Vol.2017-OS-141 No.16
2017/7/27

Apache SparkのSerialize処理最適化による処理速度向上手法

稲垣 英夫[1]　川島 龍太[1]　松尾 啓志[1]

概要：ビッグデータの解析手法として，クラスタ環境を用いた並列分散処理が注目されており，並列分散処理を効率的に記述するためのフレームワークとして Apache Spark が普及している．Spark はデータに対して多段な処理が発生する場合でもメモリ上で処理するため，他のフレームワークと比較してディスクアクセスの回数が減少し，パフォーマンスが向上する．しかし，計算機間でのデータ転送を必要とする Shuffle 処理では，大量のデータが転送されるため，処理全体の性能が低下する．Shuffle 処理の際，転送元の計算機において送信データの Serialize が行われ，同様に転送先の計算機において受信データの Deserialize が行われる．Serialize/Deserialize は Word サイズや Byte オーダなどの仕様が異なる計算機間のデータ転送では有用であるが，Spark のプロセスはすべて JVM 上で動作しているため，このような仕様はすべて統一されており，Serialize/Deserialize は簡素化できる．そこで本研究では Serialize/Deserialize 処理を最適化してデータ転送性能の向上を図る．具体的には，送信データに対する Serialize を簡素化して，クラス名などの省略可能な情報を付与せずバイト列変換のみを行い，転送先では受信データをバイト列のままデータとして扱うことで処理速度が向上する．評価から，JavaSerializer と比較した際に Serialize/Deserialize 処理にかかる時間を最大 57.3%削減，全体の処理時間を最大 19.8%削減し，手法の有効性を確認した．

キーワード：並列分散処理, Apache Spark, Serialize, Deserialize

1. はじめに

近年，産業界では大規模データを収集し，事業の拡大に活用する取り組みが活発であり，収集されるデータのサイズは年々増加している [1]．しかし，1 台の計算機ではこのような大規模データを処理できないため，複数台の計算機による並列分散処理を行う方法が注目されている．しかし，並列分散処理を実現するには，計算機間の通信処理や障害発生時の処理を考慮する必要があり，プログラミングに熟達していない人には記述が困難である．そこで並列分散処理を効率的に記述するために様々なフレームワークが開発されており，その中でも Apache Spark[2] が普及している．Spark は他のフレームワークと比べ，データをメモリに保存することで入出力を高速化し，処理速度を向上させている．特に，機械学習で見られるようなデータの繰り返し処理においては，Apache Hadoop[3] と比較して 100 倍近く性能向上するという結果が示されている [4]．

Spark では，クラスタ内でデータを再分散する Shuffle 処理が行われる．この時，大量のデータが転送されるため，処理全体の性能が低下する．Shuffle 時に転送元の計算機で Serialize 処理を行い, 転送先の計算機で Deserialize 処理を行う．Serialize/Deserialize は，Word サイズや Byte オーダなどの仕様が異なる計算機間のデータ転送では有用である．しかし，Spark では全てのプロセスが JVM 上で動作するため，このような仕様はクラスタ内計算機で変化しない．また，Shuffle 時に転送されるデータは全て同一のクラスであるため，データを送信する際にその計算機上でクラスの情報を保持しておけば，他の計算機から受信したデータを元の形式に復元できる．したがって Spark ではデータを転送する際の Serialize/Deserialize 処理を簡素化することが可能である．加えて，最近ではクラスタ内のネットワーク帯域幅は 1Gbps から 10Gbps へと増加しており，データを転送するコストは小さくなったが，それに伴って Shuffle 処理中のデータを Serialize/Deserialize 処理する時間の割合がより大きくなった．

そのため，本研究は Spark の Shuffle 時における Serialize/Deserialize 処理を最適化することで，データ転送性能を向上させ，処理時間の高速化を図る．提案手法では，KeyValue 型のデータを転送する際，転送元の計算機ではデータに対する Serialize を簡素化して，クラス名などの省

