	16.0	41	4.0	28	4.0
WikiTalk	2.4	5.0	3,311	12.2	100,022	99.9
rev-WikiTalk	2.4	5.0	100,022	99.9	3,311	12.2

表 2 実験で用いた計算機環境

項目	仕様
CPU	Intel Core i7-4770K 3.5 GHz
GPU	NVIDIA GeForce GTX 780
VRAM 容量	3 GB
OS	Ubuntu 12.04.3 LTS
開発環境	GCC 4.6, CUDA 5.5

図 5 に，提案手法，退避版および衝突版の実行時間，および Medusa のメッセージバッファ再構築時間を示す．ここで，退避版は 1 個のメモリアロケータを用いてメッセージを複製する手法であり（4.2 節），衝突版は常にリストの先頭にメッセージを挿入する手法である．また，Medusa はトポロジを変更できないため，メッセージバッファの再構築に要する時間を計測した．なお，再構築は CPU 上で処理し，CPU・GPU 間のデータ転送に要する時間を含む（3.1 節）．また，あらかじめ辺を双方向に接続することにより，有向グラフを無向グラフに変換している．したがって，rev-WikiTalk の結果は WikiTalk のものと同一である．

提案手法の実行時間は，いずれも Medusa のメッセージバッファ再構築時間よりも短い．したがって，提案手法はトポロジ変更を伴うアルゴリズムを GPU 上に実装するために有用である．なお，完全グラフを仮定してメッセージバッファを構築する場合（2 章），最も規模の小さな R-MAT に対して 4 TB のメッセージバッファが必要であり，現実的ではない．

すべてのグラフに対し，提案手法は退避版よりも実行時間を 32〜35%短縮できている．退避版は反復のたびにメッセージを複製する必要があり，そのオーバヘッドが両者の差分である．

RoadNet-CA において，提案手法は衝突版と比べて 9%速度低下している．これらの状況では入次数 i が偏らず，挿入位置を分散させるためのオーバヘッドが大きくなってしまったのが原因と思われる．一方，残りのグラフでは提案手法の方が高速である．

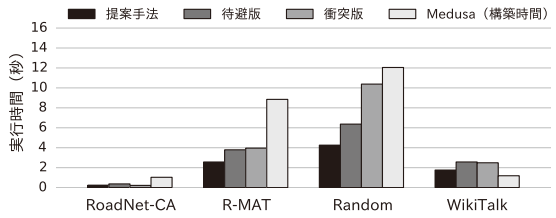


図 5 並列 Borůvka アルゴリズム（MST 問題）の実行時間および Medusa のメッセージバッファ再構築時間（秒）

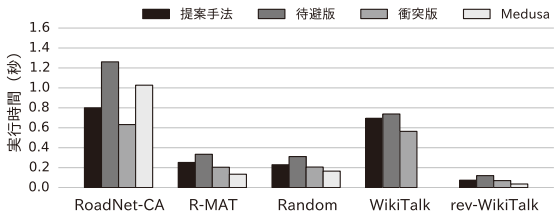


図 6 Bellman-Ford アルゴリズム（SSSP 問題）の実行時間（秒）

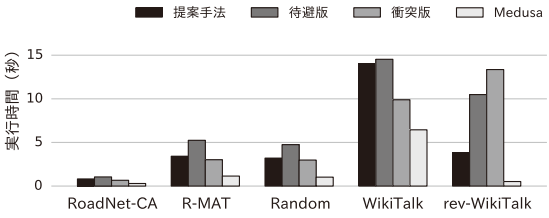


図 7 PageRank アルゴリズムの実行時間（秒）

5.2 トポロジを固定する場合の性能

Bellman-Ford アルゴリズムでは，最短経路を更新した頂点のみがメッセージを送信するため，反復回数の増大とともに頂点間の通信量が減少する．一方，PageRank アルゴリズムでは，各反復においてすべての頂点がメッセージを送信する．これらのアルゴリズムはトポロジを変更しないため，提案手法は Medusa よりも低速であることが予想される．

