高知工科大学
KOCHI UNIVERSITY OF TECHNOLOGY

Spark GraphX におけるメッセージ集約の性能解析

著者	松本 拓也
発行年	2018-03
その他のタイトル	Performance Analysis of Maggage Aggregation in Spark GraphX
URL	http://hdl.handle.net/10173/1913

平成 29 年度

修士学位論文

高知工科大学

情報システム工学科

Spark GraphX におけるメッセージ集約の性能解析

Performance Analysis of Maggage Aggregation in Spark GraphX

1205085　松本 拓也

指導教員　松崎 公紀

2018 年 2 月 2 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報システム工学コース

要 旨

高知工科大学

情報システム工学科

Spark GraphX におけるメッセージ集約の性能解析

松本 拓也

Apache Spark は大規模データ処理において汎用性と高速性を目指して設計された並列分散処理フレームワークである．Apache Spark は Scala を用いた実装や様々なコンポーネントを持っていることから他のフレームワークと比べて高い汎用性を実現している．その Apache Spark が持つコンポーネントの 1 つとして GraphX が挙げられる．

Spark GraphX はグラフ処理を並列処理をすることを目的としたコンポーネントである．この GraphX を用いることで Apache Spark による分散環境で並列グラフ処理が実装されている．しかし GraphX のグラフ処理パフォーマンスが他のフレームワークと比較して非常に遅いという報告が近年上がっており，大きな問題となっている．

本研究ではこの GraphX が持つグラフ処理パフォーマンスの低さの原因について Maximum-Flow や SSSP などのグラフアルゴリズムを用いたケーススタディを通して調査を行った．その結果 GraphX のパフォーマンスの低さの一因として GraphX の持つグラフ処理 API の中でもメッセージ集約処理を行うためのものの動作に問題があるのではないかということが分かった．この処理を司る API の性能を調べるためのベンチマーク及びデバッグを行った所，メッセージ集約処理時の辺走査や頂点データを管理しているクラスのデータ更新がグラフデータの規模に応じて増大しておりパフォーマンス問題の大きな一因ということが明らかとなった．

キーワード　　並列処理，グラフ理論, Apache Spark, GraphX

// Abstract

Performance Analysis of Maggage Aggregation in Spark GraphX

Takuya Matsumoto

Apache Spark is a parallel distributed processing framework designed for versatility and high speed in large-scale data processing.Apache Spark has high versatility by various components. One of the components that Apache Spark has is GraphX.

Spark GraphX is a component aimed at parallel processing graph processing. By using this GraphX, we can implement parallel graph processing in a distributed environment by Apache Spark easily. However, a report said that GraphX was much slower than other graph processing framework.It is a big broblem.

In this study, we researched tthe cause of this low graph processing performance of GraphX by some case study and benchmark. As a result, it turned out that the possibility that problems are caused by messages aggregation processing of GraphX. So we did benchmark and debug there, we found the probrem that edge scanning and updating VertexView are major problems in aggregateMessages.

key words Parallel processing, Graph theory, Apache Spark, GraphX

目次

第 1 章　はじめに　1

第 2 章　グラフ処理　3

2.1　グラフとグラフ処理　3

2.2　グラフ処理フレームワーク　3

第 3 章　Spark GraphX　5

3.1　Apache Spark　5

3.2　GraphX　6

3.3　GraphX API　8

3.4　GraphX のパフォーマンスに関する問題　9

第 4 章　グラフ処理プログラムの GraphX 実装によるケーススタディ　10

4.1　GraphX ケーススタディ　10

4.2　ケーススタディ1.Push-Relabel　11

4.2.1　Maximum-Flow 問題　11

4.2.2　Push-Relabel アルゴリズム　12

4.2.3　GraphX 上での実装　13

4.2.4　パフォーマンス評価実験　15

4.2.5　実験結果　15

4.2.6　結果からの考察　16

4.3　ケーススタディ2.SSSP　19

4.3.1　SSSP 問題　19

4.3.2　SSSP の解法アルゴリズム　19

4.3.3　GraphX 上での実装　20

4.3.4 パフォーマンス評価実験 21
4.3.5 実験結果 21
4.3.6 結果からの考察 21
4.4 GraphX によるグラフ処理プログラムの実装に関する問題 22
4.5 ケーススタディからの考察 23

第 5 章 Spark GraphX におけるメッセージ集約処理のベンチマーク実験 25
5.1 グラフ処理とメッセージ集約処理 25
5.2 GraphX におけるメッセージ集約処理 API
aggregateMessages について 26
5.3 aggregateMessages と
aggregateMessagesWithActiveSet の比較実験 27
5.3.1 実験内容 27
5.3.2 実験結果と考察 28
5.4 メッセージ集約処理のベンチマーク実験 29
5.4.1 実験内容 29
5.4.2 実験結果と考察 30

第 6 章 考察と提言 33

第 7 章 まとめ 36

参考文献 38

図目次

5.1 頂点数を変化させた場合 30
5.2 辺数を変化させた場合 31

表目次

4.1 入力グラフ 16
4.2 実験結果 16
4.3 実験結果 21

5.1 実験結果 28

第 1 章

はじめに

近年のビッグデータ処理の需要の高まりに伴い，その活用される場面や分野は非常に幅広いモノとなっている．情報機器の普及や多種多様な Web サービスの展開，IoT 技術などにより様々な情報がインターネットなどのネットワークを通じて発信・収集され，その情報の量は膨大で種類は多岐にわたる．そんな様々なビッグデータの中の代表の 1 つとしてグラフデータが挙げられる．

グラフデータとは，あるデータを “頂点”，そしてそれらの間を何らかの情報や関係で繋ぐ “辺”，この 2 つの要素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）などにおけるユーザ間での関係性をグラフとして表すソーシャルグラフや輸送経路と距離などの運搬コストなどを表現した輸送ネットワークグラフ，Web サイトのリンクのつながり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．このグラフ処理を応用することでグラフデータから何らかの有益な情報を得ることができ，ビッグデータ解析の対象として注目されている．

このようにビッグデータに対する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おり単一の計算機で計算することは困難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しており [7]，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．Spark GraphX[2] はその並列グラフ処理フレームワークの 1 つである．

Spark GraphX は並列分散処理フレームワーク Apache Spark[9] のグラフ処理向けコンポーネントとして開発されたものである．Apache Spark は汎用性の高い並列分散処理フレームワークとして開発され，様々なコンポーネントを実装しており GraphX はその中の 1 つである．汎用性と高速な処理を目指した Apache Spark であるが，グラフ処理について注目した時に他のグラフ処理フレームワークと比較して非常に遅いという報告があげられている [3]．

そこで本研究では Spark GraphX におけるグラフ処理性能，及び性能が低い要因について調査を行った．

第 2 章

グラフ処理

本章では研究対象である GraphX の対象アプリケーションであるグラフ計算とそれに付随する各要素技術について説明する．

2.1　グラフとグラフ処理

グラフとは何らかのデータを “頂点（Vertex）” と頂点間を何らかの関係やパラメータで繋ぐ “辺（Edge）”，この 2 つの要素の集合を用いて表現するデータ構造である．

グラフとして表現されるデータの例としては実世界や SNS（Social Networking Service）などにおけるソーシャルグラフや輸送経路と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，電子回路などが挙げられる．

このグラフに対し，頂点や辺の持つパラーメタに関する演算やグラフ構造の変形・分割・統合など様々な処理を適用し，最終的に目的となる新たなグラフや値を算出する処理をグラフ処理と呼ぶ．

グラフ処理を用いて解くグラフ問題は実社会での応用では何らかのデータや問題をグラフ・グラフ問題としてモデル化し，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様々な問題を解くことが可能となる．

2.2　グラフ処理フレームワーク

グラフ処理を計算機で行う場合，汎用のプログラミング言語を用いて処理を行うことはできるが多くの場合グラフ処理向けのライブラリやフレームワークを用いる．近年ではビッグ

2.2　グラフ処理フレームワーク

データ処理の需要などに伴い，大規模なグラフデータに対して高速に処理を行うことが求められている．その為現在ではほとんどのグラフ処理フレームワークで並列処理・分散処理の実装がされている．代表的なフレームワークとしては Spark GraphX[2] や Pregel[8]+, Giraph[11] などが挙げられる．