[1] 名古屋工業大学
Nagoya Institute of Technology

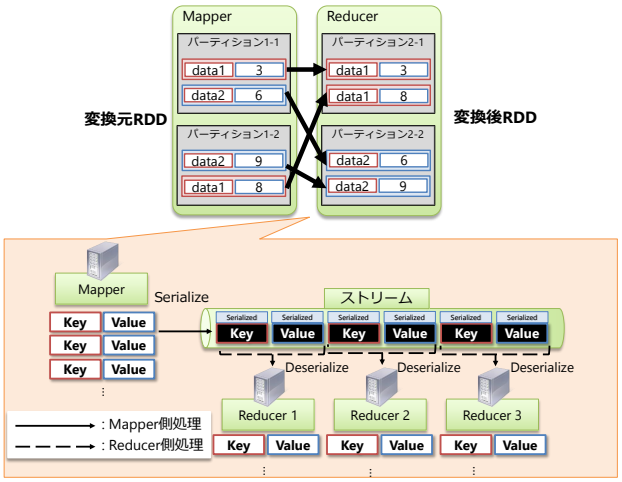

図 1　Shuffle 処理の流れ

略可能な情報を付与せずバイト列変換のみを行い，転送先の計算機ではデータの Value 部分のみを復元するように変更する．Spark では，Key を比較してから対応する Value に対する演算を行う処理が多く，Key をバイト列のまま保持しても処理速度を落とさずに実行できる．

本稿の構成は以下の通りである．まず，第 2 章では研究背景として Spark の内部動作の詳細と Shuffle 時の問題点を述べ，第 3 章で本研究での提案とその実装について述べる．また第 4 章で性能評価，考察を行い，第 5 章で関連研究を述べる．最後に第 6 章で今後の課題を述べ本稿をまとめる．

2. Apache Spark

本章では，Spark の内部動作の詳細を説明し，Shuffle 時における問題点を述べる．

2.1 Spark の概要

Spark クラスタは Master-Slave 方式を採っており，データソースとして分散ファイルシステムの HDFS[5] や分散 KVS の Cassanndra[6] などを利用できる．Spark では RDD (Resilient Distributed Dataset) というデータ形式を用いることで，データに対して多段な処理が発生する場合でもメモリ上で処理を行うことが可能なため，これまでのフレームワークと比較してディスクアクセスの回数を大幅に削減し，処理の高速化を実現している．特に，データの多段処理を伴うプログラムでは，Spark ではデータソースからの入力時と結果の出力時，他の計算機へのデータ転送時しかディスクアクセスが発生しないため，ディスク I/O オーバヘッドを削減できる．したがって，Spark は機械学習で見られるようなデータの繰り返し処理やリアルタイム処理を得意としており，そのような処理を実装している拡張コンポーネントが提供されている [7][8]．

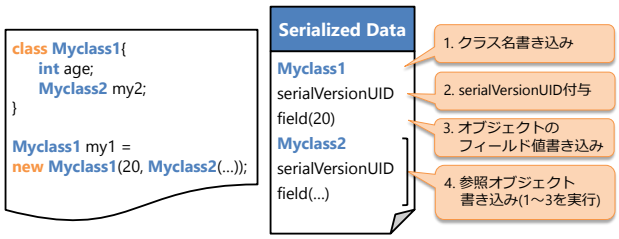

図 2　Serialize 処理の流れ

2.2 Shuffle

Spark では，RDD に対する変換を行う際に，変換後 RDD の各パーティションがそれぞれ変換元 RDD の複数パーティションを用いて生成される場合があり，この時クラスタ内計算機でデータを再分散する Shuffle 処理が行われる．Shuffle の流れを図 1 に示す．図 1 では Key 部分の”data1”と”data2”によって，それぞれ異なるパーティションにまとめてられている．Shuffle の流れは，大きく 2 つの処理に分けられる．1 つは Shuffle 前の処理によって得られた全ての中間データをストリームに書き込む Mapper 側処理，もう 1 つは Shuffle 後の処理に必要なデータを他の計算機から受信する Reducer 側処理である．

