Vol.2020-OS-150 No.8
2020/7/30

Apache Sparkにおける再計算の暗黙的な省略を考慮した性能予測

井上 達博[1,a)] 置田 真生[1,b)] 伊野 文彦[1,c)]

概要：本研究では，Spark プログラムを対象に，依存グラフ解析に基づく実行時間推定の改善手法を提案する．これまでに，実行時間推定を利用して Spark プログラムのメモリ内キャッシュ指示を最適化する手法がいくつか提案されている．しかし，これらの手法は，ノード間通信に伴うキャッシュが発生する場合に実行時間の推定に失敗し，有用性の低いキャッシュ指示の組み合わせを選択する可能性がある．本報告では，計算結果の暗黙的な再利用を再現しながら実行時間推定を改善する手法を提案する．実験の結果，従来手法と比較して，実行時間の相対誤差の平均値を 311 から 131 に改善した．提案手法を既存のキャッシュ指示選択手法と組み合わせることで，ノード間通信を含む Spark プログラムを既存手法と組み合わせた場合と比較して最大 1.1 倍高速化できた．提案手法を用いることで，依存グラフ解析に基づく Spark プログラムの性能解析の改善を期待できる．

1. はじめに

Apache Spark[1], [2]（以降，Spark）は大規模なデータ処理向けの並列分散処理基盤である．プログラマはデータ集合に対する変換操作を頂点とした依存グラフを定義することで処理を記述する．具体的には，データ集合を Resilient Distribution Dataset[3]（以降，RDD）というデータ構造を用いて表現し，RDD に対する変換操作を呼び出すことで，RDD とその操作の依存関係を定義する．Spark は RDD に対する変換操作を自動的に並列処理する．したがって，プログラマは複雑な並列処理を記述することなく，大規模なデータを簡単に処理できる [4], [5], [6].

Spark プログラムの高速化には，ソフトウェアキャッシュを用いて再計算を回避することが重要である [7]．大規模なデータを処理する Spark は，必要なデータのみをメモリ上に保持するために，RDD の変換結果を計算が完了した時点で破棄する．そのため，同じ RDD を再度利用する場合はその度に再計算が必要である．再計算を回避するために，明示的に RDD にキャッシュ指示を与えて RDD の変換結果をメモリ内のキャッシュ領域（IMC）に格納し再利用する機能を Spark は提供する．さらに，Spark は暗黙的なキャッシュ機能を備える．計算ノード間多対多通信（以降，シャッフル [8], [9]）のたびに，Spark は RDD 変換の中間結果（以降，シャッフルファイル）を自動的にディスクへ保存する．明示的なメモリ内キャッシュと同様に，シャッフルファイルを再利用することで RDD の再計算を回避する．

一般に，最適なキャッシュ指示の決定は難しい．IMC の容量は有限であるので，過剰な RDD にキャッシュ指示を与えると IMC に格納した RDD の置換が発生し，再計算が必要となる．したがって，キャッシュ指示を与える RDD の取捨選択が必要である．キャッシュすべき RDD は（1）再計算に要する時間が長いもの,（2）利用回数が多いもの,（3）実体データが小さいものである．しかし,（1),（2）および（3）のいずれも実行前の見積りが難しい．さらに,（1）および（2）は，キャッシュ指示を与えられた他の RDD との組み合わせに依存して評価値が変化する．したがって，プログラマは試行錯誤を重ねて，プログラム全体における再計算の合計時間が最小となるキャッシュ指示の組み合わせを決定する必要がある．

有用なキャッシュ指示の選択手法として，RDD の依存グラフ解析に基づく方法がある [10], [11], [12]．Yang ら [10] は事前に想定した依存グラフのパターンを対象に，プログラム実行中にキャッシュ指示を適応的に決定する手法を提案した．Nasu ら [11] は，プログラムの事後解析に基づいて，2 回目以降の実行を高速化するためのキャッシュ指示を決定する手法を提案した．これらの方法は，RDD の依

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
a) t-inoue@ist.osaka-u.ac.jp
b) okita@ist.osaka-u.ac.jp
c) ino@ist.osaka-u.ac.jp

ⓒ 2020 Information Processing Society of Japan

存グラフ解析による実行時間の予測に基づく．すなわち，依存グラフを辿ってキャッシュ指示を変更した場合のプログラムの実行を再現し，予測実行時間を最小化するキャッシュ指示の組み合せを選択する．

しかし，既存のキャッシュ指示選択手法が用いる実行時間の予測手法は，シャッフルに伴う暗黙的なキャッシュのふるまいを再現しない．この方法は，シャッフルが発生する場合に本来の実行と比較して再計算の回数を多く推定し，予測実行時間が増加する．誤った予測実行時間を基にキャッシュ指示を選択するので，シャッフルを含むプログラムを対象にした場合，有用性の低いキャッシュ指示の組み合わせを選択する可能性がある．

そこで本報告では，シャッフルを含むプログラムを対象に，キャッシュ指示を変更した場合の実行時間を正確に予測する手法を提案する．具体的には，RDD の依存グラフを辿ってプログラムの実行を再現する際，シャッフルに伴う暗黙的なキャッシュの存在を推定する．存在する可能性が高い場合には，中間結果を再利用すると見なして実行時間を予測する．Spark において，シャッフルの発生およびシャッフルファイルの生存期間は，入力データおよびディスク容量に依存して動的に決まる．提案手法は，Nasu らの手法 [11] と同様に小規模な事前実行に基づいてシャッフルの発生を判定し，ディスク容量が十分に大きいと仮定してシャッフルファイルの生存期間を推定する．