第 3 章

Spark GraphX

本章では本研究での研究対象である Apache Spark 及びその中のグラフ処理コンポーネント Spark GraphX について述べる．

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 [9]．2009 年に 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・保守が行われている．Apache Spark の大きな特徴として汎用性の高さ，オンメモリでの分散処理，RDD の 3 つが挙げられる．それぞれについて順に説明していく．

1 つ目の特徴である汎用性の高さは Scala による実装と Spark が持つ様々なコンポーネントに起因している．Apache Spark では主に Scala，Java を中心に API が提供されており，Spark 自体の実装も Scala で行われている．その為ユーザが実装する上での記述性が高く柔軟なプログラミングが可能となっている．また Spark では Spark Core と呼ばれる基本的な演算の分散処理の実装に加えて，SQL 処理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の実装が標準で行えるようになっている．加えてこれらのコンポーネントは全て後述する RDD と呼ばれるコレクションを用いていることから，それぞれのコンポーネントやアプリケーション間でその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比較的容易に実現することが可能である．これらにより Apache 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．

Apache Spark における 2 つ目の特徴はオンメモリでの分散処理に特化しているという点である．Spark では内部処理において処理データをできるだけメモリ上に保持しながら処理を実行するように設計されている．そのため従来並列分散処理でよく用いられていた Hadoop などによる MapReduce のような処理手法と比べてストレージとの I/O によるオーバーヘッドをできるだけ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，インタラクティブな対話型の処理，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．

3 つ目の特徴である RDD は Spark 上で処理データを扱うためのデータコレクションである．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．Spark では全ての演算処理を RDD に対して行う変形（Transformation）とアクション（Action）の 2 種類の集合演算として表現しており，処理の一連の流れを DAG（Directed Acyclic Graph）で表現することで処理フローを管理している．Spark ではこの RDD と DAG による処理フローのマネジメント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．

3.2　GraphX

GraphX は Apache 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 [2].

前述した通り，全ての Spark プログラムでは処理対象のデータを RDD を用いて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ではグラフの頂点データの集合を VertexRDD と呼ばれる RDD の拡張コレクションに格納し，辺データの集合を EdgeRDD と呼ばれる

RDD の拡張コレクションに格納する．VertexRDD では頂点に関する情報を保持しており，各頂点ごとに一意に割り当てられた頂点の ID である VertexID とその頂点が持つパラメータ VertexData の 2 つを単位とした集合を RDD として扱っている．EdgeRDD では辺に関する情報を保持しており，辺の Source である頂点の VertexID と Destination である頂点の VertexID, そしてその辺が持つパラメータ EdgeData の 3 つを Edge 型というデータとして管理し，その Edge 型データの集合を RDD として扱っている．これらを 2 つの集合を 1 つの 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．なお GraphX では辺が持つ属性として Source と Destination があるがこれは無向グラフの場合も同様であり，実装側で無向辺として扱うことで無向グラフによる処理を実装する．

Spark GraphX によるグラフ処理プログラムの実装では，GraphX 実装されているグラフ処理向けのデータ処理 API を用いてグラフ処理を実装する方法の他，GraphX で用いられるこの VertexRDD や EdgeRDD は RDD の拡張であるため Spark Core で用いる集合演算をある程度そのまま適用することが可能である．これは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を示す例の 1 つである．GraphX において実装されているグラフ処理 API では基本的な頂点や辺などに対する map 処理からメッセージ集約処理を行う 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．また，GraphX プログラミングではグラフを表現する要素として Triplet と呼ばれるものがる．この Triplet というのは GraphX におけるデータ型の 1 種でで任意の辺に注目したときの Source に当たる頂点と Destination に当たる頂点，そしてそれを繋ぐ辺の 3 つをセットにして扱う単位である．

3.3　GraphX API

ここでは GraphX により提供されているグラフ処理 API の中で基本的かつ重要な API である aggregateMessages, joinVertices, map について簡単に説明する．

1 つ目の aggregateMessages はメッセージ集約処理を行うための関数であり，各 Triplet 毎にメッセージの送信の可否や方向，データの内容などを計算してメッセージの送信を行う関数である．これにより各頂点や辺からそれぞれの持つパラメータなどの情報を各辺の Source や Destination あるいはその両方に送信することができる．また，この aggregateMessages にはメッセージマージの役割もあり，aggregateMessages により送信されるメッセージは最終的なメッセージとなる前にユーザが設定したメッセージマージ関数によりマージされる．ここで生成されるのは各宛先ごとのメッセージを集めた VertexRDD であり，実際にこのメッセージデータを用いて処理を行うのは後述する joinVertices を用いる．

2 つ目の joinVertices は Graph 型のデータに対して別の VertexRDD 型のデータを引数にとり，対応する各頂点のデータを基に新たな頂点のデータを再計算する処理である．この joinVertices を用いることで頂点毎に何らかの入力データを与えてデータを書き換えたり，他のグラフの頂点データとのマージ処理をするなど様々なグラフ処理を実装できる．前述した aggregateMassages のメッセージはメッセージの送信先とメッセージデータが格納された VertexRDD となっているためこの joinVertices を用いることで送信されたメッセージを宛先の頂点に対して何らかの関数を用いて適用することができる．グラフ処理ではメッセージの送信とその適用という処理は頻出するものであるため GraphX によるグラフ処理プログラミングでは aggregateMessages と joinVertices はセットで用いられることが多い．

3 つ目の map は名前の通りグラフ上のデータに対して map 処理を適用する API である．GraphX では mapVertices, mapEdges, mapTriplets の 3 つを提供している．mapVertices, mapEdges については頂点・辺に対してそれぞれの要素を引数にとって単純に map 処理を行うものである．一方 mapTriplets は少し異なっており，map する関数が取る引数は triplet である．その為関数はある辺の Source の頂点と Destination の頂点，そして辺の持つデータを

参照することができ，結果として返すのは辺の値である．そのため mapTriplets を用いることで頂点のデータに基いて辺のデータを書き換える処理が行える．逆に辺のデータを用いて頂点のデータを書き換えるような API は GraphX では提供されておらず，そのような処理を実装する場合は aggregateMessages と joinVertices を組み合わせて実装する必要がある．

以上が GraphX プログラミングにおける代表的なグラフ処理 API である．

3.4　GraphX のパフォーマンスに関する問題

GraphX は Apache Spark による分散環境上でグラフ処理を並列処理することを目的としたコンポーネントである．したがって Spark の持つコンポーネントの 1 つである GraphX は Spark による並列分散処理の恩恵を受けてグラフ処理を行えることから大規模グラフに対するグラフ処理を高速に行えると考えられた．しかし，GraphX とその他に Giraph や PowerGraph，PGX，GraphMat，OpenG などの幾つかのグラフ処理フレームワークに対してベンチマーク調査した結果 GraphX は殆どのグラフアルゴリズムにおいてグラフ処理性能が低く大幅に処理時間がかかるということや処理を終えることができない場合などがあることが明らかとなった [3]．これは高速性や汎用性を謳う Apache Spark のコンポーネントの 1 つとしては大きな問題である．現状このパフォーマンス問題に関してはあまり調査がされておらず GraphX におけるパフォーマンスの知見はあまりない．本研究ではこの GraphX におけるパフォーマンス問題について調査を行った．

第 4 章

グラフ処理プログラムの GraphX 実装によるケーススタディ

本章では GraphX のパフォーマンスの調査を目的として行った各ケーススタディについて説明していく.

4.1 GraphX ケーススタディ

前章で述べたように GraphX によるグラフ処理は他のグラフ処理フレームワークと比較して非常に遅く，何らかのパフォーマンス低下を招く要因があることは明らかである．本研究では GraphX のパフォーマンスに関する問題を調べる上でまず幾つかのグラフ処理プログラムを実装・実験するケーススタディを行った．本研究でケーススタディをするに辺り 2 つの題材を用意した.

