Spark GraphXによる最大流アルゴリズムの実装と評価

松本 拓也　佐藤 重幸　松崎 公紀

近年ビッグデータ処理の需要が高まる中，グラフ構造のビッグデータ処理を目的として様々な並列分散グラフ処理フレームワークが普及し，研究されている．しかし，グラフ処理系についての評価をする上では最大流問題を解くようなアルゴリズムはあまり用いられておらずその十分な知見がないのが現状である．広く用いられている分散処理フレームワーク 1 つとして Apache Spark が挙げられる．Spark では汎用性の高い分散処理環境を特徴として掲げており，様々なコンポーネントが提供されている．その 1 つとしてグラフ処理向けのコンポーネント Spark GraphX が提供されている．本論文では，最大流問題を解く Push-Relabel アルゴリズムについて Spark GraphX 上で 2 通りの実装し，既存の GraphX 実装や Pregel+による既存実装と比較評価を行い，その結果得られた知見を報告する．

1　はじめに

近年ビッグデータ処理の需要は高まる中，その活用される場面は非常に幅広いモノとなっている．処理対象となるデータの形式などはそれに伴い多種多様となっており，その中の 1 つとしてグラフデータが挙げられる．

グラフとは頂点とそれを何らかの情報や関係で繋ぐ辺の集合を用いて表現されるデータ構造である．グラフで表現されるデータとしては実世界や SNS（Social Networking Service）などにおけるソーシャルグラフや輸送経路や運搬コストなどを表現した輸送ネットワークグラフ，Web サイトのリンクによる関係性を表した Web グラフなどが挙げられる．

このグラフに対し，頂点や辺のデータの演算やグラフ構造の変形・分割統合などの処理を適用して問題を解き目的となるデータを算出する処理をグラフ処理と呼ぶ．実社会での応用ではこのグラフ処理によりグラフデータに対してデータ処理をすることでマーケティング戦略やコスト最適化など様々な場面で活用できる情報を得ることができる．

このようにグラフ処理の需要が高まる中，現在ではその処理対象となるデータのサイズや数が非常に膨大なものとなっている．それに伴いグラフ処理を高速処理するための手法として並列グラフ処理が注目されている．

並列グラフ処理ではグラフの頂点や辺などのデータや処理タスクを適当な並列単位で分割し，それをマルチコアプロセッサやクラスタマシンで並列処理を行う．この並列グラフ処理を実現するためのフレームワークやプログラミング言語は現在では多数存在 [8] しており，それぞれのフレームワークや並列グラフアルゴリズムについて研究が行われている．しかし，それらのパフォーマンス検証を行う際には最短距離問題やページランクのようなグラフ問題が頻繁に用いられる [5] のに対し，最大フロー問題のようなフローグラフに関して評価や実装があまり行われておらず十分な知見が得られていない．

そこで本研究では並列分散処理フレームワークである Apache Spark [11] とそのグラフ処理コンポーネントである GraphX [3] を用いて Push-Relabel アルゴリズム [1] を 2 つの方針で実装を行い，そのパフォー

---

* Implementing Maximum Flow Algorithms with Saprk GraphX.

Takuya Matsumoto, Shigeyuki Sato, Kiminori Matsuzaki, 高知工科大学, Kochi University of Technology.

マンス及び実装する上での問題点などについて考察した．

2 Push-Relabel アルゴリズム

本節では最大フロー問題についてとそれを解くための Push-Relabel アルゴリズム [1] について説明する．

2.1 最大フロー問題

フローネットワークとは例えるならばパイプ上を流れる資源や製造ラインを流れる部品のようなものをグラフとしてモデル化したものである．グラフにおける構成要素と対応付けるならばパイプや製造ラインはグラフにおける辺であり，その枝分かれ・合流する点はグラフにおける頂点，その中を流れる資源はグラフにおけるフローで表される．最大フロー問題とは，このフローネットワークと呼ばれるグラフにおいてある資源を始点とする点から十分な量だけ流入させ，ネットワーク上の各辺を流していき，最終的に終点とする別の点から流出する資源の最大値を求めるグラフ問題である．

本研究で用いるフローネットワークの定義について説明する．フローネットワークは頂点集合を V，辺集合を E としたとき，重み付き有向グラフ $G(V,E)$ で表される．グラフ G における辺の重みとは非負の整数で表される辺の容量（Capacity）であり，その辺を流れることができる資源の最大値を表す．辺 $(v,w) \in E$ におけるこの容量を $c(v,w)$ として表し，その上を流れるフローを $f(v,w)$ として表す．また，このグラフ G 上には始点（Source）s と終点（Sink）t と呼ばれる特別な 2 つの頂点 $s,t \in V$ が存在する．始点はフローネットワークにおける資源の流出する点，終点は資源が流れ着く点である．以上が本研究における最大フロー問題の対象のフローネットワークである．ただし，本研究ではこのグラフ G は自己ループ辺を持たないものとして議論を進める．

フローは容量の制限 $f(v,w) \leq c(v,w)$ を満たすと同時に，歪対称性（Skew symmetry）$f(v,w) = -f(w,v)$ も満たすものとする．全ての辺が歪対称性を満たすように，$(v,w) \in E$ かつ $(w,v) \notin E$ のときには，$c(w,v) = 0$ なる辺 (w,v) を加えるものとする．ある頂点 v に流入する資源の総和を超過（excess）と呼び，$e(v) = \sum_{(w,v)\in E} f(w,v)$ と定義される．更に，歪対称性から $\forall v \in V - \{s,t\}\ e(v) = 0$ を満たす．最大フロー問題で求める最大フローとは，これらのフローの性質を満たしたグラフにおける $e(t)$ の最大値である．

2.2 Push-Relabel アルゴリズム

本研究で実装を行った Push-Relabe アルゴリズムについて説明する．

2.2.1 フローグラフの拡張

Push-Relabel アルゴリズムを適用するにあたり前述したフローネットワークに 2 つ拡張を加える．

1 つ目に残余容量（residual capacity）と呼ばれる概念を辺に追加する．これはある辺 (v,w) について現状幾らのフローを流す余地があるかを示す概念であり，$r_f(v,w) = c(v,w) - f(v,w)$ で定義される．また，$r_f(v,w) = 0$ となっている場合その辺は飽和していると呼ぶこととする．加えて残余に関して $r_f(v,w) > 0$ となる (v,w) を残余辺と呼び，$E_f = (v,w) \in E | r_f(v,w) > 0$ を辺集合として持つ $G_f = (V, E_f)$ を残余グラフと呼ぶ．