提案手法を用いることで，RDD の依存グラフ解析に基づく Spark プログラムの性能解析手法を改善できる．例えば，キャッシュ指示の選択手法 [10], [11] と提案手法を組み合わせることで，プログラムがシャッフルを含む場合に，実行時間をより削減できるキャッシュ指示の選択を期待できる．

以降，まず 2 節で関連研究を紹介し，3 節で Spark の概要を示す．次に， 4 節で提案手法について説明し，5 節で評価実験を示す．最後に 6 節で本報告をまとめる．

2. 関連研究

Spark プログラムの実行時間予測に基づいてキャッシュ指示を与えるべき RDD を決定する研究がある．Gottin ら [12] は RDD の依存グラフ，各 RDD 操作に要する時間および IMC の読み書きに要する時間を入力として，適切なキャッシュ指示を決定するアルゴリズム S-CACHE を提案した．S-CACHE では RDD をキャッシュした場合に削減できる実行時間を予測し，削減量が大きい RDD を優先的にキャッシュする．S-CACHE では RDD にキャッシュ指示を与えた場合の再計算の省略を再現する一方，シャッフルによる再計算の省略は再現しない．したがって，Spark プログラムがシャッフルを含む場合では，より適切なキャッシュ指示が存在する可能性がある．

Nasu ら [11] は 1 度プログラムを実行して RDD の依存グラフを取得し，出次数が 2 以上のすべての頂点にキャッシュ指示を与えた．また，キャッシュする効果の高い RDD の計算結果を優先的に IMC に残すキャッシュ置換アルゴリズム MCR を提案した．MCR はデータサイズが小さく，キャッシュした場合に削減できる実行時間の予測値が大きい RDD を優先的に IMC に残す．実行時間の予測では S-CACHE と同様にキャッシュ指示による再計算の省略は再現する一方，シャッフルによる再計算の省略は再現しない．

Yang ら [10] は事前に想定した依存グラフのパターンを対象に，プログラム実行中にキャッシュ指示を適応的に決定する手法を提案した．十分にプログラム実行時間が長い場合，Yang らの手法を用いて得たキャッシュ指示が削減する実行時間は最適解の $(1-1/e)$ 倍以上である．しかし，Spark が再計算を省略する要因をキャッシュ指示のみに限定している．したがって，Spark プログラムがシャッフルを含む場合はキャッシュ指示が削減する時間を保証しない．

3. Apache Spark

Spark はマスタ・ワーカ型の並列分散処理フレームワークである．プログラマはマスタノード上で動作するドライバプログラムのみを作成する．RDD に対する操作のワーカノード上の分散実行，すなわちデータの分散，タスクの分割，タスクのスケジューリングおよびノード間通信は Spark が暗黙的に実行する．

3.1 Spark の実行モデル

ドライバ・プログラムは主に RDD 操作の連なりからなる．RDD 操作は変換とアクションの 2 種類に区別できる．**図 1** はファイルの中に出現する単語数を数える Spark プログラムを示す．Spark は変換操作呼び出し時，計算の実行を保留し依存関係のみを記憶する．アクション呼び出し時，依存関係を辿って必要なすべての操作を実行する．1 つのアクション呼び出しに起因する一連の計算をジョブと呼ぶ．各操作の実行において，Spark はデータ交換のためのノード間通信（シャッフル）を必要に応じて行う．**図 2** は図 1 のプログラムから抽出した RDD 操作の依存グラフを示す．

3.1.1 グラフ定義

Spark プログラムの実行は，RDD 操作を頂点とする有向非循環グラフ $G=(V,E,A)$，性能情報 (w,D) およびジョブの列 J で表現できる．各要素はそれぞれ，頂点集合 V，辺集合 E，アクションの集合 $A \subseteq V$，計算コスト $w: V \to \mathbb{R}$，およびデータ量 $D: V \to \mathbb{R}$ を表す．頂点は RDD 操作の実行インスタンスである．すなわち，同一ソース行を複数回実行する場合は実行ごとに対応する頂点が存在する．辺 $(u,v) \in E$ は頂点間のデータ依存関係を表し，RDD 操作 v の入力として RDD 操作 u の計算結果が必要

```
1   val sc = new SparkContext()
2   val input = sc.textFile("textData")  // 変換
3     .flatMap(line => line.split(" ")) // 変換
4     .map(word => (word, 1))           // 変換
5     .reduceByKey(_ + _)               // 変換
6     .cache()                          // キャッシュ指示
7   val numwords = input.collect()      // アクション
8
9   for (i <- 1 to 3) {
10    val rare = input.filter(_._2 == i) // 変換
11    val filwords = rare.collect()      // アクション
12  }
```

図 1 Scala で記述した Spark プログラムの例（青色および赤色はそれぞれ変換およびアクションを表す）

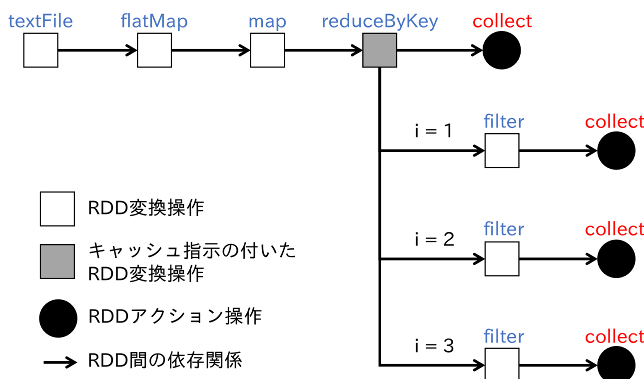

図 2 図 1 のプログラムから抽出した RDD 操作の依存グラフ

であることを意味する．以降では $v \in V$ の計算結果を $\hat{v}$ で表現する．アクションは処理の終端であるため，出次数が 0 の頂点かつそれらのみがアクションである．計算コスト $w(v)$ は v の計算に要する時間を表し，データ量 $D(v)$ は $\hat{v}$ の大きさを表す．また，ジョブの系列 $J = (a_1, a_2, .., a_{|A|})$ ($a_i \in A$) はアクションを実行順に並べた列である．