1 つ目のケーススタディは Push-Relabel アルゴリズムである．この Push-Relabel アルゴリズムは Maximum-Flow 問題を解くためのアルゴリズムである．ケーススタディとして選んだ理由としては，一般にグラフ処理フレームワークのベンチマークなどで用いられているアルゴリズムは SSSP（Single-Source Shortest Pass）や BFS（Breadth First Search），PageRank などグラフアルゴリズムの中では比較的単純なものがほとんどである [4]．一方で Push-Relabel アルゴリズムで解く Maximum-Flow 問題はフローネットワークを扱う問題であるため比較的複雑な操作をする必要が出てくる．しかしこのような複雑なグラフアルゴリズムに関してはあまり検証がされておらず，知見があまりないのが現状である．その為，よ

り応用的・実用的なアプリケーションの実装に近いグラフアルゴリズムのケーススタディとして Push-Relabel による Maximum-Flow プログラムの実装を選んだ．また，Push-Relabel アルゴリズムは Maximum-Flow 問題を解くアルゴリズムの中でも並列処理の実装がしやすく Pregel モデルに沿った実装が行えることも理由の 1 つである．

2 つ目のケーススタディは SSSP である．こちらは前述したようにグラフベンチマークなどで頻繁に用いられるグラフアルゴリズムであり，Push-Relabel と比べて非常にシンプルなグラフアルゴリズムとなっている．SSSP をケーススタディとして選んだ理由は，Push-Relabel によるケーススタディによって得られた知見から GraphX における問題点と考えられる点がある程度明らかになったため，単純なグラフアルゴリズムを用いてより詳細な GraphX のパフォーマンスについて調査をしつつ Push-Relabel 場合との振る舞いの違いや共通点を調べることが目的である．

以上 2 つが本研究で取り扱ったケーススタディである．それぞれについて以下順に述べていく．

4.2　ケーススタディ1.Push-Relabel

本節では 1 つ目のケーススタディである Maximum-Flow 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．

4.2.1　Maximum-Flow 問題

まず Maximum-Flow 問題について説明をする前にフローネットワークについて説明しておく．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何らかの節点に類するものと節点間を接続する経路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．ここではフローネットワークに関する厳密な定義について

は割愛させて頂く．

Maximum-Flow 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．

4.2.2　Push-Relabel アルゴリズム

本項では Push-Relabel アルゴリズムについて簡単に説明する．Push-Relabel アルゴリズムとは，前述した Maximum-Flow 問題を解くためのアルゴリズムの一種である．この Push-Relabel アルゴリズムでは Push と Relabel と呼ばれる 2 つのグラフ処理をアクティブと呼ばれる状態の各頂点に対して結果が収束するまで繰り返し適用することで Maximum-Flow 問題を解くアルゴリズムである．

Push はグラフ上のアクティブ状態の頂点に対して適用することで各頂点に流入しているフローを隣接する頂点へと押し流す処理である．この処理の際に隣接する頂点の中でどの頂点にフローを押し流すかは各頂点が持つラベルという数値をもとに決定する．また，Push で押し流すフローの量ついては頂点間の辺が持つ容量と既に流れているフロー量をもとに決定される．この Push 処理を繰り返し適用することでフローグラフ上の始点から終点へとフローを伝播させていくことができる．

Relabel はグラフ上のアクティブ状態の頂点に対して適用し，各頂点が持つラベルを必要に応じて再計算・再割当てする処理である．このラベルの計算は隣接する頂点と辺のパラメータを基に計算される．この Relabel 処理を適用することで Push 処理でフローを押し流す経路を制御し，適切にグラフ上でフローを伝播させることができる．

以上のような Push と Relabel の 2 つの処理をグラフデータ処理やメッセージ送受信処理として実装を行い Maximum-Flow を求める．

4.2.3　GraphX 上での実装

本研究では GraphX 上で Push-Relabel プログラムを実装するに辺り，GraphXStyle と PregelLike という 2 つの実装方針を立てて実装を行った．前者は GraphX プログラミングらしい素朴な実装であり，後者は Pregel モデルというグラフ計算モデルを基に GraphX 上で実装を行ったものである．

GraphXStyle 実装は GraphX の設計に基づいた実装を行うものである．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 におけるデータ型の扱いと API の活用という点を重視したものである．1 つ目のデータ型の扱いは，GraphX 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に準拠するというものである．2 つ目の API の活用は，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．GraphX によるグラフ処理プログラムの実装では GraphX のグラフ処理 API だけでなく Spark Core やその他の API も合わせて用いることができる．その為処理内容が同等の処理を GraphXAPI を用いずとも表現が可能である．GraphXAPI とその他のどちらが実装が容易であるかは実装するグラフ処理の内容によって異なると考えられるが GraphXStyle 実装では可能な限り GraphXAPI にあるものを活用して Push-Relabel の実装を行うこととした．

PregelLike 実装は Pregel モデルと呼ばれるグラフ処理モデルを基にグラフ処理を GraphX 上で実装を行うものである．GraphXStyle に関しては GraphX による素朴な実装に行ったのに対し，こちらでは Pregel のモデルに基づいた既存実装 [10] を基に，GraphX で Pregel 処理を実装したものである．また，Pregel に基づく計算をするにあたりグラフデータの扱いについても GraphXStyle とは異なる方式を用いた．以下それぞれについて述べていく．

そもそも PregelLike 実装で用いている 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，頂点主計算によるグラフ処理を採用したグラフ処理モデルである．頂点主体計算とは，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

各頂点がそれぞれある程度独立して並列に計算を行う計算モデルである．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，この関数に基いてグラフ処理を進める．処理の流れとしては，まず何らかの条件にマッチする頂点がメッセージ送信関数によりメッセージ送信を行う．続いて送信されたメッセージが同一の宛先に複数存在する場合にメッセージマージ関数によりマージが行われる．最後にメッセージを受信した頂点はその情報を使って自身のパラメータを適宜更新する．以上の流れを結果が収束するまで繰り返す．

GraphX 上で一連のこの処理については aggregateMessages と joinVertices を用いて実装を行う．aggregateMessages は前章で説明したようにメッセージ集約処理を行うための関数であり，各 Triplet 毎にメッセージの送信の可否や方向，データの内容などを計算してメッセージの送信を行う関数である．この aggregateMessages を用いることで Pregel モデルにおけるメッセージ送信関数とメッセージマージ関数の処理を実装する．メッセージ適用関数に関しては joinVertices を用いて実装を行い，aggregateMessages で生成されたメッセージを書く頂点ごとに適用する．

GraphX では前述したような基本的グラフ処理 API の他に特定のグラフ処理向けの API や高度なグラフ処理 API が用意されている．その中の 1 つとして Pregel API がある．GraphX では Pregel モデルに基づいた計算をするための Pregel API が提供されており，この Pregel API 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る．ただし，この Pregel API を用いて実装をすることができるのは比較的単純なグラフ処理に限り，本ケーススタディの Push-Relabel のような複雑なグラフアルゴリズムでは利用することは困難であるため用いていない．

上記の Pregel モデル [6] に基づく計算を GraphX 上で実装するに辺り，前述したように GraphXStyle とは異なるデータ型の定義を行った．Pregel モデルでは頂点主体計算で処理を進めるため GraphXStyle で EdgeRDD に格納していた辺の情報を頂点のパラメータの一部として VertexRDD に格納した．従って PregelLike 実装のグラフは頂点集合のみで構成され，

辺を持たないグラフとなっている．また，メッセージ送信をする際は必要に応じて最小限の通信用の辺を一時的に生成することでコミュニケーショングラフを形成しメッセージのやりとりを行う．

以上が本ケーススタディで実装した Push-Relabel プログラムの概要である．

また，比較用に GraphX 上での Push-Relabel の既存実装を用意した [5]．こちらの実装は基本的には GraphX 実装と似た実装となっているが GraphX 実装で別々に行っていた Push と Relabel に関する操作を同時に行うようになっている．そのため 1 つのメッセージ集約処理や計算処理が複雑な実装となっている．