Mapper は全ての中間データを Key 毎にマージされた形式で，新しく生成したパーティションへ書き込む．全てのデータが書き込まれた後，パーティションへ書き込まれた中間データはバイト出力ストリームへ改めて書き込まれる．この時ストリームへ書き込まれるデータには Serialize 処理を行う必要がある．また，Reducer 側では，処理するデータのブロックをそれぞれ受信した後，元の形式に復元する Deserialize 処理を行う必要がある．

2.3 Serialize

Mapper はデータをストリームに書き込む際，Key と Value それぞれに対して Serialize 処理をしてから書き込む．Serialize されたデータは，データが属するクラスの情報やそのデータが参照しているデータの情報を含めたバイト列に変換される．データを Serialize する処理の流れを図 2 に示す．ここではクラス Myclass1 のオブジェクト my1 に対する Serialize を例としている．まず，クラス名である Myclass1 を書き込み，次に serialVersionUID を付与する．serialVersionUID とはオブジェクトのクラス，およびそのバージョンを識別するための番号であり，クラスを定義したときにそのクラスに対してユーザが明示する必要がある．同一クラス名であっても，クラス定義が変更された場合，変更後に作成されたオブジェクトは同一のクラスを元に作成されてはいないため，serialVersionUID を用いたクラスの識別が必要になる．最後にオブジェクトのフィールド値が書き込まれるが，この時 Myclass1 のフィールド値に参照オブジェクトが含まれる場合はその参照オブジェクトに対して同様に Serialize を行い，オブジェクトを書き込む．

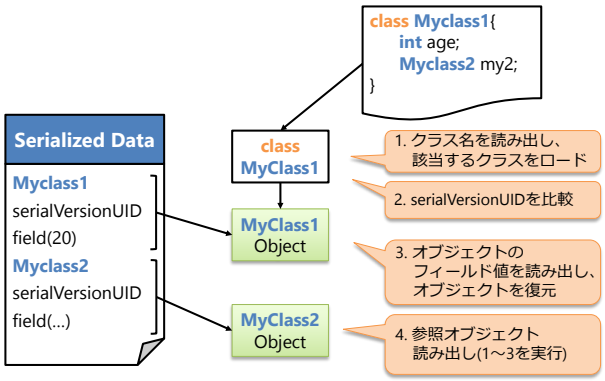

図 3　Deserialize 処理の流れ

このように Key と Value に対して Serialize を行い，Key, Value, Key, Value, ... と繰り返しストリームに書き込む．

2.4　Deserialize

データを Deserialize する処理の流れを図 3 に示す．ここでは図 2 で Serialize したオブジェクト my1 の Deserialize を例としている．まず，Mapper から受信したオブジェクトの属しているクラスを確認するために，Serialize されたオブジェクトからクラス名を読み出し，Reducer 側が持つ同一クラス名のクラス情報をロードする．次に，オブジェクトに付与されている serialVersionUID とロードしたクラスの serialVersionUID を比較する．最後にオブジェクトのフィールド値を読み出し，クラスの情報を元に復元するが，Myclass1 のフィールド値に参照オブジェクトが含まれる場合は，その参照オブジェクトに対して同様に Deserialize を行い，オブジェクトを読み出す．このように転送されたデータを復元するために，転送元と転送先の計算機で Serialize/Deserialize を通して多段な処理が実行されている．

2.5　Spark 内の Serialize/Deserialize

Spark ではデータを Serialize する方法として JavaSerializer と KryoSerializer[9] が提供されており，デフォルトでは JavaSerializer を使用する．JavaSerializer は一般的に全てのアプリケーションで使える Serializer である．また，KryoSerializer とは JavaSerializer よりも効率よく Serialize するために開発された Serializer であり，JavaSerializer よりストリームに書き込まれるデータ量が少なくなるように設計されているため，高速，高圧縮となる．しかし，KryoSerializer の性能を最大限活用するためには，ユーザは定義したクラスを KryoSerializer へ登録する必要がある．そのため，独自定義のクラスについては JavaSerializer が標準として利用されるが，Spark 2.0.0 以降では int 型や Double 型，およびその配列など基本的なクラスについては設定によらず KryoSerializer が自動的に使われる．