ここでは，RDD と中間結果とを区別している．RDD は RDD 操作を仲介する仮想概念であり，中間結果は処理およびキャッシュされる実質的なデータである．

i 番目のジョブの実行は，a_i を始点に G を逆探索することで再現できる．**アルゴリズム 1** は，ジョブの実行手順を簡略化した手続きである．ここで，$P(v)$ は頂点 v の処理に必要な直前の RDD 操作の集合を表し，$P(v) = \{u \in V \mid (u, v) \in E\}$ である．v から入次辺を逆向きに辿り，入次数が 0 である頂点あるいは IMC 上に結果が存在する頂点に到達するまで深さ優先順で再帰的に繰り返す．実際の Spark は探索後に複数の RDD 操作を一括して実行するが，ここでは簡単のために逐次実行を前提とした．

IMC に中間結果を格納する目的で，プログラマは RDD 操作に対して `cache()` や `persist()` を呼び出して明示的にキャッシュ指示を与える必要がある．v にキャッシュ指示があることを $\gamma(v)$ で表現する．Spark は，もし $\gamma(v)$ が

アルゴリズム 1 ジョブの実行手続き X

Input:
　a: 実行する RDD 操作
Output:
　$\hat{a}$: a の計算結果
Global variables:
　C: IMC 上に保持する計算結果の集合
1: $Q \leftarrow \emptyset$; ▷ a の計算に必要なデータ
2: **for all** $v \in P(a)$ **do**
3: 　**if** $\hat{v} \in C$ **then**
4: 　　$Q \leftarrow Q + \{\hat{v}\}$
5: 　**else**
6: 　　$Q \leftarrow Q + \{X(v)\}$
7: $\hat{a} \leftarrow$ (Q を入力として a を実行)
8: **if** $\gamma(a)$ **then**
9: 　$U(C, \hat{a})$ ▷ IMC を更新
10: **return** $\hat{a}$

真であれば $\hat{v}$ を IMC に格納する．アルゴリズム 1 に示すように，$\hat{v}$ を IMC に格納することで v の再計算を回避できる．加えて，Spark は容量の制約が原因で中間結果のすべてを IMC に格納できない場合は一部分のみを格納する．

Spark は IMC の置換アルゴリズムとして LRU を採用している．アルゴリズム 1 において，$U(C, \hat{v})$ は $\hat{v}$ を C に格納して IMC を更新する手続きを表す．$\hat{v}$ を加える前に，Spark は IMC の空き容量を調べる．空き容量が不足している場合，$\hat{v}$ の容量を確保する目的で，Spark は LRU に基づいて中間結果を IMC から削除する．

3.1.2 キャッシュ指示の利得予測

プログラムの高速化を目的として，IMC の置換アルゴリズムを改善する様々な研究がある [10], [11], [13]．多くの研究では，中間結果 $\hat{v}$ を C に追加した場合の利得 $\epsilon(v, C)$ が大きい $\hat{v}$ を IMC に残す手法を提案している．$\epsilon(\hat{v}, C)$ にはプログラムの推定実行時間を使用することが多い．したがって，実行時間の予測正確度を改善すると多くの既存手法の改善が期待できる．

Nasu らの手法 [11] では，キャッシュ指示の利得 $\epsilon(v, C)$ を式 (1) で定義する．

$$\epsilon(v, C) = R(v, C)/D(v) \tag{1}$$

ここで，$R(v, C)$ は IMC C に $\hat{v} \notin C$ を追加した場合に削減できる推定実行時間を表す．2 つの $u, v \in V$ のキャッシュ利得 $\epsilon(u, C)$ および $\epsilon(v, C)$ の大小は，$D(u) = D(v)$ の場合は削減する時間の大きさによって決まり，$R(u, C) = R(v, C)$ の場合は中間結果の大きさによって決まるので合理的である．

$R(v, C)$ の推定方法を式 (2) に示す．

$$R(v, C) = \sum_{a \in J} T(a, C) - \sum_{a \in J} T(a, C + \{\hat{v}\}) \tag{2}$$

ここで，$T(a, C)$ は IMC C のもとで RDD 操作 a の実行に要する時間を表す．再計算以外の要因で実行時間が変化し

アルゴリズム 2 実行時間を予測する手続き T

Input:
a: 実行する RDD 操作
C: IMC 上に保持する計算結果の集合

Output:
t: $\hat{a}$ を得るために要する時間

```
1: t ← 0;
2: for all v ∈ P(a) do
3:     if v̂ ∉ C then
4:         t ← t + w(v) + T(v, C)
5: return t
```