4.2.4　パフォーマンス評価実験

前述した GraphX 上で実装した Push-Relabel プログラムを用いてパフォーマンス評価実験を行った．実験内容としては各種実装に対して Maximum-Flow の問題グラフを与え，それを実際に処理させて時間を計測するというものである．入力には表 4.1 のように 6 頂点 8 辺の非常に小規模なグラフと 100 頂点 261 辺のグラフを用意した．実験環境としては Intel Core i5 2500(3.30GHz 2 core 4 threads)，8GB RAM の PC9 台で構成される Spark クラスタを用いた．また実験に使用した Spark はバージョン 2.0.2 である．次項にて実験の結果について説明する．

4.2.5　実験結果

実験結果は表 4.2 のようになった．

GraphXStyle 実装は他の実装と比べて非常に遅く PregelLike 実装は効率的実装になっているようであることがわかった．また，今回実装した 2 つの実装に関してはいずれも Graph2 を入力した際に DAGScheduler エラーが発生し処理を最後まで完遂することができなかった．この問題に関してだが，Push と Relabel1 回ずつ行う処理を 1 ステップとした時に GraphXStyle 実装の場合は 10 ステップ前後でプログラムが落ちてしまい，PregelLike 実装

の場合は 150 ステップ前後でプログラムが落ちてしまっている．多少実行順序によってステップ数は前後するが概ね Graph2 を入力とした場合には 200 ステップ前後で終了する．したがって GraphXStyle ではほとんど処理を終えないまま終わっており，PregelLike 実装では処理をある程度まで行えている．また，最終結果が出ないため Graph2 の処理時間は明記していないが途中経過に限って言えば Graph1 の場合と同様に Graph2 でも PregelLike 実装が最も処理を早く行っていた．既存実装はいずれの入力でも処理を終えることができたが Graph2 の場合処理時間は 240 秒以上もかかってしまった．これはグラフの規模から考えても非常に時間がかかっている結果である．

入力グラフ	頂点数	辺数
Graph1	6	8
Graph2	100	261

表 4.1　入力グラフ

実装	Graph1	Graph2
GraphXStyle 実装	19.90 秒	– 秒
PregelLike 実装	3.59 秒	– 秒
既存実装	4.86 秒	241.85 秒

表 4.2　実験結果

4.2.6　結果からの考察

Push-Relabel を用いたケーススタディから得られた知見について説明していく．

1 つ目に GraphX プログラミングにおける DAGScheduler 関係のエラーが発生した問題について述べていく．Spark では実際に RDD の計算を行う前にまず計算フローを表す DAG を形成する．その後あるタイミングでその DAG を辿りながら実際に RDD に対して演算を実行し結果を得るという処理形態を取っている．Spark ではイミュータブルな分散コレクションである RDD とこの DAG による計算フローの管理を駆使することで分散環境でのデータに対する耐障害性を高めたり，処理を効率的に行うための最適化を施したりすることを実現している．今回実装した 2 つのプログラムでは DAGScheduler に関するエラーが出るケースが多々見られ，ある程度の規模のグラフデータに対して処理を完遂できないという問題が発生した．この原因として考えられるのは今回実装した 2 つのプログラムにおいて

用いた中間データが挙げられる．Push-Relabel アルゴリズムの実装にあたり，アルゴリズムの性質上 Push や Relabel，加えてそれぞれの処理の中で幾つかの中間データを生成している．特に GraphXStyle では PregelLike 実装とは異なり Push や Relabel に必要な情報が辺や Destination の頂点に存在しており，Push などの計算を行うためには一度それらを Source の頂点に対して集約した後にグラフの外部の RDD 内で計算を行う必要がある．そのため頂点主体計算に沿ったデータ構造の定義をしている PregelLike 実装と比べてグラフ以外の中間データがより多く生成され，またそこでの計算が行われるという性質を持っている．この結果 Spark が管理している DAG が大きくなりすぎたり複雑になってしまった可能性があり，結果として処理フローを DAGScheduler がトレースする際にランタイムのキャパシティを超えてしまい DAGScheduler エラーが発生したと考えられる．その為 GraphX では複雑なアルゴリズムを実装する場合可能な限りグラフ演算の演算数を減らし，DAG の管理を考慮したプログラミングが必要である可能性が考えられる．ただし，今回の実装では過度に中間データを生成することはなくできるだけ可読性を保ったままの実装を行っているにも関わらずエラーが発生している．これは可読性や保守性とのトレードオフになっており，更に計算数を減らすために複数の処理をまとめたり中間データをユーザが作らないようにしてしまうと可読性や保守性が下がってしまう問題が発生する．

2 つ目に PregelLike 実装は効率の良い実装である可能性があるということである．PregelLike 実装は今回実験を行った中では最も処理速度が早いという結果が得られた．この要因として考えられるのは PregelLike 実装では辺を含めグラフに関する情報を頂点に持たせ，通信をする際には通信用に一時的な辺を生成して通信を行っているという点である．この Pregel モデルに沿った実装の GraphX におけるメリットとしては第一に辺の情報を伴う計算をする際であっても mapVertices のみで計算処理自体を実装することができ実装が完結になり，ランタイムが実行する処理としてもシンプルなものとなることである．これにより複雑な実装をすることなく効率的な処理ができると考えられる．第二に aggregateMessages における辺のスキャンを行うコストを下げることができるという点である．GraphX におけるメッセージ集約処理を行うための API である aggregateMessages では実際にメッセージ集約を行う前

に各 Triplet 毎に対してマッチング処理を施す必要が出てくる．その為暗黙的にグラフ上に存在する全ての Triplet に対してスキャンが走ることは避けられない．しかしグラフアルゴリズム，特に Push-Relabel のようなアルゴリズムでは実際にメッセージ送信をする必要のある Triplet は少なく，辺のスキャンにかかるコストが辺の数に伴って増加する場合は本来必要のない計算コストが大幅にかかっている可能性が考えられる．aggregateMessages では辺のスキャンを Triplet 単位で行うため辺の数がスキャン対象の数になるが PregelLike 実装では通信を行う必要のある辺のみを生成してネットワークを形成するためスキャン処理において無駄な処理が発生することは殆ど無い．ただし，この実装では通信を行う前に辺データを生成してグラフに統合するという処理が挟まるため一概に全体の処理コストが大幅に減るとは言えない場合が考えられる．このようなメリットから GraphX による実装の中では比較的高速な処理が行えたということが考えられた．しかし，GraphX の中では効率がいいプログラムであるとは言えてもグラフ処理フレームワークとしては処理性能が低く，今回実装したプログラムと同様のアルゴリズムを用いて Pregel+による実装を行ったケース [10] と比較した場合 1000 倍以上の処理時間がかかっており実用性のあるものであるとは言えないことは明らかである．

3 つ目の知見として Push-Relabel プログラムの中でも処理が重たいと考えられるのはメッセージ集約処理である可能性が高いということである．Push-Relabel における主な処理として Push 及び Relabel におけるメッセージ集約処理，集約した情報を用いたフローなどの計算の 2 つの処理をそれぞれ繰り返すがそれぞれについてのみ計算させるように試した所メッセージ集約処理，特に aggregateMessages が多くの時間を費やしているようであることが分かった．今回の Push-Relabel プログラムでは aggregateMessages を用いたメッセージ送信に関する計算が少し複雑な物となっており aggregateMessages のメッセージ集約そのものが重たい可能性と aggregateMessages 内で行っているユーザコードが重たい可能性の 2 つが考えられる．そのため次項ではこの点に着目し，単純な aggregateMessages と joinVertices で実装できる SSSP を持ちたケーススタディを用いて調査を行った．

4.3　ケーススタディ2.SSSP

本節では 2 つ目のケーススタディである SSSP について説明する．

4.3.1　SSSP 問題

このケーススタディの題材である SSSP(Single-Source Shortest Path) 問題とは頂点間の距離（移動コスト）をパラメータとして持つ重み付きグラフにおいて，ある単一の頂点からグラフ上の各頂点毎の最短距離・経路を求める問題である．本研究では最短距離を求め，経路については求めないこととする．