2 つ目にラベルと呼ばれる概念を頂点に追加する．このラベルは $d(v)$ で表され，グラフ上の頂点数が n であるとして $d(s) = n, d(t) = 0, \forall_(v,w) \in E_f d(v) \leq d(w) + 1$ と定義する．このラベルは G_f 上での v と t の間の距離の下限となっており，Push-Relabel アルゴリズムではこのラベルを基に終点に向けてフローを押し流す．

最後にアクティブという概念を追加する．アクティブとは $v \in V - \{s,t\}$ に関して $e(v) > 0$ かつ $d(v) \leq \infty$ を満たす頂点を指し，この頂点はその超過 $e(v)$ を出接辺へと超過を流そうとする頂点である．

2.2.2 Push-Relabel アルゴリズムの概要

ここでは Push-Relabel アルゴリズムの概要を簡単に説明する．

Push-Relabel アルゴリズムではまず対象のグラフに対して始点の出接辺を飽和させ，他の辺をすべてフロー量を 0 として初期化する．また，始点のラベルを頂点数 n として初期化する．この初期化されたフ

ローグラフに対して Push-Relabel アルゴリズムを適用していく．

Push-Relabel アルゴリズムでは Push と Relabel と呼ばれる 2 つの操作をアクティブな頂点に対して可能な限り適用を繰り返し，操作の適用とメッセージ送信，そして受信したメッセージの適用を幾度も行い最終的にグラフ上にアクティブな頂点が存在しなくなると結果の収束とみなして処理を終える．この時の $e(t)$ が最大フローの解答となる．

続いて Push-Relabel 処理の核となる Push と Relabel について述べていく．Push はアクティブな頂点 v に流れてきている超過を隣接する頂点 w へと押し出す処理である．この Push 処理では頂点がアクティブである他に Push を行う辺 (v, w) に関して $d(v) = d(w)+1$ であることが Push 処理の条件となっている．この条件に合う辺について $e(v)$ を $r_f(v, w)$ 以下の範囲で頂点 v から w へ (v, w) を通して伝播させていく．この処理は $e(v) > 0$ でありかつすべての (v, w) が飽和するまで可能な限り Push を適用する．この処理によりフローグラフ内でフローをメッセージとして隣接する頂点へと送信して伝播させていく．この Push において伝播するこのフロー量を δ と呼び，この δ を受信した頂点 w は自らの超過 $e(w)$ に δ を加算して新たな $e(w)$ の値とする．

Relabel はアクティブな頂点 v に対して隣接頂点の $d(w)$ の情報を元に必要に応じてラベルを更新する処理である．Relabel 操作も Push 操作と同様にアクティブ以外の条件が存在する．Relabel の場合は Relabel 処理の対象となる v とその隣接点 w を結ぶ辺 (v, w) について $r_f(v, w) > 0$ であることが条件となっている．この処理では条件に合う辺に関して $d(w)$ の情報を頂点 v へとメッセージとして送信し，その中でも最も小さい $d(w)$ に 1 を加えた値を新たなラベル $d(v)$ として設定する．

以上のようにこの Push と Relabel の 2 つの処理を用いてグラフデータの処理やメッセージ送信を繰り返し最大フローを求める．

3 Spark GraphX

本節では前節において説明した Push-Relabel アルゴリズムを本研究で実装するにあたって用いた Apache Spark [11] 及び GraphX [3] についての概要を述べていく．

3.1 Apache Spark

Apache Spark とは，大規模データ処理を目的としたオープンソースの並列分散処理フレームワークである．2009 年にカルフォルニア大学バークレー校 AMPLab にて開発が行われ現在では Apache のトップレベル Project の 1 つとして開発が行われている．

Spark における特徴の 1 つとしてオンメモリでの分散処理に特化しているという点が挙げられる．Spark では内部処理において処理データをメモリ上に保存することにより従来よく用いられていた MapReduce のような処理手法と比べてストレージとの IO によるオーバーヘッドを削減し高速なレスポンスを実現している．これにより MapReduce などが不得手としていたループ処理で何度もデータを処理・参照する必要のある数値処理や機械学習，対話型の処理，ストリーミングデータ処理などを効率的に処理することを実現している．

また，Spark では処理データを扱うデータセットとして RDD（Resilient Distributed Dataset）と呼ばれるイミュータブルな分散コレクションを採用しており，この RDD とそれに対するオペレーションで Spark の処理は構成される．この Spark の処理の一連の流れは DAG（Directed Acyclic Graph）で表現される．Spark ではこの RDD と DAG を用いて分散環境における安全性や耐障害性，処理の効率化などを実現している．

もう 1 つの Spark における大きな特徴として Spark Core と呼ばれる基本的な分散処理に加えて SQL やストリーミング処理，機械学習，グラフ処理など様々なコンポーネントを実装しており幅広いアプリケーションで容易に使用・連携を可能としている点が挙げられる．前述した通り，Spark では全ての処理を RDD とそのオペレーションで構成しているがこれは他のコン

ポーネントでも同様であり，それぞれ扱い方に差はあるが基本的に同じデータ構造を用いることからシームレスな処理の連携やデータの変換が実現されている．これにより Spark は従来の並列分散処理フレームワークの中でも高い汎用性を実現している．

3.2 GraphX

GraphX は Spark のコンポーネントの 1 つであり，Spark 環境上でグラフ計算を行うことを目的として開発されたものである．

Spark プログラムでは RDD を用いてデータ表現を行っていたがこれは GraphX でも同様であり，RDD の拡張データ型を用いてグラフデータを表現している．GraphX におけるグラフ表現とはグラフの頂点データの集合を VertexRDD と呼ばれるコレクションに格納し，辺データの集合を EdgeRDD と呼ばれるコレクションに格納し，これらを Graph 型というデータでラップすることで頂点データ集合と辺データ集合を関連付けたプロパティグラフによってグラフデータを管理している．