2.6　Spark における Shuffle 時の問題点

先に述べたように，Spark では Shuffle 時に転送データに対して転送元の計算機で Serialize 処理を行い，転送先の計算機で Deserialize 処理を行う．しかし，近年ネットワーク帯域幅は 1Gbps から 10Gbps へと増加しているが，CPU の性能向上には限界があるため，データを転送する時間は短くなっているが，Serialize/Deserialize 処理を行う時間は変化しない．そのため Shuffle 処理中のデータの Serialize/Deserialize にかかる時間の割合がより大きくなっている．Ousterhout ら [10] は，Network を通じて転送する時間を極限まで短くしたとしても全体の実行時間は最大 2%しか削減できない事を示している．この理由としてデータを転送する際にデータに対して圧縮や解凍，Serialize/Deserialize 処理を行うため，Network にかかる時間は減少するが，代わりに CPU 処理時間が増大してしまうことが挙げられている．これは，いくつかのクエリでは CPU が処理している時間の約半分がデータの圧縮，Serialize にかかる時間となっている結果からも示されている．これらの結果から Spark のデータ処理では CPU がボトルネックとなっていると結論づけられる．

Spark では Shuffle 時に転送される Key, Value のデータは全て同じ形式であるため，ストリームにデータを書き込む際に，その計算機上で Key, Value のデータ形式をあらかじめ保持しておけば，元の形式に復元できる．したがって，Serialize/Deserialize 処理を簡素化することが可能である．

3.　提案手法

本章では，Shuffle 時の転送データに対する Serialize/Deserialize を最適化し，データ転送性能を向上させる手法を提案する．この手法を実装することで，前章で述べた Serialize/Deserialize のオーバヘッドを削減し，問題の解決を図る．

3.1　概要

Spark の Shuffle 時，既存仕様では Mapper 側で中間データをストリームに書き込む際にそのデータに対して Serialize 処理を行い，Reducer 側でデータブロックを受け取った後に Deserialize を行っていた．提案手法では，KeyValue 型のデータを転送する際，Mapper 側では Serialize を簡素化して，クラス名などの省略可能な情報を付与せずバイト列変換のみを行い，Reducer 側ではデータブロックを受け取った後にデータの Value 部分のみを復元するように変更する．具体的な処理の流れを Mapper 側処理と Reducer 側処理に分けて説明する．

3.1.1　Mapper 側処理

Mapper 側では Serialize 処理を簡素化して，バイト列変換のみを行い，ストリームに書き込むように処理を変更す

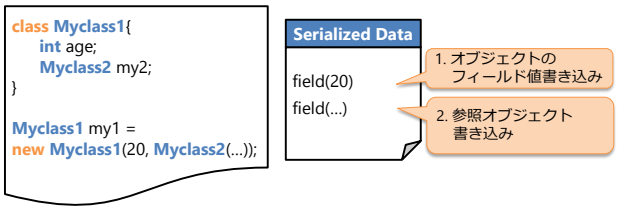

図 4　提案手法でバイト列変換する処理の流れ

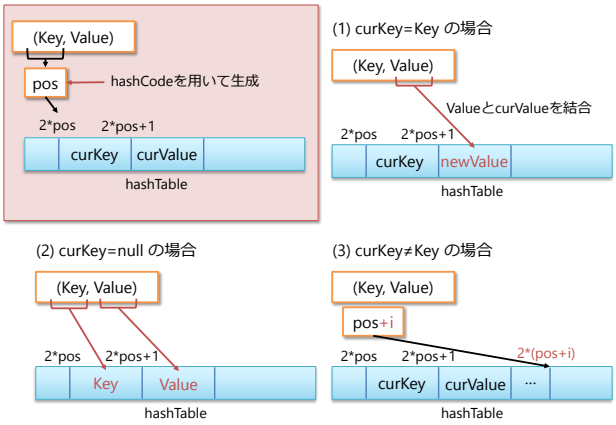

図 5　reduceByKey の動作図

る．Key と Value をバイト列変換する処理の流れを図 4 に示す．ここでは，図 2 と同様に Myclass1 のオブジェクト my1 に対するバイト列変換を例としている．提案手法では，既存の Serialize と比較して，クラス名や serialVersionUID を付与せず，オブジェクトのフィールド値のみを書き込んでいる．JVM 上では，仕様が異なる計算機に依らず復元可能にするためのヘッダをデータに付与する必要がない．また，計算機間では同じ形式のデータを送受信するため，Mapper は送信するデータのクラスの情報を，ストリームに書き込む際に記憶しておくことで，他の計算機から受け取ったデータにクラス名や serialVersionUID のようなクラスの情報が付与されていなくても，元の形式に復元することが可能である．