また，問題となるグラフにおける辺の重みについては任意の範囲でのランダムな値の場合と全ての辺が均一な重みである場合の 2 つ場合について扱う．この点の違いは，前者では 1 つの頂点に対して複数の経路が存在する場合にはその経路ごとに最短距離が異なる可能性がある．そのため最短距離が何度か更新され再計算が発生する可能性がある．対して後者は一度最短距離が設定された場合には複数の経路が存在したとしても辺の重みが定数であることから経由する辺の数が経路の長さになるため再計算発生しないという違いがある．したがって前者と比べて後者は計算コストが低く単純であることが特徴である．

4.3.2　SSSP の解法アルゴリズム

SSSP 問題を解くアルゴリズムには幾つかの種類があるがここではダイクストラ法を用いた実装を行う．このダイクストラ法について簡単に説明する．ダイクストラ法ではまず，各頂点が持つ始点からの距離のパラメータ（$d(v)$）を無限（最短距離不明），始点の $d(v)$ を 0 として初期化する．その後 $d(v)$ が更新された頂点 v は隣接する各の頂点 w に対して辺 (v, w) の重み $w(v, w) + d(v) < d(w)$ であれば $d(w)$ の値を更新するという処理を適用する．この処理を結果が収束するまで繰り返し適用することでグラフ上の各頂点と始点との距離を計算する．

このアルゴリズムでは更新された頂点を逐次的に処理していくが，GraphX などの並列グ

ラフ処理では各頂点について並列に計算を行うため更新された頂点が並列に処理される．

4.3.3　GraphX 上での実装

SSSP プログラムの GraphX 実装では Push-Relabel の実装の際に用いた 2 つの実装方針を用いて実装を行う．データ構造の定義や処理手順は基本的に Push-Relabel と同様のものであり，GraphXStyle では頂点を VertexRDD，辺を EdgeRDD へ格納し，PregelLike では全て VertexRDD に格納して頂点を持たないグラフを形成した．また，Push-Relabel の際との違いとしては SSSP のアルゴリズムは非常にシンプルであるため GraphXStyle による実装ではの Pregel API を用いた実装を行ったということである．PregelLike 実装に関しては処理内容がシンプルであるという点以外には違いはない．

GraphX による実装をするにあたり，SSSP の本処理を以下の形で実装することとした．

1. d(v) が更新された頂点 v は隣接頂点へ更新された値をブロードキャストする
2. 複数のメッセージを受信する場合は最小値をメッセージとするようにマージする
3. メッセージを受信した頂点はそれを基に必要に応じて自身の d(v) を更新する

PregelLike 実装ではこの内 1,2 のメッセージの送信とメッセージのマージについては aggregateMessages により実装される．3 のメッセージ適用については aggreagateMessages によって生成されたメッセージが格納された VertexRDD を joinVertices によって適用処理を実装する．GraphXStyle 実装では Pregel API を用いて実装を行うため上記の処理を実現するためのメッセージ送信関数とメッセージマージ関数，メッセージ適用関数を Pregel API に渡すことで処理を実装した．

Push-Relabel の GraphX 実装ではアルゴリズムの性質上いくつかの複雑な計算や通信が行われるが SSSP では通信や計算内容は非常に単純であることから Push-Relabel のケーススタディから考えられたメッセージ集約処理の問題や単純なアルゴリズムにおける GraphX の振る舞いがわかると考えられる．

4.3.4　パフォーマンス評価実験

前述した GraphX 上で実装した SSSP プログラムを用いてパフォーマンス評価実験を行った．実験内容としては頂点数約 87 万，辺数 510 万の Web グラフを入力として 2 種× 2 実装の系 4 つのプログラムで問題を解く実験を用いた．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 クラスタを用いた．また実験に使用した Spark はバージョン 2.0.2 である．次項にて実験の結果について説明する．

4.3.5　実験結果

実験結果は表 4.3 のようになった．

今回の実験では Push-Relabel の場合とは異なり GraphXStyle の方が高速という結果になった．この要因として考えられるのは GraphXStyle では Pregel API を用いたグラフ処理実装をしていることが影響していると考えられる．また，PregelLike 実装において入力グラフのデータが増大するというランタイムのバグが発生してしまい，計算ステップ数の多いランダムな重み付きグラフの場合に処理が終わらないという問題が発生した．

実装	ランダムな重み付きグラフ	定数の重み付きグラフ
GraphXStyle 実装	20.70 秒	16.25 秒
PregelLike 実装	– 秒	25.66 秒

表 4.3　実験結果

4.3.6　結果からの考察

SSSP を用いたケーススタディから得られた知見について説明していく．

まず今回の実験では結果から明らかなように Push-Relabel の場合とは異なり GraphXStyle 実装の方が PregelLike 実装よりも高速且つ安定した動作を実現できているということがわかる．SSSP は非常にシンプルなグラフアルゴリズムであるため GraphX の Pregel API に

当てはめて容易に実装することが可能である．この API ではメッセージ送信関数 sendMsg，メッセージマージ関数 mergeMsg，頂点更新関数 vprog を与えることで自動的に Pregel モデルに沿ったグラフ処理を実現してくれる．この API の実装としては aggregateMessages によるメッセージ集約や joinVertices によるその適用を繰り返すというものになっており，基本的には Push-Relabel のときの GraphXStyle とは処理内容に大きな差はない．しかし Pregel モデルによる処理というスケルトンを用いたグラフ処理実装を行うため Pregel クラス内で Spark レベルでの様々な最適化やチューニングが施されている．これにより簡易的な実装ながら PregelLike 実装よりも高速に処理ができたと考えられる．これは裏を返せばグラフ処理 API を単純に用いただけではパフォーマンスを発揮するのは困難であり，SSSP のようなメッセージの集約とその適用を繰り返すというだけの単純なグラフアルゴリズムであっても適切なチューニングを施さねばパフォーマンスが容易に低下してしまう可能性を表していると考えられる．

2 つ目は SSSP における GraphX のパフォーマンスについてである．SSSP ではここまでで何度か説明したようにメッセージの送信及びその適用をただ繰り返すだけのプログラムであり，それらの内部で用いられる数値処理も非常に単純なものである．Push-Relabel ではグラフ処理の中では非常に複雑なものを扱っていたのに比べ単純な内容である．しかしこのような単純なグラフアルゴリズムであってもグラフ処理性能は低く，プログラムの実行に問題が発生する事もあった．Push-Relabel のような複雑なモノに限らず SSSP のようなグラフ問題であってもこのような問題が発生することから，GraphX における基本的なグラフ処理 API である aggregateMessages など API，あるいはそれら GraphX API を実装している Spark そのもののいずれかがに問題があると考えられる．

4.4　GraphX によるグラフ処理プログラムの実装に関する問題

2 つのケーススタディで生じた GraphX プログラミングにおける問題について述べていく．2 つのケーススタディを通して GraphX プログラミングを一通り行って頻繁に起こった問題

として GraphX 及び Spark におけるバグらしき挙動が多々見られたことが挙げられる．今回実装したいずれのケーススタディでも何かしらプログラムの実行に問題が起こるケースが発生した．この問題はユーザプログラムのレベルではプログラム的に間違っていることやバグを作っているということはないと考えているのにも関わらずランタイム側でエラーが発生し，プログラムが正常に動作しないケースが起こりうるということである．Push-Relabel のような複雑なアルゴリズムに限らず SSSP のような単純で基本的アルゴリズムであっても問題が起こるというのはグラフ処理フレームワークとしては致命的であると考えられる．

4.5　ケーススタディからの考察

2 つのケーススタディを総括して分かったこととその考察についてのまとめる．

GraphX におけるグラフ処理性能の問題は従来のベンチマークに用いられるような単純なアルゴリズムだけではなくよりアプリケーション寄りの複雑なプログラムにおいても顕著に現れており，特定のケースにのみ発生するような問題ではないことが明らかとなった．いずれの場合でもパフォーマンスが低いことから SSSP の節で述べた様に GraphX におけるメッセージ集約処理を行う aggregateMessages やその実装に使われている Spark Core プログラムなどに原因がある可能性がある．

aggregateMessages に関する問題として考えられる要因として 2 つ考えられる．1 つ目に PregelLike 実装についての言及で言及した辺のスキャン処理，そして Triplet でのデータのやり取りなどが考えられる．前者は前述した通り aggregateMessages では暗黙的にグラフ上全ての辺及び Triplet に対してスキャンを実行するため明らかにスキャンが不要である辺に対してもスキャンを行ってしまうという問題である．後者は GraphX におけるパーティション管理方法の問題である．Spark における分散コレクションの RDD は通常は複数のパーティションに分けられてワーカに分散配置され，それぞれのパーティションごとに計算処理を実行する．GraphX のグラフも同様に VertexRDD と EdgeRDD の 2 つの RDD を用いて処理を行っているため辺と頂点のデータは複数のパーティションに分割されて処理が行われ