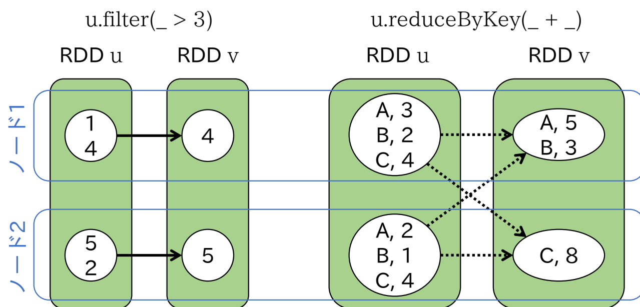

図 3 狭い依存関係（左）とシャッフル依存関係（右）の例

ないと仮定すると，$R(v, C)$ は式 (2) のように実行時間の差分で推定できる．

IMC C のもとで RDD 操作 a の実行に要する時間 $T(a, C)$ を予測する手続きを**アルゴリズム 2** に示す．アルゴリズム 1 より a の依存関係を再帰的に辿ることで a の実行時間を推定する．IMC 内に $\hat{v}$ が存在する RDD v の実行時間は 0 とする．

3.2 シャッフル

Spark は RDD 操作の依存関係 $(u, v) \in E$ を狭い依存関係とシャッフル依存関係の 2 種類に区別する [14]．2 つの依存関係の例を**図 3** に示す．狭い依存関係は計算ノード内で計算が完結する依存関係である．例えば，条件に合う要素のみを抽出する `filter()` は狭い依存関係を形成する．一方，シャッフル依存関係は計算ノード間で中間結果を共有する必要のある依存関係である．例えば，キーごとに値を集約する `reduceByKey()` はシャッフル依存関係を形成する．Spark はシャッフルと呼ばれる通信処理によってノード間で中間結果を共有する．シャッフル依存関係 (u, v) におけるシャッフルを $S(u, v)$ とする．

Spark は次の（s1）から（s4）の一連の処理を実行することで $S(u, v)$ を実現する [8], [9], [14]．（s1）各計算ノードで $\hat{u}$ に変換 v の一部分 v' を実行する．v' は v の内，計算ノード内で完結する変換操作であり，通信するデータ量を削減する目的で v' を実行する．（s2）計算ノードのディスク上に（s1）の変換結果 $F(u, v)$ を格納する．$F(u, v)$ をシャッフルファイルと呼ぶ．（s3）計算ノードは $F(u, v)$ を他の計算ノードから受信する．（s4）$F(u, v)$ に対して v' 以外の変換 v を実行する．

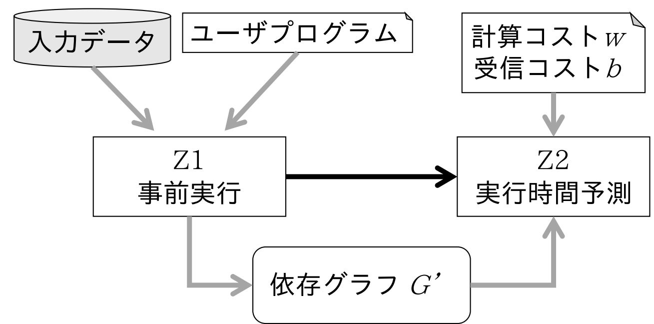

図 4 提案手法における処理の流れ

Spark は $S(u, v)$ を再度実行する際，ディスク上に $F(u, v)$ が存在する場合は（s1）および（s2）を省略する．3.1.2 節で挙げた手法 [10], [11], [13] は $F(u, v)$ が存在しないことを前提としている．したがって，$F(u, v)$ の再利用による（s1）および（s2）の省略を再現することで実行時間の予測正確度を改善できる．$F(u, v)$ の再利用を再現するためには，依存関係 (u, v) がシャッフル依存関係である条件の特定および $F(u, v)$ の生存期間の再現が必要である．

依存関係 (u, v) がシャッフル依存関係である条件は v によって異なる．例えば，v が `filter()` であれば (u, v) は必ず狭い依存関係であり，v が `reduceByKey()` であれば (u, v) は必ずシャッフル依存関係である．RDD 変換操作の中にはどちらの依存関係にもなり得る変換が存在する．変換 `cogroup()` は RDD の中間データの分配方法が u と v で同じであれば (u, v) は狭い依存関係，異なればシャッフル依存関係となる．中間データの分配方法はユーザが動的に変更できるので，(u, v) がシャッフル依存関係であるかどうかは実行前に確定しない．

$F(u, v)$ が作成されるのは，Spark が $S(u, v)$ の（s2）を実行するときである．$F(u, v)$ が削除されるのは，Spark が $F(u, v)$ を不要と判断したとき，ユーザが明示的に削除を指示したときおよびエラーが発生したときである．Spark が不要と判断するのは，(u, v) を参照する RDD がガベージコレクションされたときおよび Spark アプリケーションの終了処理時である．

4. 提案手法

実行時間の予測においてシャッフルに伴う中間結果の再利用を再現する方法を提案する．本手法は Nasu らの手法 [11] と同様に（Z1）小規模な事前実行および（Z2）実行時間の解析部で構成される（**図 4**）．（Z1）は，動的に決まるシャッフルの発生を検出することを目的に，小規模な入力データをプログラムに与えて 1 度実行する．（Z2）は，シャッフルファイルが存在した場合の Spark の実行を再現しながら実行時間を予測する．

Spark が暗黙的に再計算を省略するのはシャッフル

アルゴリズム 3 実行時間を予測する手続き T'

Input:
a: 実行する RDD 操作
C: IMC 上に保持する計算結果の集合
Output:
t: $\hat{a}$ を得るために要する時間
Global variables:
$H \subseteq E'$: 一度辿ったシャッフル依存関係の集合

```
1: t ← 0;
2: for all v ∈ P(a) do
3:     if v̂ ∉ C then
4:         if (v, a) ∈ H then
5:             t ← t + b(v, a)
6:         else
7:             t ← t + w(v) + T'(v, C,)
8:             if (v, a) ∈ E' then
9:                 H ← H + {(v, a)}
10: return t
```