GraphX で用いられるこの VertexRDD や EdgeRDD は RDD の拡張であるため Spark Core で用いる Spark 処理をある程度そのまま適用することも可能であるし，GraphX で実装されているグラフ処理向けのデータ処理メソッドを用いてグラフ処理を実装する事もできる．その為前述した様にグラフ処理とその他のアプリケーションの連携がシームレスに行えるということがわかる．GraphX において実装されているグラフ処理 API では基本的な map 処理から aggregate 処理，Pregel モデルに基づいた PregelAPI，PageRank 処理など様々なものが提供されており，これらを用いてグラフ処理の実装を行う．

4 Push-Relabel アルゴリズムの Spark GraphX 実装

本節では Push-Relabel アルゴリズムの Spark GraphX 上での実装について述べていく．また，比較対象として，GraphX による既存実装 [6] も紹介する．

表 1　実装 A におけるグラフの型情報

	型	格納される情報
VertexData	(Int, Int)	$(e(v), d(v))$
EdgeData	(Int, Int, Int)	$(r_f(v,w), r_f(w,v), \delta)$

4.1 実装方針について

本研究では GraphX による Push-Relabel アルゴリズムの実装をするに辺り 2 つの実装方針を立てて実装を進めた．1 つ目は GraphX の設計思想に沿った実装，2 つ目は Pregel [7] と呼ばれる並列グラフ処理モデルを基にした実装である．どちらの実装方針でも基本的には同等の処理を実装しておりアルゴリズムレベルでは同じものであるが，それぞれが用いるデータ構造の定義やそれに伴う処理手順の差異などにより多々異なる点が存在しているため次項よりそれぞれ説明していく．

4.2 A 実装

A 実装では Push-Relabel アルゴリズムを実装するにあたり，GraphX の設計に基づいた実装を試みた．つまりは Push-Relabel アルゴリズムを GraphX を用いて素直に実装したものとなっている．具体的には GraphX において頂点に関するデータは VertexRDD へと格納し，辺に関するデータは EdgeRDD へと格納するというデータ構造の設計と可能な限り GraphXAPI によるグラフ操作を活用するという点を意識して実装を行った．

まず本実装で用いたグラフデータのデータ型定義を表 1 に示す．VertexRDD は頂点が持つ情報として頂点 v における $e(v)$ と $d(v)$ の 2 つを持たせている．EdgeRDD では頂点が持つ情報として辺 (v,w) における $r_f(v,w)$ と $r_f(w,v)$，そして (v,w) 上を伝播するメッセージ δ の 3 つを持たせることとした．

今回実装した A 実装の Push 処理を図 1 に示す．Phase1 では aggregateMessages メソッドを用いて辺 (v,w) に関してアクティブな頂点に対して自らの辺情報をメッセージ送信を行う．ここで用いた aggregateMessages メソッドだが，トリプレット（辺 (v,w) と頂点 v,w の 3 つを 1 つの単位とするデータ構造）の情報を基にその接続頂点へと情報集約をするメ

```
var activeSubgraph = resultGraph.subgraph(isActive)
//Push:Phase1
val pushMsg = activeSubgraph.aggregateMessages[(Int, Seq[(VertexId, Int, Int, Boolean)])](
   //sendMessege
   triplet => {
      if (isActive(triplet.srcId, triplet.srcAttr) &&
          triplet.srcAttr._2 == triplet.dstAttr._2 + 1) {
         triplet.sendToSrc(triplet.srcAttr._1,
                           Seq((triplet.dstId, triplet.attr._1, triplet.attr._2, true)))//send to v
      } else if (isActive(triplet.dstId, triplet.dstAttr) &&
          triplet.dstAttr._2 == triplet.srcAttr._2 +1) {
         triplet.sendToDst(triplet.dstAttr._1,
                           Seq((triplet.srcId, triplet.attr._1, triplet.attr._2, false)))//send to w
      }
   },
   //mergeMsg
   (msg1, msg2) => (msg1._1, msg1._2 ++ msg2._2)
)
//Push:Phase2
val pushedData = pushMsg.map{case (vId, (ev, list)) =>
   var nextEv = ev//push 後の e(v)
   var delta = 0 //delta
   var edges = ArrayBuffer[Edge[(Int, Int, Int)]]()
   for ((wId, frf, brf, isSentForward) <- list){
      if (isSentForward) {
         delta = if(nextEv <= frf) nextEv else frf //min(e(v), residual)
         nextEv -= delta
         edges += Edge(vId, wId, (frf - delta, brf, delta))
      } else {
         delta = if(nextEv <= brf) nextEv else brf //min(e(v), residual)
         nextEv -= delta
         edges += Edge(wId, vId, (frf, brf - delta, - delta))
      }
   }
   (vId, (nextEv, edges))
}
//Push:Phase3
val pushedERDD = pushedData.flatMap({case (_, (_, edges)) => edges})
val augmentedERDD = activeSubgraph.subgraph(epred = edge => !(isActive(edge.srcId, edge.srcAttr) &&
                                                              edge.srcAttr._2 == edge.dstAttr._2+1)
   .mapEdges(edge => (edge.attr._1, edge.attr._2, 0)).edges.union(pushedERDD)
val evUpdatedGraph = activeSubgraph.joinVertices(pushedData)((vid, vd, vd2) => (vd2._1, vd._2))
val pushedGraph = Graph(evUpdatedGraph.vertices, augmentedERDD)
```