このように Serialize を簡素化することで，データを Serialize する処理のコストを軽減でき，データ転送時に Serialize にかかるオーバヘッドを減らすことができる．

3.1.2　Reducer 側処理

Reducer 側ではデータブロックを受け取った後，データの Value 部分については元の形式に復元するが，Key 部分についてはバイト列のままデータとして扱うように変更する．Spark では，Key を比較してから対応する Value に対する演算を行う処理が多い．例として，同じ Key をもつデータの Value を結合する関数 reduceByKey の，Reducer 側処理の動作を図 5 に示す．reduceByKey では，Mapper 側で計算機毎に同じ Key をもつデータの Value を結合し，全計算機の結果を集約するために Shuffle され，Reducer 側で結果が統合される．この時，Key 毎の Value の結合はハッシュテーブルを用いて行われ，テーブル内のデータの

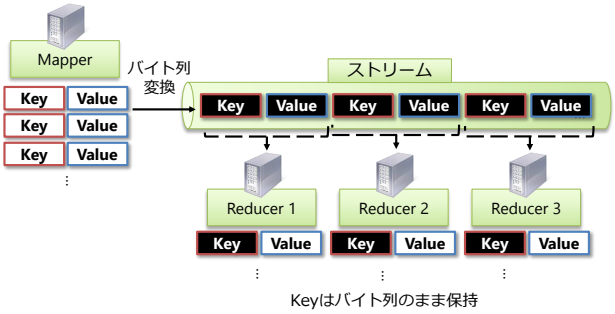

図 6　提案手法-Shuffle1 回目

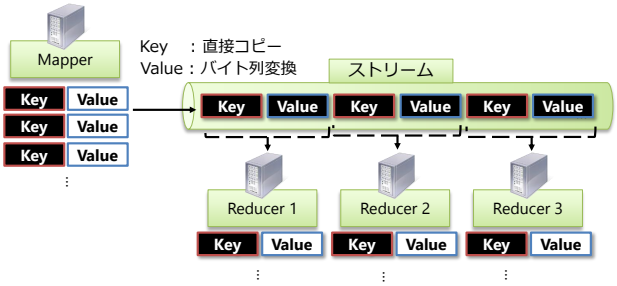

図 7　提案手法-Shuffle2 回目以降

位置は Key のハッシュコードによって決定する．位置を取得したら，その位置にあるデータの Key（図中の curKey）との比較によって，一致している場合は Value 同士を結合して値を更新，curKey に何も入っていない場合はその位置にデータを格納，一致していない場合は位置をずらす，といったように処理が分岐する．このような動作により reduceByKey が実行されるが，この時，Key はハッシュコードの生成と Key 同士の比較に使われているため，バイト列のまま実行できる．また，データを再度他の計算機へ転送する際は，前回の Shuffle 時にデータを受信した Reducer が Mapper となり，既に Key がバイト列変換されているため，Key はストリームへ直接コピーするだけで転送できる．

このようにバイト列のまま使用することで，バイト列から復元する処理のコストを軽減できる．さらに再度処理中に Shuffle が実行された時には Key のバイト列変換を行う処理を削減できる．

3.2　提案手法の利点

提案手法を用いることで，Shuffle 時の Serialize にかかるオーバヘッドを軽減できる (1 回目の Shuffle 時)．また，2 回目以降は Key の Serialize 処理自体を削減できる．図 6 に示すように，1 回目の Shuffle 時に，既存の Spark ではデータに対して Serialize を行うのに対し，提案手法ではバイト列変換のみを行うため，Serialize 処理のコストを軽減できる．また，図 7 に示すように，2 回目以降の Shuffle 時に，既存の Spark では 1 回目と同様にデータに対して Serialize を行っていたのに対し，提案手法では Key についてはデータとして保持しているバイト列をそのままスト

