Spark GraphX におけるメッセージ集約の性能解析

1205085　松本 拓也　【 高度プログラミング研究室 】

Performance Analysis of Maggage Aggregation in Spark GraphX

1205085　Takuya Matsumoto　【 High-Level Programming Lab. 】

1　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．様々なビッグデータの種類の中の代表の 1 つとしてグラフデータが挙げられる．

ビッグデータに対するグラフ処理の需要が高まりに伴いグラフ処理を高速処理するための手法として近年並列グラフ処理が注目されている．

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，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 はその並列グラフ処理フレームワークの 1 つである．

この Spark GraphX は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 [1]．

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．

2　Spark GraphX

2.1　GraphX の概要

GraphX は並列分散処理フレームワーク 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．Apache Spark は汎用性と高速性を目指したフレームワークでありオンメモリでの分散処理，RDD などの特徴を持つ．

Spark プログラムでは処理データを RDD と呼ばれる分散コレクションを用いて管理しており，GraphX も同様に 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頂点データの集合と辺データの集合をそれぞれ VertexRDD と EdgeRDD と呼ばれる RDD の拡張コレクションに格納する．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と辺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．

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．また GraphX で用いられるデータは RDD であるため Spark Core で用いる集合演算をほぼそのまま適用することが可能である．

2.2　GraphX API

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，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．Triplet とは辺と辺に接続している 2 つの頂点をあわせた組である．これにより各 Triplet が持つパラメータなどの情報を各頂点に送信することができる．

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 map 処理で新たな頂点のデータを再計算する処理である．グラフ処理ではメッセージの送信と適用という処理は頻出するものであり，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 Vertices, Edges, Triplets の 3 つの map を提供している．これらは GraphX におけるグラフデータの構造に合わせた map ではあるが処理内容そのものはコレクションに対する通常の map と同様である．

3　ケーススタディ

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，パフォーマンス問題を招く要因があると考えられる．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幾つかのグラフ処理プログラム用いたケーススタディを行った．

3.1　2 つのケーススタディ

1 つ目のケーススタディは Push-Relabel アルゴリズムである．Push-Relabel は Maximum-Flow 問題を解くためのアルゴリズムである．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で用いられているアルゴリズムは SSSP や BFS，PageRank など

表 1　辺数 50 万で固定で頂点数を変化させた場合

頂点数	100k	200k	300k	400k	500k	600k	700k	800k	900k	1000k
EdgeScan (秒)	4.967	4.922	4.979	6.01	4.98	5.464	5.116	5.132	4.94	5.229
UpdateVertexView (秒)	4.507	6.407	6.868	7.636	7.777	8.135	8.183	9.5	9.985	10.12
Total (秒)	9.321	11.121	11.648	13.418	12.558	13.39	13.082	14.388	14.689	15.113

表 2　頂点数を 50 万で固定で辺数を変化させた場合

辺数	100k	200k	300k	400k	500k	600k	700k	800k	900k	1000k
EdgeScan (秒)	2.959	3.703	3.86	4.323	4.89	5.65	6.354	6.944	7.218	8.395
UpdateVertexView (秒)	4.926	5.893	6.629	7.589	8.385	8.646	8.817	8.992	9.824	8.948
Total (秒)	7.77	9.444	10.327	11.745	13.074	14.05	14.914	15.647	16.773	17.033

グラフアルゴリズムの中では比較的単純なものが多い．一方で Push-Relabel で解く Maximum-Flow 問題はフローネットワークを扱う問題であり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．その為より実用的・具体的なアプリケーションの実装に沿った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．

2 つ目は SSSP である．こちらは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．選んだ理由は，Push-Relabel によって得られた知見を基に単純なグラフアルゴリズムの場合との振る舞いの違いや共通点を調べる目的である．

4　ケーススタディから得られた知見と考察

前述したケーススタディから得られた知見について述べていく 1 つ目の Push-Relabel では処理速度が非常に遅く処理を完遂できないケースが発生するなど非常に不安定な動作が見られた．アルゴリズムの性質上，一連のグラフ処理を幾つかの段階に分けて実装を行う．そのためで処理の中間データとなるグラフが生成される．これにより Spark が処理フローの管理に用いる DAG が複雑且つ膨大なものとなり，処理系の動作が不安定になってしまったと考えられる．2 つ目の SSSP では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，それらに用いられる演算も非常に単純なものである．これは Push-Relabel と比べ非常に単純な内容である．しかし処理速度が遅く，プログラムの実行に問題が発生する事もあった．SSSP の結果から基本的なグラフ処理 API であるこの 2 つ，あるいはそれらを実装している Spark そのもののいずれかがに問題があると考えられる．

5　メッセージ集約処理のベンチマーク

GraphX によるグラフ処理ではメッセージ集約処理を aggregateMessages と joinVertices を用いて実装を行う．したがってこの 2 つを用いて頂点からデータを送信し，宛先の頂点のパラメータに適用する処理をループさせ，最後にグラフ要素の集計を行うという内容のベンチマークを用いて調査を行った．辺数 50 万辺に固定し頂点数を変数とした場合の結果を表 1 に，頂点数を固定し辺数を変数とした場合の結果を表 2 に示す．実験の結果 joinVertices 及びグラフの集計を行う関数に関しては処理時間全体のほんの僅かな時間であったのに対して処理時間内殆どが aggregateMessages が占めていたことが分かった．表の内 Total が aggreageteMessages が処理を行っていた時間の合計で EdgeScan は実際に辺を走査してメッセージを生成する処理である．EdgeScan に関しては辺を走査するという点から辺の数が変わらなければ変化処理時間は変化せず辺の数の増加に伴い処理時間が増加することがわかる．UpdateVertexView は GraphX における頂点 RDD のパーティションを管理するための VertexView を更新する処理である．aggreagateMessages が実行される際にはその設定に応じて View を更新する処理が行われ，その処理は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．従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れによるものである．

6　まとめ

GraphX におけるグラフ処理のパフォーマンス低下の一因としてメッセージ集約処理に関する API の問題が明らかとなった．グラフ処理においてメッセージ集約処理は基本的演算でありこれがボトルネックであることはグラフ処理フレームワークとして致命的であるため実用に耐えうるためには改善が必須であると考えられる．

参考文献

[1] Iosup, A.: 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.