図 1　A 実装の Push

ソッドである．これにより Push 処理で必要な情報が pushMsg へと格納されアクティブな頂点毎に集約された VertexRDD となる．Phase2 では Phase1 で集約した情報を基に各頂点毎に map 処理により Push する δ や Push 後の $e(v), r_f(v,w)$ の値の計算をし，それを pushedData へと格納する．最後に Phase3 では Push 後の計算結果が格納されている pushedData からデータを取り出して元のグラフへと適用していく．この際用いている joinVertices メソッドは VertexRDD を引数に取り，その VertexRDD とグラフにある頂点のデータに基いて新たな頂点データを計算するものであり，このメソッドを用いることで頂点に

```
//Phase1
val relabelMsg = pushedSubgraph.aggregateMessages[Int](
   //sendMsg
   edgeCtx => {
      if (isActive(edgeCtx.srcId, edgeCtx.srcAttr))
         edgeCtx.sendToSrc(if (edgeCtx.attr._1 > 0) edgeCtx.dstAttr._2 + 1 else INF_LABEL)
      if (isActive(edgeCtx.dstId, edgeCtx.dstAttr))
         edgeCtx.sendToDst(if (edgeCtx.attr._2 > 0) edgeCtx.srcAttr._2 + 1 else INF_LABEL)
   },
   //mergeMsg
   (msg1, msg2) => if (msg1 < msg2) msg1 else msg2
)
//Phase2
val relabeledSubgraph = pushedSubgraph.joinVertices(relabelMsg)
   ((vid, vd, vd2) => if(vd._2 != vd2) (vd._1, vd2) else (vd._1, vd._2))
```

図 2　A の実装の Relabel

```
//Phase1
val deltaData = resultGraphWithDelta.aggregateMessages[Int](
   //sendMsg
   triplet => {
      if      (triplet.attr._3 > 0) triplet.sendToDst(triplet.attr._3) // forward
      else if (triplet.attr._3 < 0) triplet.sendToSrc(-triplet.attr._3) // backward
   },
   //mergeMsg
   (msg1, msg2) => (msg1 + msg2)
)
//Phase2
val msgAppliedVD = resultGraphWithDelta.joinVertices(deltaData)((vid, vd, vd2) => ((vd._1 + vd2),
     vd._2))
val newResultGraph = msgAppliedVD.mapEdges({ed =>
   if      (ed.attr._3 < 0) (ed.attr._1 - ed.attr._3, ed.attr._2)
   else if (ed.attr._3 > 0) (ed.attr._1,          ed.attr._2 + ed.attr._3)
   else                     (ed.attr._1,          ed.attr._2)
})
resultGraph = Graph(newResultGraph.vertices, resultGraph.subgraph(epred = edge =>
     !isActive(edge)).edges.union(newResultGraph.edges))
```

図 3　A 実装のメッセージ処理

処理結果を適用する．以上により Push 処理が実現される．ただし，この段階では Push のメッセージ δ は伝播する辺 (v, w) 上に記録されており，その先にある頂点 w にはまだ適用されていない点に注意してもらいたい．δ の適用処理については後述するメッセージの適用処理の段階で行う．

次に Relabel 処理を図 2 に示す．Phase1 では Push の際にも用いた aggregateMessages メソッドを用いて周囲の Relabel の条件を満たす頂点から $d(w)$ を集約して relabelMsg を生成している．Phase2 ではその relabelMsg を基に $d(v)$ の更新を行っている．以上により Relabel 処理を GraphX 実装している．

最後に δ の適用処理を図 3 に示す．Phase1 では aggregateMessages メソッドを用いて各辺に記録されているメッセージ δ をその送信先の頂点 v へと送信して deltaData へと格納している．Phase2 ではこの

```
object Pregel{
  var graph:Graph[VertexData, EdgeData] = Graph(vertexRDD, edgeRDD)
  while(isContinue){
    //Phase1
    val msg = graph.subgraph(isActive).aggregateMessages[MsgData](
      val sendMsg: EdgeContext[VertexData, EdgeData, MsgData] => Unit,
      val meargeMsg: (MsgData, MsgData) => MsgData
    )
    //Phase2
    var graph = graph.joinVertices(msg)(
      val applyMsg : (VertexId, VertexData, MsgData) ⇒ VertexData)
    )
  }
}
```

図 4　GraphX による Pregel モデルの表現

表 2　実装 B のおけるグラフの型情報

	型	格納される情報
VertexData	(Int, Int, EdgesData)	$(e(v), d(v), (v, w))$
EdgesData	Map[VertexId, (Int, Int, Int)]	Map$[w, (r_f(v,w), r_f(w,v), \delta)]$

deltaData を基に送信されてきた δ を宛先の $e(v)$ へと加算し，逆辺のフローを変動させている．

以上の 3 つのステップを繰り返すことで Push-Relabel アルゴリズムを実装している．

4.3　B 実装

B 実装では Push-Relabel アルゴリズムを実装するにあたり，Pregel モデルによるグラフ計算を基に実装方針を立て，その上で Push-Relabel アルゴリズムを実装した．A 実装に関しては Push-Relabel アルゴリズムを直接 GraphX による実装を素直に行ったのに対し，こちらでは Pregel のモデルに基づいた既存実装 [12] を，GraphX の API に対応付ける形で翻訳したものである．

Pregel とは Google により発表された分散グラフ処理モデルであり，頂点主体プログラミングモデルによるグラフ処理を採用している．頂点主体プログラミングモデルとはグラフ計算において各頂点を処理の処理単位としてみなし，グラフ全体の各頂点がそれぞれ並列に計算を行うモデルである．この Pregel 処理を GraphX での実装を簡易的に記述したものを図 4 に示す．Pregel ではメッセージ送信関数（sendMsg）とメッセージマージ関数（meargeMsg），メッセージ適用関数（applyMsg）の 3 つの関数を設定し，この関数に基いて Phase1 の aggregateMessages によるメッセージ送信と Phase2 の joinVertices によるメッセージの適用処理を指定された条件の間繰り返すというものである．GraphX ではこの Pregel モデルに基づいた計算をするための PregelAPI が提供されており，この PregelAPI はメッセージ送信関数やメッセージマージ関数，頂点データ更新関数，各種設定パラメータを与えてやると自動的に Pregel モデルに沿ったグラフ処理を実現してくれるというものであり，内部実装としては概ね同じ処理を行っている．