る．aggregateMessages では Triplet 単位で処理を行うため Source と Destination の頂点及びそれを繋ぐ辺の 3 つが必要となるがこれらを適切なパーティションに配置する必要があるためパーティションの再編成やワーカ間での通信などが発生する可能性がある．この処理が重たい場合 aggregateMessages の処理性能の問題になっている可能性があると考えられる．

以上より GraphX のパフォーマンス問題の一因としてメッセージ集約処理，特に aggregateMessages についての問題を明らかにすることが必要である．

第 5 章

Spark GraphX におけるメッセージ集約処理のベンチマーク実験

前章のケーススタディの結果から GraphX のパフォーマンス問題の一因として GraphX におけるメッセージ集約処理の性能が考えられるわかった．本章ではそのメッセージ集約処理に関する問題について明らかにするために行った GraphX のメッセージ集約処理に関するベンチマークについて述べていく．

5.1　グラフ処理とメッセージ集約処理

今回行ったベンチマークなどについて説明する前にグラフ処理におけるメッセージ集約処理について説明する．

グラフ処理におけるメッセージ集約処理とは，グラフ上に存在するデータなどをメッセージとして送信し，それを何らかの形で集約する処理である．具体的な例を挙げるならばグラフ上の各頂点が自身の持つパラメータ送信し，それを集めて総和を取るような処理，Push-Relabel アルゴリズムで各頂点が自分の Push するフローの値を隣接頂点へ送信し，各頂点でそのメッセージを集約するような処理が該当する．この例の場合，前者の処理ではグラフ全体に対するメッセージ集約処理であり，後者は各頂点でのメッセージ集約処理である．

このようなメッセージ集約処理はグラフにおいて各頂点や辺，その部分集合など様々なグラフ要素間で情報をやり取りするための基本的グラフ処理であり，ほとんどのグラフアルゴリズムで頻出するものである．つまりグラフ処理を行うにあたって基本的にはこのメッセー

ジ集約処理を避けて通ることは難しく，グラフアルゴリズムを実装する上で必要不可欠なグラフ処理である．従ってこのグラフ処理そのものにパフォーマンス的問題がある場合殆どのグラフアルゴリズムにその影響が発生し，処理性能を下げてしまうこととなる．

5.2 GraphX におけるメッセージ集約処理 API aggregateMessages について

前述したメッセージ集約処理を GraphX 上で実装する場合，多くの場合 aggregateMessages を用いることとなる．aggregateMessages はその名前の通り GraphX においてメッセージを集約するための API である．実際に行う処理内容としてはグラフ上の各 Triplet 毎にメッセージ送信関数を用いてメッセージの送信の可否や方向，データの内容などを計算してメッセージの送信を行う．また，メッセージが同一の宛先に複数存在する場合はマージ関数により 1 つのメッセージにマージされる．この際用いられるメッセージ送信関数とマージ関数はユーザが引数として与えるものであり，これにより任意のメッセージ集約処理を実装することができる．この aggregateMessages により集められたメッセージはメッセージの宛先の ID とそれに対応するメッセージデータの組による VertexRDD として生成され，この VertexRDD を用いて集約したメッセージを用いた処理を行う．

また，aggregateMessages による辺のスキャン処理を効率化するための API として aggregateMessagesWithActiveSet と云う API も用意されている．この API では通常の aggregateMessages の処理に加えて，通信が発生する頂点（以下アクティブ頂点）を集めた ActiveSet と呼ばれる VertexRDD を引数として渡すことで不必要な辺走査を削減することが可能であるという API である．この aggregateMessagesWithActiveSet によって辺走査が削減されるのはグラフ上の全頂点うちのアクティブ頂点の割合（以下アクティブレート）が一定の値以下であり，不必要な走査が多く行われると判断された場合に実行される．逆にアクティブレートが高い場合は通常の aggregateMessages と同じ処理となる．

5.3　aggregateMessages と aggregateMessagesWithActiveSet の比較実験

ここまでのケーススタディなどではメッセージ集約処理を行う際 aggregateMessages を用いて実装を行ってきており，aggregateMessagesWithActiveSet を用いた実装は行っていない．また，SSSP の GraphXStyle で使用している Pregel API についても内部的には aggregateMessages を用いているため同様である．前章末にて aggregateMessages に対する予想される問題点として辺のスキャンに関するコストを挙げているが aggregateMessagesWithActiveSet がそのコストを下げ，メッセージ集約を効率的に行えるのであればこの点については問題ではなくなる．そこでまずこの 2 つのメッセージ集約 API に対してベンチマーク実験を行った．

5.3.1　実験内容

ベンチマークの内容としては，入力グラフ上のアクティブ頂点からその頂点を Source とする辺の Destination に対して Int 型のデータを送信し，その値を頂点のデータに加算していくという処理を一定回数ループさせるものである．具体的な GraphX による実装は aggregateMessages（WithActiveSet）により Int 型の数値 “1” を送信し，それにより生成された VertexRDD を用いて joinVertices により Destination のパラメータに加算するという処理になる．この際のアクティブ頂点の決定方法については VertexID の下一桁が閾値より高いか低いかで決定するものとした．また，グラフ計算を実行させるタイミングを制御させるためにループを抜けメッセージ集約が終わった後には count により要素数の集計を行う．

実験に用いる入力グラフは SSSP の実験でも用いた Web グラフと同じもの，加えて約 100 万頂点，300 万辺の道路ネットワークを入力グラフとして与え 100 回メッセージ集約とその適用処理を行わせた．また，aggregateMessagtesWithActiveSet ではアクティブレートに応じてスキャン方法を切り替えて効率化をするためアクティブレートが 10%の場合と 90%場合の 2 パターンで実験を行った．

実験環境は Intel Core i5 2500(3.30GHz 2 core 4 threads)，8GB RAM の PC9 台による Spark 環境を用いた．また実験に使用した Spark はバージョン 2.2.0 である．次項にて実験の結果について説明する．

次項にて実験結果を示す．

5.3.2 実験結果と考察

API	Web グラフ 10%	Web グラフ 90%	道路 NW10%	道路 NW90%
aggregateMessages	253.84 秒	520.60 秒	180.83 秒	497.63 秒
WithActiveSet	273.42 秒	560.30 秒	209.54 秒	529.28 秒

表 5.1 実験結果

実験結果は 5.1 の様になった．いずれのグラフやアクティブレートでも aggregateMessages が aggregateMessagesWithActiveSet よりも早く処理を終えているという予想に反した結果となった．aggregateMessages は実装として引数の Active セットを無しで aggregateMessagesWithActiveSet を呼び出す形で実装されており，aggregateMessagesWithActiveSet との違いとしては ActiveSet を用いた辺走査をするための前処理として withActiveSet メソッドを用いた前処理が行われるという点とその後の辺走査を行うときに呼び出される走査メソッドが異なるという 2 点である．これから考えられることは aggregateMessagesWithAciveSet による辺のスキャンの効率化による削減コストよりもそのための前処理などにかかるコストのほうが重いという可能性である．また余談ではあるが，aggregateMessagesWithActiveSet が定義されている GraphX におけるグラフのクラスである Graph クラスには定義されていなく，実際に実装を行っている GraphImpl クラスのメソッドである．このメソッドは Graph クラスからの直接的な呼び出し方では使用することができない実装となっており，ユーザ側で意図的にキャストを行った上で呼び出さなければ ActiveSet のオプションを用いることができない．加えて aggregateMessageWithActiveSet について前述した様に Pregel API のよう