$S(u,v)$ を実行する際，シャッフルファイル $F(u,v)$ がディスク上に存在している場合である．本手法では，ディスク容量が十分に大きいと仮定する．すなわち，2 回目以降の $S(u,v)$ 実行時に $F(u,v)$ がディスク上に存在すると仮定する．$F(u,v)$ を削除すると再計算が発生するのでユーザが明示的に $F(u,v)$ を削除する場合は限定的であり，ディスク容量不足以外の原因によるエラーの発生を予見するのは困難であるからこの仮定は妥当であると考える．

4.1 事前実行でシャッフルを検出する方法

シャッフルは RDD 間でシャッフル依存関係がある場合に発生する．依存関係がシャッフル依存関係かどうかは動的に決定する場合がある．そこで本手法では，（Z1）の実行時情報に基づいて依存関係がシャッフル依存関係であるか判定する．Spark では狭い依存関係およびシャッフル依存関係に対応するクラスを定義しており，RDD 間の依存関係をどちらかのクラスのインスタンス i で表現している．したがって，i がシャッフル依存関係を表すクラスのインスタンスかどうかで判定する．

4.2 グラフの拡張

Spark をモデル化したグラフ G では，依存関係がシャッフル依存関係かどうか判断できない．本手法ではシャッフル依存関係の集合 $E' \subseteq E$ を G に加えた $G' = (V, E, A, E')$ を定義する．（Z1）で G' を作成することで（Z2）においてシャッフル依存関係を判別可能にする．

4.3 実行時間の予測方法

本手法では，アルゴリズム 2 に示した Nasu らの手法 [11] における実行時間を予測する手続き $T(a,C)$ を**アルゴリズム 3** に示す $T'(a,C)$ に拡張する．ここで，$b(v,a)$ は他の計算ノードから $F(v,a)$ を受信するのに要する時間を表す．$F(v,a)$ が存在する場合（$(v,a) \in H$），$\hat{v}$ の計算時間 $T'(v,C)$ の代わりに $b(v,a)$ を予測実行時間に加算することで中間結果の暗黙的な再利用を再現する．

表 1 実験 1 の環境

CPU	Intel Xeon E5-2650 v3 2.30 GHz
主記憶	4 GB （Spark に指定した値）
ヒープ	2.1 GB
コア数	1 個
OS	CentOS 7.6.1810
Spark	v2.4.4（local モード）

表 2 実験 2 および 実験 3 の環境（PC クラスタ）

ノード数	8 台
CPU	Intel Xeon E5-2650 v3 2.30 GHz
ネットワーク	Gigabit Ethernet，スター型
主記憶	4 GB（ノードあたり，Spark に指定した値）
ヒープ	2.1 GB (ノードあたり)
コア数	1 個（ノードあたり）
OS	CentOS 7.6.1810
Spark	v2.4.4（Standalone モード）

5. 評価

実際の Spark プログラムを用いて，次の 3 点について提案手法を評価する．
（実験 1）実行時間予測に要する時間
（実験 2）実行時間予測の正確度
（実験 3）提案手法の応用に対する実用性
実験 3 では，提案手法を適用した予測実行時間に基づく IMC の置換アルゴリズム [11] の有用性を評価して間接的に提案手法の有用性を評価する．IMC の置換アルゴリズム [11]（以降，MCR）では式 (1) で定義したキャッシュ指示の利得 $\epsilon(v,C)$ が大きい $\hat{v}$ を優先して IMC に残す．キャッシュ指示の利得の推定がより正確であるほど Spark プログラムの実行時間は小さくなる．

5.1 実験方法

表 1 および**表 2** に実験環境を記す．実験 2 および実験 3 では 8 台の計算機で構成する PC クラスタを用いた．ノードあたりの IMC 容量は，キャッシュの置換を誘発するために，比較的小さい値に設定した．また，実験の単純化のためにコア数は 1 に設定した．

対象プログラムは Spark が提供する機械学習ライブラリ群 MLlib [15] の MovieLensALS を用いた．MovieLensALS における RDD 依存グラフの概要を**図 5** に示す．実験 2 ではシャッフルの再現が予測時間に与える影響を調べる目的で，IMC による再計算の省略を排除するためにプログラム中のキャッシュ指示をすべて無効にした．また，入力データは MovieLens [16] から得た．