B 実装では前述の通り Pregel モデルによるグラフ計算を実装方針として実装を行う．まず B 実装でのデータ型定義を表 2 に示す．Pregel モデルでは頂点主体プログラミングモデルという処理形態取っており，それを GraphX 上で実装するにあたり A 実装とは異なったデータ型定義となっている．具体的には A 実装では辺に関するデータは EdgeRDD に，頂点に関するデータは VertexRDD に格納していたのに対して B 実

```
val pushedSubgraph = activeSubgraph.mapVertices(push)

val push: (VertexId, VertexData) => VertexData = {case (id, attr@(ev,dv,eds)) =>
   if (isActive(id,attr)) {
      var newEv = ev
      var newED: Map[VertexId, (Int, Int, Int, Int)] = Map()
      for ((wid, (cap,rf,dw,_)) <- attr._3) {
         var delta = 0
         if (dv == dw + 1) {
            delta = if (newEv <= rf) newEv else rf //delta = min(e(v), residual)
            newEv -= delta
         }
         newED += wid -> (cap, rf - delta, dw, delta)
      }
      (newEv, attr._2, newED)
   } else {
      attr
   }
}
```

図 5　B 実装の Push

```
val selfRelabeledSubgraph: Graph[(Int, Int, EdgesData, Boolean), Int] =
     pushedSubgraph.mapVertices(relabel)

resultGraph = resultGraph.joinVertices(selfRelabeledSubgraph.vertices)((vid, vd, vd2) => (vd2._1,
     vd2._2, vd2._3))

val relabel: (VertexId, VertexData) => (Int, Int, EdgesData, Boolean) = {case (id,
     attr@(ev,dv,eds)) =>
   if (isActive(id,attr)) {
      var newdv = INF_LABEL
      for ((_, (_,rf,dw,_)) <- eds) {
         if (rf > 0 && dw + 1 < newdv)
            newdv = dw + 1
      }
      if (newdv == dv) (ev, dv, eds, false) else (ev, newdv, eds, true)
   } else {
      (ev, dv, eds, false)
   }
}
```

図 6　B 実装の Relabel

装では基本的に計算に必要なデータを VertexRDD に全て持たせるために辺情報は EdgesData と呼ばれる Map の中に格納され，それを各出頂点が VertexData の中で持っている．そのため辺情報を持たないグラフとなっている．

B 実装での Push 処理を図 5 に示す．B 実装では頂点 v が Push に必要な各辺の情報を持っているため aggregateMessages による辺情報の集約は必要がなく各頂点に対して push 関数を mapVertices により適用することで Push 処理を行っている．push 関数の処理の内容そのものに関しては A 実装におけるデータ集約後の pushedData の計算と同様である．

```
//Phase1
val concernedERDD = selfRelabeledSubgraph.vertices.flatMap({case (vid, (_, dv, eds, isRelabeld)) =>
   for ((wid,(_,_,_,delta)) <- eds) yield Edge(vid,wid,(delta,dv,isRelabeld))})
val concernedSubgraph = Graph(resultGraph.vertices, concernedERDD)
val neighborLabelMsg = concernedSubgraph.aggregateMessages[Map[VertexId, Int]](
   // sendMessege
   edgeCtx => if (edgeCtx.attr._3) edgeCtx.sendToDst((Map(edgeCtx.srcId -> edgeCtx.attr._2))),
   // mergeMsg
   (_ ++ _)
)

//Phase2
val neighborRelabeledSubgraph = concernedSubgraph.joinVertices(neighborLabelMsg)((vid, vd, vd2) => (
   vd._1,
   vd._2,
   vd._3 ++ (for ((wId, dw) <- vd2) yield wId -> (vd._3(wId)._1, vd._3(wId)._2, dw,
        vd._3(wId)._4))))

//Phase3
var deltaMsg = concernedSubgraph.aggregateMessages[(Int, Map[VertexId, Int])](
   triplet => {
      val delta = triplet.attr._1
      if (delta > 0) triplet.sendToDst((delta, Map(triplet.srcId -> delta)))
   },
   (msg1, msg2) => (msg1._1 + msg2._1, msg1._2 ++ msg2._2))

var deltaApply = neighborRelabeledSubgraph.joinVertices(deltaMsg)((vid, vd, vd2) => (
   vd._1 + vd2._1,
   vd._2,
   vd._3 ++ (for ((wId, delta) <- vd2._2) yield wId -> (vd._3(wId)._1, vd._3(wId)._2 + delta,
        vd._3(wId)._3, vd._3(wId)._4))))
//clean delta
var deltaAppliedGraph = deltaApply.mapVertices({case (id, (ev,dv,eds)) =>
   (ev, dv, for ((wid, ed) <- eds) yield wid -> (ed._1, ed._2, ed._3, 0))
})
resultGraph = resultGraph.joinVertices(deltaAppliedGraph.vertices)((vid, vd, vd2) => (vd2._1,
     vd2._2, vd2._3))
```

図 7　B 実装のメッセージ処理

B 実装での Push 処理を図 6 に示す．B 実装における Relabel では辺 (v, w) の持つ情報の中に $d(w)$ が含まれているため隣接頂点から情報のの集約をせずとも Relabel に必要な情報は各頂点が保持している．したがって Push 同様にこちらも relabel 関数を mapVertices により各頂点に適用することで Relabel 処理を行う．relabel 関数の処理内容は実装 A で行っている処理と概ね同じであり，Relabel の条件を満たす隣接頂点 w の $d(w)$ の集合の中で最も小さいラベルに 1 加えたものを新たな $d(v)$ として適用するという内容である．

最後に Push と Relabel の適用後のメッセージ送信とその適用の処理を図 7 に示す．Phase1 では Push/Relabel 処理の適用済みの部分グラフ selfRelabeledSubGraph から δ と $d(v)$，そして v に関して Relabel が行われたかのフラグを持った EdgeRDD を生成した後，それを辺集合とした concernedSubgraph を用いて aggreageteMessages により更新されたラベルの情報を neighborLabelMsg として隣接頂点へと送信している．Phase2 ではこのメッセージを各頂点の

表 3　実装 C におけるグラフの型情報

	型	格納される情報
VertexData	(Int, Int, VertexPushMap)	$(e(v), d(v), \delta(v, w))$
VetexPushMap	Map[(VertexId, Boolean), Int]	Map[$(w, pushDirection), \delta$]
EdgeData	(Int, Int)	$(f(v, w), c(v, w))$