なグラフ処理用のクラスやメソッドの実装においてもこのメソッドが用いられている場面は全くなく，GraphX の現状としては効率的でないことが明らかであるため使用していない可能性が考えられ使用されていないものであると考えられる．以上から aggregateMessages による辺の走査コストの削減は aggregateMessagesWithActiveSet を用いることによっては行えないということが分かった．

5.4　メッセージ集約処理のベンチマーク実験

続いて前節では aggregateMessages 及び aggregateMessagesWithActiveSet の API 毎の性質に対して実験を行ったのに対して入力グラフごとの性質に対するベンチマークを行った．既に何度も述べているが aggregateMessages における予想される問題点として辺のスキャンのコストの問題がある．もしもこの点について真であるならば aggregateMessages の処理時間は辺の数に応じて増減するはずである．また，辺の他の要素として頂点の数についても実験を行った．

5.4.1　実験内容

ベンチマーク内容は aggregateMessagesWithActiveSet に関するベンチマークを行うときに用いたものと同一の内容である．入力グラフに関しては出次数に偏りのないランダムグラフを用いて実験を行い，辺の数もしくは頂点の数を定数で固定しもう一方を変化させて辺と頂点の変化に対する影響を調べた．またこの実験ではアクティブレートを 10%として設定した．

実験環境は Intel Core i5 2500(3.30GHz 2 core 4 threads)，8GB RAM の PC1 台による Spark 環境を用いた．また実験に使用した Spark はバージョン 2.2.0 である．次項にて実験の結果について説明する．

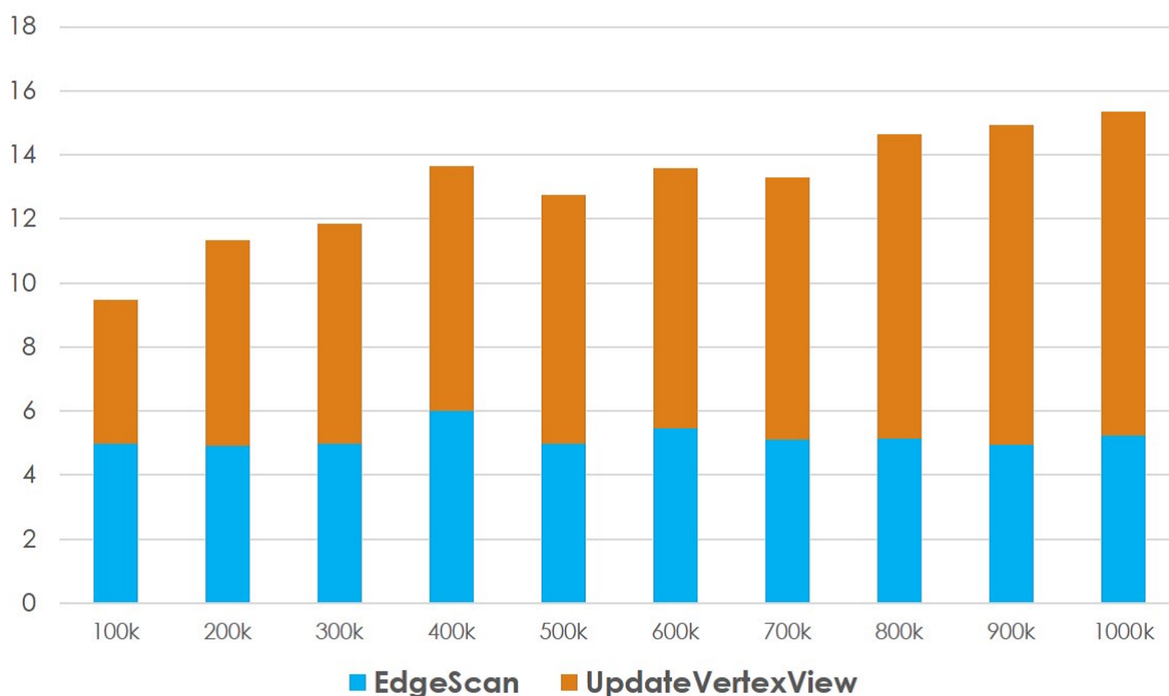

図 5.1　頂点数を変化させた場合

5.4.2　実験結果と考察

辺の数を 50 万に固定し，頂点の数を変化させた場合の結果を図 5.1 に示す．

グラフのうち EdgeScan は aggregateMessages にかかった時間のうちの辺のスキャンとその後のメッセージ集約にかかった時間の累計，UpdateVertexView は aggregateMessages にかかった時間のうちの UpdateVertexView にかかった時間の累計を示しており，その和が aggregateMessages にかかった時間の累計時間を示している．ちなみに集約されたメッセージを joinVertices によって各頂点に適用している処理と最後に行っている count，そして実際に計算を行う前に DAG を形成している時間についてはこの処理に含まれておらず，その処理時間はほぼ定数時間且つ短時間で終えられるためここでは触れないものとする．ここで出て来る UpdateVertexView とは GraphX における VertexView の更新処理を意味しており，それについて簡単に説明する．

GraphX では EdgeRDD と VertexRDD の計 2 つの RDD を扱うがこのうち VertexRDD は少し特殊な管理方法をしている．GraphX の概要について触れた際に説明したように GraphX

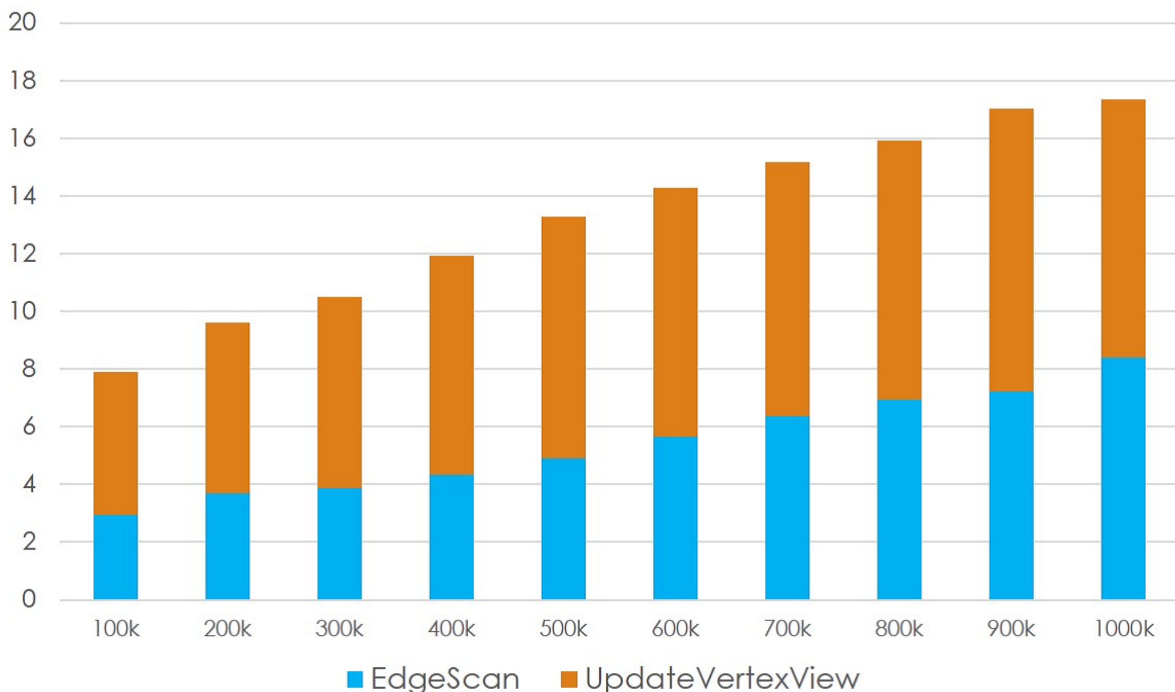

図 5.2　辺数を変化させた場合

では Triplet という処理単位が存在している．aggregateMessages や幾つかのグラフ処理ではこの Triplet 単位で処理を実行する．そのためケーススタディに対する考察の項で述べたようにグラフ上での辺と頂点の関係性とパーティションの配置が重要になっており，適切に処理するために ReplicatedVertexView と呼ばれるクラスを用いて辺からの頂点に対する View を管理している．この View を用いることで各辺や辺のパーティションから見たときにその Source と Destination の頂点が持つパラメータがどこに存在しており，どのように送信・配置することで情報を取得・設定できるかを管理している．この View を現在の状態やその後に行う計算の性質や必要な情報の種類に合わせて更新する処理が UpdateVertexView である．