リームにコピーすることで Serialize 処理自体を削減できる．さらに全ての Shuffle において Value のみバイト列から復元すればよいため，Deserialize 処理のオーバヘッドも軽減できる．以上の理由から，Shuffle が複数回発生するジョブでは，既存の Spark と比較して処理速度を向上できると考えられる．k-means に代表される機械学習では，モデルを構築する際，ユーザが指定した回数分テストデータに対する学習を繰り返すか，またはモデルが収束するまで繰り返し，学習中には計算機間でデータを送受信する必要があるため，ジョブ中に複数回 Shuffle が実行される．よって提案手法は機械学習のような多段処理において，効果が大きいことが期待できる

4. 評価

提案手法の有効性を確認するため，既存仕様の Spark に追加実装を行い，ベンチマークプログラムを実行することで性能評価を行った．

表 1　評価環境

評価環境	
OS	Ubuntu 16.04
CPU	Core i7- 6900K (3.2 GHz)
メモリ	メモリ 64 GB
ネットワーク	1 Gbps
Spark のバージョン	2.0.0
クラスタ数	3 台 (Master 1 台，Slave 2 台)
Serializer	JavaSerializer, KryoSerializer
データソース	HDFS

4.1 評価環境

評価環境を表 1 に示す．クラスタは 3 台を用いて構成し，Master1 台，Slave2 台とした．これは Shuffle 時に転送元の計算機と転送先の計算機の関係を明確にするためである．ここで Slave2 台を Slave1, Slave2 とすると，このクラスタ構成では Master から Slave1, Slave2 に処理を依頼し，Slave で実際に処理を実行する．また，Shuffle 時には Slave1, Slave2 間でのみデータを送受信する．ネットワークはクラスタ内全計算機間を帯域幅 1 Gbps で接続している．

提案手法を評価するため，Serializer の異なる 2 種類の既存の Spark と，2 種類の追加実装をした Spark でベンチマークを実行し，処理時間を計測した．比較対象は以下の 4 種類である．

- 既存 (1): JavaSerializer を使用する既存の Spark
- 既存 (2): KryoSerializer を使用する既存の Spark
- 提案 (1): Serialize の簡素化のみを追加実装 （転送先で Key, Value ともにバイト列から元の形式に復元）
- 提案 (2): 提案 (1) の手法に，転送先で Value 部分のみ復元する手法を追加実装

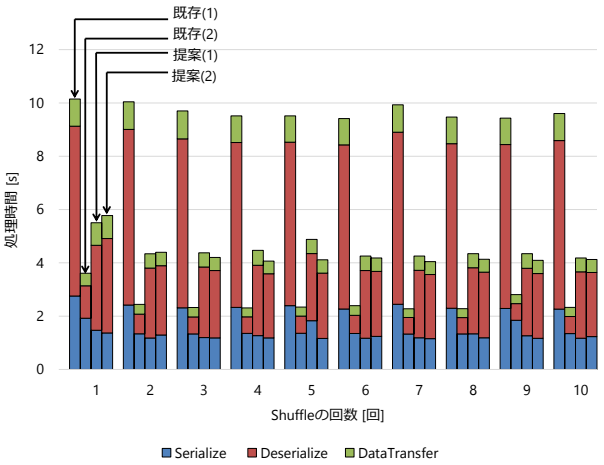

図 8　PageRank の Shuffle 回数とデータ転送時の処理時間

また，評価項目として，全体の処理時間と Shuffle 時の詳細な時間を計測した．既存手法では Serialize をしてからストリームに書き込む時間，受け取ったデータブロックを Deserialize する時間，データ転送にかかる時間の 3 つを計測し，提案手法ではバイト列変換してからストリームに書き込む時間，受け取ったデータを元のデータ形式に復元（またはそのまま保持）する時間，データ転送にかかる時間の 3 つを計測した．評価結果ではそれぞれの計測時間を Serialize, Deserialize, DataTransfer と表記する．