持つ辺情報の $d(w)$ として適用する．これにより更新されたラベルが隣接頂点へと伝達される．Phase3 では Phase1 で生成した concernedSubgraph を用いて aggreageteMessage により delta の情報を送信し，joinVertices により $e(w)$ へと適用されていく．このようにこの実装 B ではメッセージ送信をする段階に一時的にメッセージを保持した辺集合を必要に応じて生成してメッセージの送受信を実装している．

4.4　C 実装

比較対象の関連研究として既存の GraphX による Push-Relabel 実装をここで紹介する．この実装では表 3 のようなデータ型定義となっている．これは A 実装のデータ型定義と本質的には同じものである．

ここでは C 実装について Push-Relabel の処理を幾つかの段階に分けて実装を説明する．プログラムの詳細は付録に掲載する．まず Push 及び Relabel は 2 つの段階で処理を行う．Phase1 では Push 及び Relabel で必要な情報を各頂点へと送信している．A 実装や B 実装では共に Push と Relabel の 2 つの処理について別々にメッセージ送信（それに該当する処理）とその結果の適用を行っているがこの実装では一度のメッセージ送信（aggregateMessages）により Push と Relabel の情報を同時に送っているという点が大きな違いである．続く Phase2 ではメッセージ情報を使い各頂点で Push 及び Relabel の適用を行っている．この適用処理も同一の outerJoinVertices により Push 及び Relabel 結果の適用を行っている．ある時点の頂点 v に関して Push と Relabel の 2 つの処理は高々どちらか一方しか適用条件が満たされないという点を利用した実装であり，これにより 1 度の処理でアクティブな各頂点に対して Push か Relabel の適用可能な処理を同時に適用することができてる．

続いてメッセージの適用処理を 3 段階に分けて示す．Phase1 では各頂点に格納されている Push/Relabe 操作の情報を基に辺 (v, w) のフローの情報を更新している．Phase2 では aggregateMessages で各頂点にある Push 結果の情報からメッセージを送信している．Phase3 ではそのメッセージを outerJoinVertices で各宛先で適用している．こちらのメッセージ反映の際にも外部に余計な中間データを生成せずに outerJoinVertices 内でメッセージと頂点データの処理の中で処理を全て行っている．つまり全体として 1 回のグラフ処理内容は計算量が増え多少複雑になっているが余計な中間データやメッセージ送信処理を削減し，全体としてコンパクトなグラフ処理として実装している．

4.5　実装上での問題などについて

本研究において GraphX による Push-Relabel の実装をする際に感じた問題点などについて述べていく．

1 つ目に処理結果適用対象のグラフの辺に対してグラフ外部から EdgeRDD を入力として辺データに適用・演算するために効率的な API が用意されていないことである．外部の VertexRDD を引数としてグラフ上の頂点に同様の処理を行う場合は joinVertices メソッドなどを用いることで実現が可能であり，VertexID 単位での map 演算を容易に処理が記述可能である．しかし，辺データの場合はその処理に該当する API が存在しておらず，予め外部で適用結果となる EdgeRDD を生成してから union を行うという方法や union をした後に gropupEdges で多重辺の集約処理で実現するなど冗長な実装しか行えず効率が悪い．

2 つ目はグラフ処理の適用範囲が限定しづらい点である．GraphX では各種 map 処理や aggregateMessages などの各種処理においてその処理の適用範囲が限定しづらいという点である．GraphX では基本的に全ての頂点や辺，トリプレットに対して処理を適用

	頂点数	辺数
Graph1	6	8
Graph2	100	261

表 4　入力グラフ

	Graph1	Graph2
A 実装	19.90 sec	–
B 実装	3.59 sec	–
C 実装	4.86 sec	241.85 sec

表 5　実験結果

する必要があり，実際に処理を必要とする対象が少ない場合は効率が悪い．subgraph メソッドなどを用いることで部分グラフを取り出して処理を適用する範囲を限定することは可能であるが，その範囲が変動することが往々にしてあるグラフ処理では結局元のグラフから再度取得する必要があるため適用範囲が小さくなり続ける場合以外は改善することはできないと考えられる．

5　実験

本節では，本研究で GraphX 上で実装した Push-Relabel プログラムについての実験とその結果について説明していく．

5.1　実験概要

本研究で前節で説明した実装した 2 つの Push-Relabel プログラムと既存実装の Push-Relabel プログラムに対して入力グラフを与えてそのグラフについて最大フローの計算を行った．実験に使用したグラフの情報を表 4 に示す．

実験環境は Intel Core i5 2500(3.30GHz 2 core 4 threads)，8GB RAM の PC9 台で構成される Spark クラスタである．実験に使用した Spark はバージョン 2.0.2 用いた．

5.2　実験結果

実験の結果を表 5 に示す．

小規模なグラフ Graph1 では実装 A が実装 B,C と比較して非常に遅いという結果となった．大規模なグラフ Graph2 の場合は実装 C が正しく処理を終えられたのに対し，実装 A,B に関しては処理途中で DAGScheduler 関連のエラーやスタックオーバーフローエラーが発生して処理が最後まで完遂できないという結果となってしまった．この場合の入力グラフ Graph2 に対する Push-Relabel 処理は適切に処理ができれば 1000 サイクル程で処理を終えられるが実装 A では 10 サイクル強，実装 B では 150 サイクル前後の段階でそれぞれ処理が中断されてしまっておりほとんど処理が行えていないという結果となった．また，実装 B,C に関しては処理の繰り返し回数にかかわらずある程度一定の処理時間で 1 サイクル分のグラフ処理が行えていたのに対し，A 実装では繰り返しの回数が増えるに連れて 1 サイクルの処理時間が長くなるという現象が見られた．

6　考察

本節では本研究から得られた知見を基に考察をする．

6.1　Spark GraphX 実装のパフォーマンスについて

まず今回実装した 2 つの GraphX プログラムについて考えていく．実装 A では GraphX の API をシンプルに使用した実装となっていたが，そうした場合他の実装と比べて中間データが多く生成する事となり処理速度が他の 2 つと比べて大幅に遅くなったと考えられる．実装 B では処理速度そのものは実装 C と比較して少し早いという成果を出ており処理の実装に関しては効率的なものだったと考えられる．しかし，この B 実装に関しても A 実装と同じく規模の大きいグラフに対して処理を行う場合に適切に処理できないという致命的な問題が発生してしまった．今回新たに実装した 2 つに関して起きたこの問題は実装したプログラムの処理過程で生じる中間データやグラフ処理の段階を幾つかに切り分けた結果，グラフ処理を示す DAG の構造が複雑になり，Spark が想定しているよりも早い時間で大きな DAG が形成され