（Z2）で指定した計算コスト w を**表 3** に示す．既存手

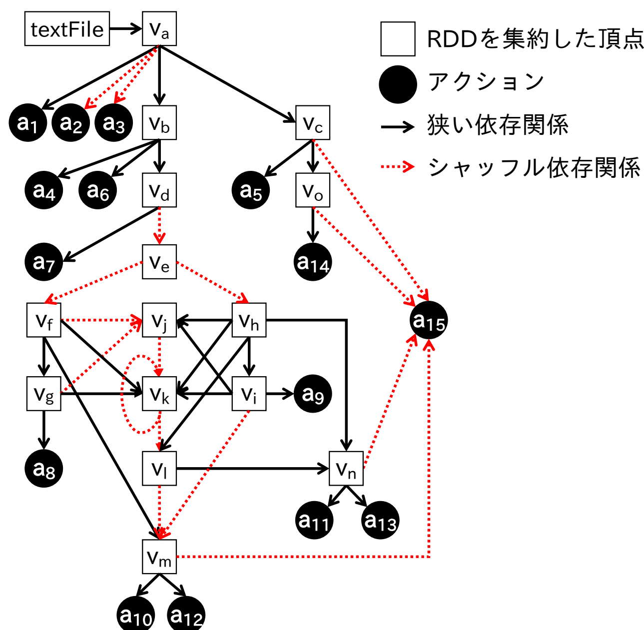

図 5 MovieLensALS における依存グラフの概要

法 [11] と実験結果を比較する関係で，Nasu らが評価実験に使用した計算コストと同一にした．

5.2 事前実行・キャッシュ優先度解析の処理時間

既存手法 [11] および提案手法において，小規模な事前実行およびキャッシュ優先度解析に要する実行時間を**表 4** に示す．ただし，入力データセットの先頭 0.15 GB を入力して 11 回実行し，それぞれの実行時間の中央値を取った．キャッシュ優先度解析とは，MCR においてキャッシュ指示を与えたすべての RDD v のキャッシュ指示の利得 $\epsilon(v, C)$ を推定する処理である．提案手法は既存手法と比較してキャッシュ優先度解析の実行時間を約 8 倍高速化できた．高速化の原因は，実行時間の予測においてシャッフルファイルの存在を推定した場合にグラフの依存関係を辿る計算をやめる枝刈りにあると考えられる．一方で提案手法を用いると事前実行の処理時間は 0.8%増加した．事前実行における処理時間増加の原因は，RDD 間の依存関係がシャッフル依存関係かどうか判定する処理およびグラフにシャッフル依存関係の情報 E' を加える処理を追加したからであると考えられる．

5.3 実行時間予測の正確度

実際に計測した実行時間と既存手法および提案手法で予測した実行時間を基に予測の正確さを 2 つの指標で評価した（**表 5**）．ただし，実際の実行時間はプログラムに 5.7 GB のデータセットを入力して 5 回計測し，中央値を取った．5.7 GB のデータセットは入力データセット [16] の内容を複製して作成したものである．今回は簡単のためにシャッフルファイルの受信に要するコスト b をシャッフル依存関係に関わらず一定値に指定した．

表 3 RDD 操作 v の計算コスト $w(v)$

v	textFile	map	reduceBykey	groupByKey	others
$w(v)$	47	2	18.25	18.25	1

表 4 事前実行およびキャッシュ優先度解析の実行時間

	既存手法	提案手法
事前実行（s）	34.4	34.7
キャッシュ優先度解析（ms）	8552	997

表 5 予測実行時間の正確度

	既存手法	提案手法 ($b = 0$)	提案手法 ($b = 5$)	提案手法 ($b = 10$)
決定係数	-62.821	0.844	0.910	0.957
MRE	311.546	131.057	131.294	131.530

計算コスト w および通信コスト b は map() 操作に要する時間 m を 2 とした相対値である．したがって，ジョブ a_i の予測実行時間 $T(a_i, \emptyset)$ および $T'(a_i, \emptyset)$ と実際の a_i の実行時間 t_i には式 (3) のような関係がある．

$$t_i \approx \frac{m}{2} T(a_i, \emptyset) \approx \frac{m}{2} T'(a_i, \emptyset) \qquad (3)$$

本実験ではシャッフル通信を含まないジョブ a_1 において実行時間の予測に誤差はないと仮定し，ジョブ a_i（$i \neq 1$）における予測実行時間の尺度を実際の実行時間の尺度に合わせた．すなわち，$t_1 = \frac{m}{2} T(a_1, \emptyset)$ および $t_1 = \frac{m}{2} T'(a_1, \emptyset)$ で定数 m を決定し，a_i（$i \neq 1$）における予測実行時間 $\frac{m}{2} T(a_i, \emptyset)$ および $\frac{m}{2} T'(a_i, \emptyset)$ と t_i の比較によって予測正確度を評価した．

各手法による予測実行時間を決定係数および平均相対誤差（MRE）の指標で評価した結果を表 5 に記す．既存手法による予測実行時間の決定係数 r^2 および MRE は式 (4) および (5) で算出した．提案手法における指標も式 (4) および (5) と同様に算出した．

$$r^2 = 1 - \frac{\sum_{i=2}^{|A|} \left(t_i - \frac{m}{2} T(a_i, \emptyset)\right)^2}{\sum_{i=2}^{|A|} \left(t_i - \bar{t}\right)^2} \qquad (4)$$

$$\mathrm{MRE} = \frac{1}{|A| - 1} \sum_{i=2}^{|A|} \frac{\left|t_i - \frac{m}{2} T(a_i, \emptyset)\right|}{t_i} \qquad (5)$$

ここで，$\bar{t}$ は t_2 から $t_{|A|}$ の平均を表す．決定係数 r^2 は値が 1 に近いほど，MRE は 0 に近いほど予測正確度が高い．

表 5 より，どちらの指標でも提案手法は既存手法よりも予測正確度が高い．提案手法による MRE は既存手法の MRE の約 43%である．しかし，すべての手法において MRE が 100 以上であり誤差の程度が大きい．

提案手法が誤差を小さくできた原因および両手法ともに相対誤差が大きい原因を調べるために各ジョブにおける予測実行時間の相対誤差を調べた（**図 6**）．図 6 のジョブ番号 i は，ジョブ a_i を表す．a_{11}〜a_{13} において，既存手法と比較して提案手法による相対誤差が小さい．図 5 より，a_{11}〜a_{13} は計算時間の大きい反復処理 v_k を含む．実際に