表 2　ベンチマークの設定

プログラム	設定
PageRank	ページ数 500000
k-means	サンプル数 100 万 サンプルの次元数 20 k=10000

ベンチマークプログラムとして，Intel が提供している HiBench[11] を利用した．ベンチマークの設定を表 2 に示す．また，PageRank, k-means 共に学習の繰り返し回数を 10 回と指定した．評価結果は，ベンチマークをそれぞれ 10 回試行した際の平均を示す．

4.2 評価結果

Pagerank の Serialize 関連の評価を図 8 に示す．図中のグラフは横軸が Shuffle の回数，縦軸がそれぞれの処理にかかる時間を秒単位で表す．また，それぞれの Shuffle 回数において，左から既存 (1)，既存 (2)，提案 (1)，提案 (2) を示す．このグラフから，JavaSerializer を使用した場合と比較すると，提案 (1) では Serialize にかかる時間を 45.0%，Deserialize にかかる時間を 58.4%，データ転送にかかる時間を 43.8%削減した．また，提案 (2) ではそれぞれ 48.8%，59.2%，47.7%削減した．しかし，KryoSerializer を使用した場合と比較すると，提案 (1) では Serialize にかかる時間

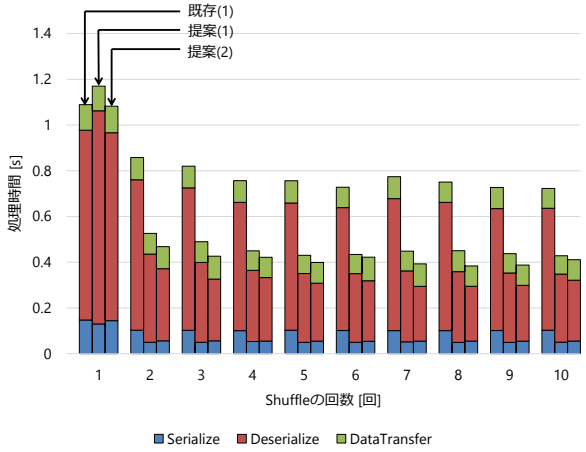

図 9　k-means の Shuffle 回数とデータ転送時の処理時間

は 9.9%削減したが，Deserialize にかかる時間は 272%増加し，データ転送にかかる時間は 59%増加した．提案 (2) ではそれぞれ 17.1%削減，264%増加，48%増加となった．全体の処理時間は既存 (1) が 58.6 秒, 既存 (2) が 48.2 秒, 提案 (1) が 47.7 秒, 提案 (2) が 47.0 秒となり，JavaSerializer と比較すると提案手法は処理時間が削減できているが，KryoSerializer との処理時間差は誤差の範囲内であると考えられる．

次に，k-means の Serialize 関連の評価を図 9 に示す．Spark mllib で実装されている k-means では，KryoSerializer がデフォルトで使用されないため，既存 (2) は省略している．このグラフから，JavaSerializer を使用した場合と比較すると，提案 (1) では Serialize にかかる時間を 45.0%, Deserialize にかかる時間を 37.4%, データ転送にかかる時間を 7.5%削減した．また，提案 (2) ではそれぞれ 39.6%削減，46.6%削減，0.1%増加となった．全体の処理時間は既存 (1) が 750 秒, 提案 (1) が 771 秒, 提案 (2) が 754 秒となった．処理時間の削減が確認できない理由は，k-means が CPU インテンシブなワークロードであるため，Shuffle 時の Serialize/Deserialize にかかる時間を削減しても，その他の処理による処理時間への影響が大きく，変化があまり見られなかったからである．

4.3　考察

評価結果から，データインテンシブなワークロードである PageRank では，Serialize/Deserialize 処理にかかる時間と全体の処理時間共に，JavaSerializer に対する提案手法の有効性を確認できたが，KryoSerializer との比較では性能が低下した．これは，2.5 で述べたように，KryoSerializer によってストリームに書き込まれるデータ量が少ないため，提案手法よりも Serialize/Deserialize にかかる時間が短くなったと考えられる．しかし，KryoSerializer は独自定義したクラスについてはデフォルトで使用できないため，提案手法は KryoSerializer が使用されないような，独自定義のクラスにおいて有効である．