てしまったことが原因だと考えられる．そのためこのDAG がを処理系がチェックポインティングなどの処理で DAG を走査した時にその構造が深すぎてスタックオーバーフローエラーなどを引き起こしてしまったのだと考えた．比較対象とした実装 C では 1 回のグラフ処理の計算量は他の実装よりも多く，複雑な内容となっているがグラフ処理そのものの回数は非常に少ない．また，複数の中間データを生成せず 1 つのグラフと 1 つの中間データのみを更新と適用していることから分岐の起きない 1 本の DAG のとして表現されている．これらが要因となり他と比べて安定した動作とパフォーマンスを実現できてるのだと考えられる．

次に GraphX によるグラフ処理のパフォーマンスについて考える．今回実装した中でも比較的処理の早い実装 B,C では入力グラフ Graph2 に対して処理を行った場合 1 サイクルの処理が大体数百ミリ秒で実行されていた．この処理速度との比較として Pregel+ [10] による Push-Relabel アルゴリズム実装 [12] を行った例のパフォーマンスを見る．Pregel+を用いて同等の環境で Pregel+による Push-Relabel 処理を行った結果は GraphX 実装における 1 サイクルにかかる時間以下で処理全体を終えることができている．つまり，GraphX による Push-Relabel アルゴリズムの実装は非常に遅いと考えられる．

6.2 Spark GraphX の問題点

GraphX の問題点として次の 3 点を考えた．

1 点目の問題点として考えたのは Spark の処理形態を起因とする無駄の多い処理である．Spark では RDD に対する操作により処理を記述するという性質上基本的に RDD 全体に対して処理が適用される．しかし本研究で実装したような Push-Relabel アルゴリズムのようなグラフ処理では実際に処理の対象となる要素はコレクション全体の中の少数に限られる．そのため並列処理とはいえ毎回データ全体への処理が施行され，多量の本質的には必要ない走査が処理性能の低下を招く可能性が考えられる．また，この状態を回避するために処理対象の範囲を制限しつつその範囲を動的に変動させることは，変動する範囲がある程度自明であるような処理でも難しいのが現状である．

2 点目の問題点として考えたのはグラフ処理の記述をする上で必要だと感じる API が用意されていないことである．この点に関しては前節でも述べたが，頂点に関しては join メソッドが用意されているが辺に関する join メソッドが用意されていない．そのため辺に関する処理とその適用をする際に外部データを利用しようとすると本質的に行いたい辺の処理から考えると冗長な処理にならざるを得ないという問題がある．

3 点目の問題点として考えたのはグラフ処理の記述をする上で処理過程におけるある程度の意味を持つ単位で中間データを持ったり処理を分割した場合に著しく処理速度が低下するという点である．プログラミングをする上ではメンテナンス性などの観点からある程度の可読性を保たせるべきである．今回の実装では中間データを必要だと考えられる範囲で定義し，不必要に中間データを生成しないようにすることでプログラムの可読性を保ちつつかつ効率的に実装を行った．しかし，今回の実装では不必要に中間データを多量に生成したり，明らかに非効率的な処理を行っているわけでもないにも関わらず DAG の増大を起因とすると思われるエラーが発生してしまう．

6.3 改善案と今後の展望

前述の 3 つの問題のうち，最初の 2 点は，GraphX が採用するプログラミングモデルに起因するものである．GraphX API は，GAS [2] 及び辺主体モデル [9] に基づいる．これらのモデルでは，モデルの上で辺を明示的に選択する術を提供されず，辺走査はシステムで定義されている．これを素直に Spark の共有メモリモデルに移植したことで，グラフ中の全ての辺を調べる非効率的なプログラミングを強制する API になっている．Push-Relabel アルゴリズムのように辺選択を柔軟かつ効率良く行うことが求められる処理では，それらとは異なるモデルに基づく API の提供が望まれる．実際，本研究では，GraphX の API では提供されていない頂点から辺を選択する操作を，Pregel モデルを翻訳する形で実装した（B 実装）．このことから，既存のグラフ処理モデルを参考しつつ，現状の GraphX では十分整備されていない

操作を，Spark 上で効率良い形で追加することは十分可能であると考える．

この GraphX の API 上の制限による非効率性は，典型的なグラフ解析問題を解くときですら影響しうるものであり，GraphX が既存のグラフ処理システムの中で際立って遅いこと [4] と，深く関係すると考えている．GraphX の API の不足を補うことには，Push-Relabel アルゴリズムだけでなく幅広いアプリケーションにおいて大きな価値を見込める．

前述の 3 つの問題のうち，最後の 1 点は，Apache Spark のランタイムモデルに起因するものである．RDD という名前の定義上，fault resilience が Spark にとって必要不可欠であることは読み取れる．しかし，そのための Spark の機構が，我々の A 実装や B 実装では過剰であり，結果として実行を妨げる要因となった．かとって，このランタイムの働きを抑制するように，C 実装のようなプログラミングを行うとプログラムの可読性や保守性が低下する上に，プログラミングの柔軟性を売りとする Spark の設計思想にも反する．したがって，Push-Relabel アルゴリズムのように，複数の軽い操作を合成した反復計算については，fault resilience の観点で特別な対処が求められる．実際，軽い計算であればその途中結果について fault resilience は必要なく，正しい元データから再計算すれば実用的には十分であることは大いに考えられる．このように，提供する fault resilience の指標を緩和した実行モードを開発すれば，複数の軽い操作を合成した反復計算への対処が可能であると考えられる．

参 考 文 献

[1] Goldberg, A. V. and Tarjan, R. E.: A New Approach to the Maximum-Flow Problem, *J. ACM*, Vol. 35, No. 4(1988), pp. 921–940.

[2] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C.: PowerGraph: Distributed Graph-parallel Computation on Natural Graphs, *Proc. 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, USENIX, 2012, pp. 17–30.