図 6 に，最大の出次数 o を持つ頂点を始点としたときの Bellman-Ford アルゴリズムの実行時間を示す．なお，実行の失敗が原因で一部の計測結果が存在しない．RoadNet-CA を除いて，提案手法は Medusa よりも 28〜51%低速である．しかし，RoadNet-CA に対しては予想に反して 1.3 倍高速である．高速化の理由は，Medusa のメッセージ管理手法にある．Medusa は，辺ごとにメッセージバッファを用意しているため，メッセージが送信されたか否かに関わらず，受信側は自身の受信バッファ全体を参照する．一方，リストを用いる提案手法は，実際に送信されたメッセージのみを参照できるため，Medusa よりも参照量が少ない．特に，Bellman-Ford アルゴリズムのように，一部の頂点のみがメッセージを送信する場合に両者の差が大きくなる．さらに，RoadNet-CA のように直径が大きく各頂点の次数が小さいグラフにおいては，最短距離の伝搬が遅く，更新される頂点は全体に対して比較的少ない．グラフアルゴリズムに加え，このようなグラフのトポロジに関する特徴も，両者の差を大きくしている．

提案手法は，衝突版よりも 6〜21 %低速である．この理由は，衝突の少なさおよび条件分岐のオーバヘッドにある．Bellman-Ford アルゴリズムでは，メッセージを送信する頂点が一部に限られることから，送信時の衝突は少ない．また，提案手法は条件分岐を含む（図 4 の 9 行目）．したがって，条件分岐のオーバヘッドが衝突回避による改善効果を上回り，実行効率が低下した．

図 7 に，反復回数を 100 回としたときの PageRank アルゴリズムの実行時間を示す．提案手法は R-MAT に関して性能が Medusa の約 46%に低下し，rev-WikiTalk に関しては Medusa の約 13%に低下している．Medusa のメッセージ管理手法は連続領域をまとめて読み出せるため，参照効率がよい．一方，提案手法のリストは，不連続領域の参照，挿入時のアトミック演算およびメモリの動的確保などに起因するオーバヘッドを伴う．したがって，Medusa と同数のメッセージを提案手法で処理する場合に要する実行時間は長い．

rev-WikiTalk において，提案手法は衝突版よりも 3.7 倍高速である．$\max(i)$ の大きな rev-WikiTalk では（表 1），挿入位置の衝突が頻繁に発生するため，挿入位置の分散が有効である．一方，残りのグラフに対しては，性能が 7〜29%低下した．

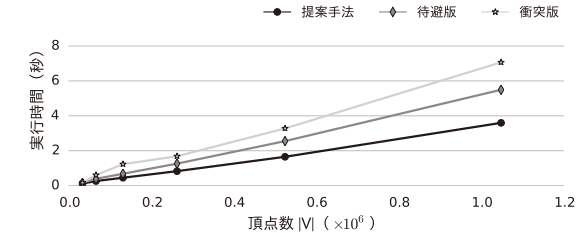
5.3 性能スケーラビリティ

グラフの規模を変動させたときの性能の振る舞いを調べるために，$|E| = 16|V|$ として，$2^{15} \leq |V| \leq 2^{20}$ の範囲で実行時間を計測した（**図 8**）．いずれのアルゴリズムにおいても，実行時間は頂点数 $|V|$ に比例している．したがって，リストに起因するオーバヘッドは大規模なグラフに対しても許容できる．ただし，トポロジ変更を伴う並列 Borůvka アルゴリズムは実行効率が少し低下していて，$|V| = 2^{20}$ の実行時間は $|V| = 2^{20}$ のときよりも 2.2 倍に増大している．

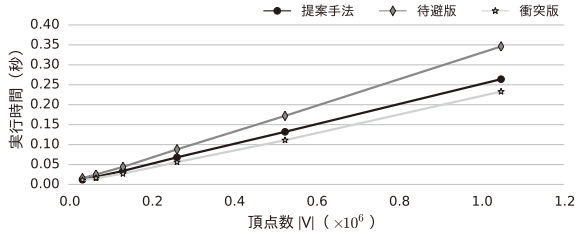
6. まとめと今後の課題

本論文では，トポロジ変更が可能な反復型グラフ処理フレームワークの GPU による実現を目的として，リストに基づくメッセージ管理手法を提案した．提案手法は，メッセージバッファを再構築することなく，辺の追加および削除を実現する．また，メッセージの送受信を高速化するために，ダブルバッファリング技術を用いてメッセージの複製を回避する．さらに，メッセージ送信時に発生しうるスレッド衝突を軽減するために，リストへの挿入位置を分散する．