また，Serialize の時間に限れば KryoSerializer に対する提案手法の有効性を確認できた．これは Serialize の簡素化が有効に働いたためである．よって，Value 部分も含めた Deserialize の省略や，ストリームに書き込まれるデータ量の削減により Deserialize の時間を削減できれば，KryoSerializer よりも性能が向上し，汎用性も高い手法になると考えられる．

5.　関連研究

ZeroFormatter[12] は C#専用の Serialize フォーマットである．ZeroFormatter では Serialize して転送したデータを，Deserialize せずにそのままバイト列として保持しておき，使用するときに Deserialize する．Deserialize を遅延することで，転送先計算機で受信したデータを部分的にしか使わない場合は Deserialize にかかる時間を削減できる．また，データの値を変更する場合は，固定長の値を変更する時はその値をバイト列に直接書き込み，可変長の値を変更する時は変更箇所以外の値を新しいバイト列へコピーする．変更する値を Serialize して該当する箇所に書き込むことで，Deserialize せずに値を変更できる．このように Serialize したバイト列をそのまま保持しておく事で再転送する際の Serialize 処理を省略できる．しかし，転送した全てのデータに対して何らかの処理が実行される場合は Deserialize の遅延は有効ではない．また，ZeroFormatter は C#しか対応していないため Spark では使用できない．

6.　まとめ

本稿では，Spark の Shuffle 時における Serialize/Deserialize 処理を最適化することで，データ転送性能を向上させる手法を提案した．提案手法では，データを送信する際の Serialize を簡素化して，必要のない情報を付与せずバイト列変換のみを行い，データを受信する際の Deserialize 処理を簡素化して，データの Key 部分についてはバイト列のままデータとして扱うように処理を変更した．提案手法の有効性を確認するために，Serialize の簡素化と Key の Deserialize を省略する手法を実装し，ベンチマークを実行した．評価結果から，データインテンシブなワークロードである PageRank では，JavaSerializer と比較すると，Serialize/Deserialize にかかる時間を 57.3%削減，全体の処理時間を 19.8%削減した．しかし，KryoSerializer と比較すると，Serialize/Deserialize にかかる時間は 75.7%増加してしまった．以上の結果から，KryoSerializer が使用されない場面においては，提案手法が有効であると考えられる．

今後の課題として，独自定義のクラスにおける提案手法の有効性の評価，および Value 部分も含めた Deserialize の省略や，ストリームに書き込まれるデータ量の削減による

Deserialize の時間の削減が挙げられる．

謝辞 本研究の一部は，科研費基盤研究 (C) 24500113, (C) 15K00168 による．

参考文献

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and AH. Byers. Big data: The next frontier for innovation, competition, and productivity. A report by the McKinsey Global Institute, May 2011. `https://bigdatawg.nist.gov/pdf/MGI_big_data_full_report.pdf`.

[2] Matei Zaharia. Spark: In-Memory Cluster Computing for Iterative and Interactive Applications. In *Invited Talk. NIPS Big Learning Workshop: Algorithms, Systems, and Tools for Learning at Scale (Granada, Spain.)*, December 12–17, 2011.

[3] Apache Hadoop. `http://hadoop.apache.org/`.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX conference on Networked Systems Design and Implementation (California, USA.)*, p. 2, April 2012.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (Nevada, USA.)*, pp. 1–10, May 2010.

[6] Avinash Lakshman and Prashant Malik. Cassandra-a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, pp. 35–40, 2010.

[7] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, Vol. 17, No. 34, pp. 1–7, 2016.

[8] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles (Pennsylvania, USA)*, pp. 423–438, Nobember 2013.

[9] Kryo. `https://github.com/EsotericSoftware/kryo`.

[10] K. Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (California, USA.)*, pp. 293–307, March 2015.

[11] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

[12] ZeroFormatter. `https://github.com/neuecc/ZeroFormatter`.