ここで図 5.1 を見てみる．この実験では辺の数を 50 万に固定して頂点の数を 10 万から 10 万単位で増加させている．累計の時間は頂点数の増加に伴い処理時間が増加している．また，この実験の場合 EdgeScan に関してはほとんど変化がなく，主に UpdateVertexView が aggregateMessages の増加に影響していることがわかる．

続いて辺数を変更した場合の実験の結果である図 5.2 を見てみる．こちらでは頂点の

数を 50 万に固定し，辺の数を 10 万から 10 万単位で増加させている．この実験では aggregateMessages の処理時間の累計時間及び EdgeScan の推移に対して先程の実験との違いが見られる．こちらでは UpdateVertexView のみならず EdgeScan も増加しており，結果として前の実験よりも aggregateMessages の増加率が増している．

これらの結果からわかることとして 1 つ目に aggregateMessages における EdgeScan 及びメッセージ集約処理は主にグラフ上の辺の数に影響を受けて増減することである．これは GraphX における aggregateMessages の辺のスキャンの性質から考えて予想通りであり妥当な結果である．2 つ目にわかることとして UpdateVertexView に関しては辺と頂点いずれの要素の増減に対しても影響を受けるという点である．また，UpdateVertexView の推移としてどちらの実験でも似た結果となっていることもわかる．これは UpdateVertexView を実行する際，辺のパーティション毎に頂点のパーティションの走査・演算が実行される．したがって概ね頂点数と辺数の積に相当する計算コストが発生する．2 つの表を見た時，どちらの場合でも概ね同様の変化が見られるのはこの性質によるものだと考えられる．

第 6 章

考察と提言

各種ケーススタディ及び aggregateMessages に対するベンチマークを通しての考察と提言について述べていく．

2 つのケーススタディによるアプリケーションの実装から aggregateMessages が行うメッセージ集約処理が処理速度低下の一因と考え，aggregateMessages について調査した所無駄な辺のスキャンなどによる処理時間の増加と VertexView の更新処理の処理の重さが明らかとなった．メッセージ集約処理を担う aggregateMessages は GraphX におけるグラフ処理 API として最も基本的な処理の 1 つであり 1 度のプログラムの実行で何度も実行されることは常である．大規模データに対する並列分散処理を目的とした Apache Spark のグラフ処理コンポーネントの API としてはこのようにグラフの規模の増大に伴い大きく増加することは大きな問題であると考えられる．したがって GraphX を実用的なレベルで運用して行くためには aggregateMessages の処理性能の改善が必要不可欠である．

1 つ目の問題である EdgeScan に関しては辺の数に応じて処理時間の増減が起こることは明らかである．多くのグラフアルゴリズムでは常に殆どの頂点が通信を行うことはあまりなく，グラフ全体のうちごく一部が通信などの処理を実行しそれが伝播隣接頂点間で伝播していくことで全体の処理が進んでいく．そのため全体の辺の数に処理時間が影響されるということはオーバーヘッドが大きいということである．この問題に対する対処として考えられるのは本研究のケーススタディで用いた PregelLike 実装のように必要最低限の辺のみを持ちそれを用いて通信を行うという処理を実装することである．今回のケーススタディでは十分なチューニングが施せなかったことや Spark ランタイムのバグなどにより正常に処理が行えない場合が見られたが処理性能は優秀だと考えられ，Spark の振る舞いに適した実装を施し，

GraphX 上でのユーザレベルの実装ではなく GraphX の内部実装などのランタイム側での実装を行い API を提供することで GraphX のパフォーマンス向上を図ることができると考えられる．

2 つ目の問題である UpdateVertexView に関しては GraphX におけるデータ管理に関する問題である．その為データ更新の方法を変更するもしくは VertexRDD と EdgeRDD の 2 つの RDD で管理するという方式を変更することで対応する必要がある．データの管理方法を変更するアプローチの場合本研究のケーススタディで用いた PregelLike 実装の様に VertexRDD によって辺のデータも一括管理するという方法がある．現在の GraphX のデータ管理では 2 つの RDD にデータが別れており，それは効率的に実装することを考えてのことだと思われるが概念上のグラフ上での頂点と辺の関係性と実際に分散環境上配置している場所の関係性が影響していることからアップデートやそのやり取りが必要以上複雑になっている可能性が考えられる．頂点主体計算的な計算モデルに基づいて VertexRDD のような 1 種類の RDD に各頂点のデータとその頂点が持つ辺の情報を持たせ，それを用いた実装を適切に行う事ができれば各頂点からのメッセージ送信は mapVertices またはそれに類する処理を用いて実装することができ，その適用も従来 joinVertices によって実装することができると考えられる．そうすることで頂点と辺を独立した RDD として扱うことができなくなる代わりに 1 種の RDD によって管理することができ，少なくとも VertexRDD と EdgeRDD の 2 つによって管理することによるコストを軽減することが可能性があると考えられる．

以上の 2 つの提言について共通して言えることとして現在の GraphX の計算モデルは Apache Spark による分散環境上での並列処理には適していない可能性があり，GraphX の性能改善をするためには現在のグラフデータの管理モデルを根本的に変更する必要がある．現在の GraphX ではメッセージ集約処理などの重要な処理に関して実質的に辺主体モデル的な処理手法を取っている．辺データはグラフデータにおいて Source と Destination の頂点と接続しており，その関係性を無視して処理することはできず，Spark のようなデータが分散配置されている環境ではそのデータのやり取りが複雑になってしまう．この複雑さが大きな問題となっているためこれを軽減するという観点から辺のデータを Source か Destination（も

しくはその両方）の持つパラメータとして各頂点に持たせた頂点主体モデル的なデータ管理を行い，現在のデータ管理方法よりも処理単位のデータの独立性を高めることで分散環境上で管理する際にかかるオーバーヘッドを削減することが可能であると考えられる．当然これはケーススタディのような GraphX 上でのユーザレベル実装では僅かな性能向上しか得られないため GraphX の内部実装から手を入れることは避けられないものであると考えられ容易ではない．

第 7 章

まとめ

本論文では Apache Spark は大規模データ処理を高速に処理することを目指して設計された並列分散処理フレームワーク Apache Spark が持つグラフ処理コンポーネント GraphX におけるメッセージ集約処理の問題点を明らかにした．現状の GraphX ではパフォーマンスの問題や動作の安定性の問題が有り，実用的運用には幾つかの課題が残されていると考えられる．特にメッセージ集約処理というグラフ処理における基本的グラフ操作において問題があることは致命的であり実用的運用を考える上では改善することが必須である．

今後の課題として今回提言した幾つかの新たな GraphX のデータ管理方法やメッセージ集約処理の手法の実装をすることで実際に GraphX のパフォーマンスの改善が図ることができるかについて調査することが必要である．また，今回の aggregateMessages に関する性能調査ではグラフに関する性質を中心に調査を行ったが aggregateMessages によるメッセージ集約処理の内部で実行される計算処理の複雑さ・計算量などに対する影響などについては調べることができていない．Push-Relabel のような計算では著しく処理速度が遅いという結果から aggregateMessages 内で処理される計算コストもメッセージ集約処理にどの程度影響があるかを調べることは有益な情報であると考えられる．

謝辞

本研究を行うにあたり, 指導教員である高知工科大学情報学群の松崎公紀准教授から, 様々な研究題目を提案して頂いたことや, 研究の進め方についてアドバイスをして頂いたこと, 参考となる資料の提供など長期に亘って多くのご指導を頂きました. ここに, 深く感謝致します. また, 本論文の副査をお引き受けいただいた鶴川始陽准教授, 高田喜朗准教授に心からお礼を申し上げます. 最後に松崎研究室の皆様には, 研究を進めるにあたって, 様々な意見や感想, 助言を頂きました. 本当にありがとうございました.

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., 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Apache Giraph, "`http://giraph.apache.org`", 2017.