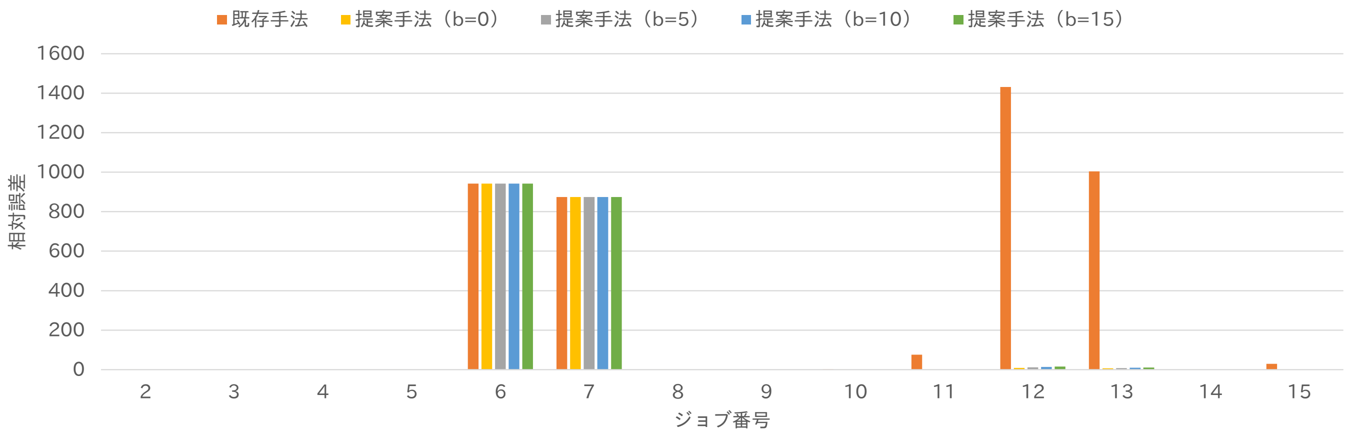

図 6 実測値に対する予測実行時間の相対誤差

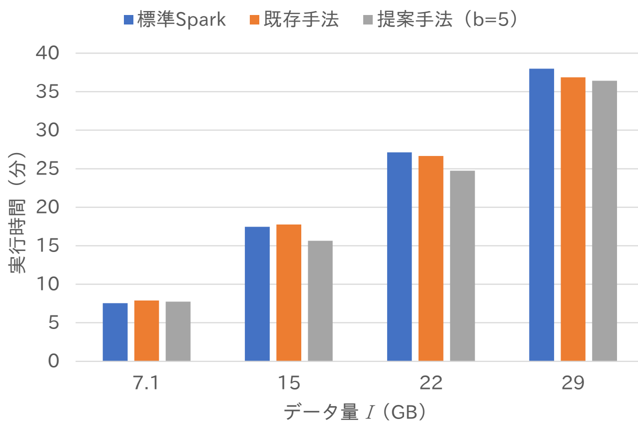

図 7 提案手法による実行時間の削減

は a_{10} において Spark が $F(v_k, v_l)$ を作成するので v_k の再計算は a_{11} 以降発生しない．しかし，既存手法はシャッフルファイルの再利用を再現しないので a_{11} 以降も v_k の計算時間を加算した．結果，実際の実行時間よりも大きく見積った．一方，提案手法はシャッフルの再現によって a_{11}〜a_{13} において v_k の計算時間を加算しないので，実行時間の予測誤差は小さい．

また，図 6 よりすべての手法で a_6 および a_7 の予測実行時間の相対誤差が大きい．これは，0.03 秒未満だった t_6 および t_7 をすべての手法で約 19 秒と推定したことが原因である．他のジョブ実行時間と比較して t_6 および t_7 が小さいのは，a_6 および a_7 が `isEmpty()` というアクション操作だからである．`isEmpty()` は RDD の実体データが存在すれば真を，存在しなければ偽を返すアクション操作であり，一部の実体データのみを計算するので計算に要する時間が小さい．したがって，予測実行時間の計算にはアクション操作の種類も考慮する必要があり，これは今後の課題である．

5.4 キャッシュ自動選択への応用

標準の Spark と，既存手法および提案手法を適用した MCR によるプログラムの実行時間を**図 7** に示す．ただし，入力データ量 I の変更は，入力データセットの内容を複製して調整した．また，各入力データについて 7 回ずつ実行し，実行時間の中央値を取った．なお，表 5 より $b = 5$ の場合で決定係数および MRE が中央値だったので，通信コストの代表値として妥当と考えて $b = 5$ に指定した．

図 7 より既存手法と提案手法を比較すると，すべての入力データ量について実行時間を小さくできた．特に $I = 15$ の場合に 1.1 倍高速化した．標準の Spark と提案手法を比較すると $I \geq 15$ の場合，実行時間を小さくできた．特に $I = 15$ で 1.1 倍高速化した．また，実行時間の増大は $I = 7.1$ で高々 3%である．

実行時間を小さくできた原因を調べるために，$I = 29$ の場合においてジョブごとに実行時間を計測した（**図 8**）．図 8 のジョブ番号 i は，ジョブ a_i を表す．提案手法は特にジョブ a_4 において実行時間を小さくできた．これは提案手法はキャッシュ指示の利得が低い $\hat{v_b}$ をキャッシュしないからである．a_4 において，すべての手法は図 5 の RDD v_a および RDD v_b にキャッシュ指示を与える．$I \geq 15$ の場合，$\hat{v_a}$ および $\hat{v_b}$ はどちらも IMC 容量よりも大きいため，すべてを IMC に格納できない．a_4 の開始時，IMC には a_1 においてキャッシュした $\hat{v_a}$ が格納されている．