[3] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework, *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, USENIX, 2014, pp. 599–613.

[4] Iosup, A., Hegeman, T., Ngai, W. L., Heldens, S., Prat-Pérez, A., Manhardt, T., Chafi, H., Capotă, M., Sundaram, N., Anderson, M., Tănase, I. G., Xia, Y., Nai, L., and Boncz, P.: LDBC Graphalytics: A Benchmark for Large Scale Graph Analysis on Parallel and Distributed Platforms Lifeng Na Peter Boncz, *PVLDB*, Vol. 9, No. 13(2016), pp. 1317–1328.

[5] Kalavri, V., Vlassov, V., and Haridi, S.: High-Level Programming Abstractions for Distributed Graph Processing, 2016.

[6] Langewisch, R. P.: A performance study of an implementation of the push-relabel maximum flow algorithm in Apache Spark's GraphX, Master's thesis, Colorado School of Mines, 2015.

[7] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing, *Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, ACM, 2010, pp. 135–146.

[8] McCune, R. R., Weninger, T., and Madey, G.: Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, *ACM Comput. Surv.*, Vol. 48, No. 2(2015), pp. 25:1–25:39.

[9] Roy, A., Mihailovic, I., and Zwaenepoel, W.: X-Stream: Edge-centric Graph Processing using Streaming Partitions, *Proc. ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP '13)*, ACM, 2013, pp. 472–488.

[10] Yan, D., Cheng, J., Lu, Y., and Ng, W.: Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation, *Proc. 24th International Conference on World Wide Web (WWW '15)*, ACM, 2015, pp. 1307–1317.

[11] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I.: Apache Spark: a unified engine for big data processing, *Commun. ACM*, Vol. 59, No. 11(2016), pp. 56–65.

[12] 佐藤重幸: Push-Relabel アルゴリズムの頂点主体プログラミング, 第 *19* 回プログラミングおよびプログラミング言語ワークショップ *(PPL2017)* 予稿集, 2017.

A　C 実装の GraphX 実装

```
//Phase1
val eligiblePushesRDD = graph.aggregateMessages[SurveyMessage] (
   edgeContext => {//sendMsg
      if (v != {s,t} && residual(v,w) > 0 && e(v) > 0) {//forward
         if (d(v) == d(w) + 1)) {//Send PushData
               val pushAmount = math.min(edgeContext.attr._2 - edgeContext.attr._1,
                    edgeContext.srcAttr._1)
               edgeContext.sendToSrc((Map((edgeContext.dstId, true) -> pushAmount),
                    edgeContext.dstAttr._2))
         } else {//Send RelabelData
            edgeContext.sendToSrc((Map(), edgeContext.dstAttr._2))
         }
      }
      if (w != {s,t} && residual(w,v) > 0 && e(w) > 0) {//backward
         if (d(w) == d(v) + 1) {
            val pushAmount = math.min(edgeContext.attr._1, edgeContext.dstAttr._1)
            edgeContext.sendToDst((Map((edgeContext.srcId, false) -> pushAmount),
                edgeContext.srcAttr._2))
         } else {
            edgeContext.sendToDst((Map(), edgeContext.srcAttr._2))
         }
      }
   },
   (a, b) => {// mergeMsg
      (a._1 ++ b._1, math.min(a._2, b._2))
   }
)
//Phase2
graph = graph.outerJoinVertices(eligiblePushesRDD) {
   (id: VertexId, data: VertexData, msg: Option[SurveyMessage]) => {
      if (msg.isEmpty) {
         (data._1, data._2, Map[(VertexId, Boolean), Int]())
      } else if (msg.get._2 >= data._2) {//Relabl
         (data._1, msg.get._2 + 1, Map[(VertexId, Boolean), Int]())
      } else {//Push
         var excess = data._1
         val selectedPushes = scala.collection.mutable.Map[(VertexId, Boolean), Int]()
         breakable {
            msg.get._1.foreach(pushData => {
               val dstId = pushData._1._1
               val forwardPush = pushData._1._2
               val pushAmount = pushData._2
               if (excess > 0) {
                  val selectedPushAmount = math.min(pushAmount, excess)
                  excess -= selectedPushAmount
                  selectedPushes((dstId, forwardPush)) = selectedPushAmount
               } else {
                  break
               }
            })
         }
         (excess, data._2, selectedPushes.toMap)
      }
   }
}
```

図 8　C 実装の Push/Relabel 処理

```
//Phase1
graph = graph.mapTriplets[EdgeData](
   (edgeTriplet: EdgeTriplet[VertexData, EdgeData]) => {
      if (edgeTriplet.srcAttr._3.contains((edgeTriplet.dstId, true))) {//Push forward
         val pushAmount: Int = edgeTriplet.srcAttr._3((edgeTriplet.dstId, true))
         (edgeTriplet.attr._1 + pushAmount, edgeTriplet.attr._2)
      } else if (edgeTriplet.dstAttr._3.contains((edgeTriplet.srcId, false))) {//Push backward
         val pushAmount: Int = edgeTriplet.dstAttr._3((edgeTriplet.srcId, false))
         (edgeTriplet.attr._1 - pushAmount, edgeTriplet.attr._2)
      } else {//Push なし
         edgeTriplet.attr
      }
   }
)
//Phase2
val executedPushesRDD = graph.aggregateMessages[Int] (
   edgeContext => {
      if (edgeContext.srcAttr._3.contains((edgeContext.dstId, true))) {//Push forward
         val pushAmount: Int = edgeContext.srcAttr._3((edgeContext.dstId, true))
         edgeContext.sendToDst(pushAmount)
      }
      if (edgeContext.dstAttr._3.contains((edgeContext.srcId, false))) {//Push backward
         val pushAmount: Int = edgeContext.dstAttr._3((edgeContext.srcId, false))
         edgeContext.sendToSrc(pushAmount)
      }
   },
   (a, b) => {
      a + b
   }
)
//Phase3
graph = graph.outerJoinVertices(executedPushesRDD) {
   (id: VertexId, data: VertexData, msg: Option[Int]) => {
      (data._1 + msg.getOrElse(0), data._2, data._3)
   }
}
```

図 9　C 実装のメッセージ処理