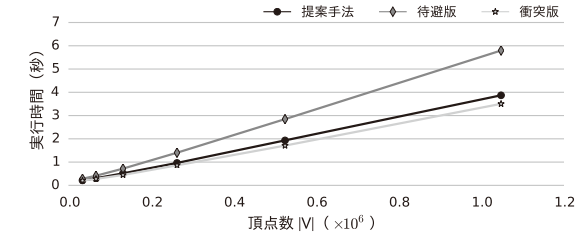
実験では，トポロジ変更を必要とする Borůvka アルゴリ



(a) MST



(b) SSSP



(c) PageRank

図 8 頂点数 $|V|$ を変化させたときの実行時間（秒）

ズムにおいて提案手法の有用性を確認した．しかし，トポロジを変更しない Bellman-Ford アルゴリズムや PageRank アルゴリズムに対しては，配列に基づく Medusa よりも 50〜90% 低速であった．ただし，一部の頂点のみがメッセージをやりとりする場合，Medusa よりも 1.3 倍高速であった．

今後の課題としては，頂点の追加や削減に対応することが挙げられる．また，複数ノードからなる分散メモリ環境において，より大規模なグラフを処理することが挙げられる．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」および科研費 25136711 の補助による．

参考文献

[1] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing, *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD'10)*, pp. 135–145 (2010).

[2] Valiant, L. G.: A Bridging Model for Parallel Computation, *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111 (1990).

[3] NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (2012). `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[4] Zhong, J. and He, B.: Medusa: Simplified Graph Processing on GPUs, *IEEE Trans. Parallel and Distributed Systems*, Vol. 25, No. 6, pp. 1543–1552 (2014).

[5] JáJá, J.: *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA (1992).

[6] The Apache Software Foundation: Giraph - Welcome To Apache Giraph! `http://giraph.apache.org/`.

[7] Salihoglu, S. and Widom, J.: GPS: A Graph Processing System, *Proc. 25th Int'l Conf. Scientific and Stastistical Database Management (SSDBM'13)* (2013).

[8] Harish, P. and Narayanan, P. J.: Accelerating Large Graph Algorithms on the GPU using CUDA, *Proc. 14th Int'l Conf. High Performance Computing (HiPC'07)*, pp. 197–208 (2007).

[9] Okuyama, T., Ino, F. and Hagihara, K.: A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-compatible GPU, *Proc. 6th Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'08)*, pp. 284–291 (2008).

[10] NVIDIA Corporation: CUDA C Programming Guide Version 5.5 (2013). `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[11] Bellman, R.: On A Routing Problem, *Quarterly of Applied Mathematics*, Vol. 16, pp. 87–90 (1958).

[12] Valois, J. D.: Lock-Free Linked Lists Using Compare-and-Swap, *Proc. 14th ACM Symp. Principles of Distributed Computing (PODC'95)*, pp. 214–222 (1995).

[13] Hong, C., Chen, D., Chen, W., Zheng, W. and Lin, H.: MapCG: Writing Parallel Program Portable between CPU and GPU, *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'10)*, pp. 217–226 (2010).

[14] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proc. 6th Symp. Operating System Design and Implementation (OSDI'04)*, pp. 137–150 (2004).

[15] Borůvka, O.: O jistém problému minimálním, *Práce Moravské Přírodovědecké Společnosti*, Vol. 3, pp. 37–58 (1927). (in Czech).

[16] Chung, S. and Condon, A.: Parallel Implementation of Borůvka's Minimum Spanning Tree Algorithm, *Proc. 10th Int'l Symp. Parallel Processing Conf. (IPPS'96)*, pp. 302–308 (1996).

[17] Page, L., Brin, S., Motwani, R. and Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web, Technical report, Stanford InfoLab (1999).

[18] Leskovec, J.: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

[19] Chakrabarti, D., Zhan, Y. and Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining, *Proc. 4th SIAM Int'l Conf. Data Mining (SDM'04)*, pp. 442–446 (2004).

[20] Bader, D. A. and Madduri, K.: GTgraph: A Synthetic Graph Generator Suite (2006). `http://www.cse.psu.edu/~madduri/software/GTgraph//`.