暗黙的な再計算の省略を再現しない既存手法では，$\hat{v_b}$ を使用するジョブは a_4 および a_6〜a_{13} であると推定する．a_4 以降で $\hat{v_a}$ をキャッシュした場合に削減できる計算コスト $R(v_a, C)$ は 588，$R(v_b, C)$ は 477 である．$\hat{v_a}$ と $\hat{v_b}$ のサイズ比 $D(v_a) : D(v_b)$ は約 1.25 : 1 であるから，キャッシュ指示の利得の比 $\epsilon(v_a, C) : \epsilon(v_b, C) = R(a, C)/D(a) : R(b, C)/D(b)$ は約 470.4 : 477 となり，$\hat{v_a}$ よりも $\hat{v_b}$ を優先して IMC に格納する．

一方，暗黙的省略を再現する提案手法では，a_8 において作成される $F(v_d, v_e)$ の存在により，$\hat{v_b}$ を使用するジョブは a_4 および a_6〜a_8 であると推定する．a_4 以降では $R(v_a, C) = 343$，$R(v_b, C) = 212$ となり，どちらも既存手法よりも小さくなる．中間結果のサイズ比は既存手法と同

Vol.2020-OS-150 No.8
2020/7/30

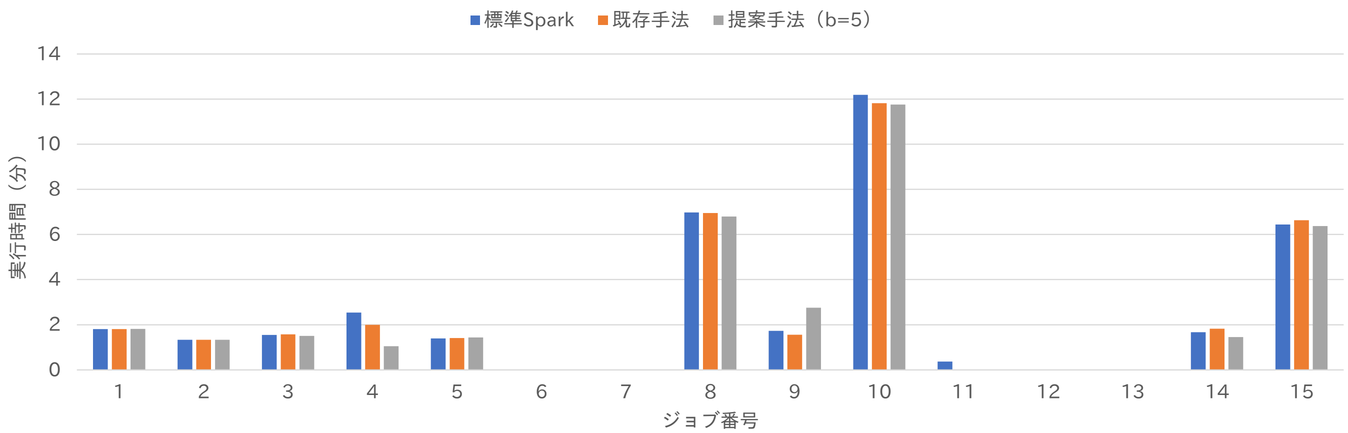

図 8 $I = 29$ における実行時間の内訳

じ $D(v_a) : D(v_b) \approx 1.25 : 1$ であるから，キャッシュ指示の利得の比は $\epsilon(v_a, C) : \epsilon(v_b, C) \approx 274.4 : 212$ となり，$\widehat{v_b}$ よりも $\widehat{v_a}$ を優先して IMC に格納する．したがって，提案手法は既に IMC に存在する $\widehat{v_a}$ を優先して残すので $\widehat{v_b}$ をキャッシュしない．IMC の置換アルゴリズムが LRU である標準の Spark も既存手法と同様に $\widehat{v_b}$ をキャッシュする．したがって，既存手法と標準 Spark では $\widehat{v_b}$ をキャッシュする処理時間が加わり a_4 の実行時間が大きくなった．

既存手法と比較して標準 Spark の方が実行時間が大きくなった原因は，キャッシュスラッシングにある．標準 Spark は a_1 において格納できなかった $\widehat{v_a}$ の一部も a_4 でキャッシュするので，$\widehat{v_a}$ および $\widehat{v_b}$ を交互にキャッシュし，キャッシュスラッシングが発生する．既存手法では $\widehat{v_a}$ よりも $\widehat{v_b}$ を優先して IMC に格納するので，a_4 において新たに $\widehat{v_a}$ をキャッシュしない．

提案手法では他の手法と比較してジョブ a_9 において実行時間が大きくなった．これは，a_8 における $\widehat{v_e}$ のキャッシュ優先度が他の手法よりも低く，a_9 で v_e の再計算が発生するからである．図 5 より a_{10} 以降は $F(v_e, v_f)$ および $F(v_e, v_h)$ が存在し v_e を再計算することはない．$\widehat{v_e}$ をキャッシュして削減できるのは a_9 の実行時間のみであるから，提案手法では $\widehat{v_e}$ よりも $\widehat{v_c}$ を優先してキャッシュし a_9 で v_e を再計算した．結果，a_9 の実行時間は大きくなったが，$\widehat{v_c}$ が IMC に存在していたので a_{14} および a_{15} の実行時間は削減できた．しかし，$I = 29$ の場合では IMC と比較して RDD の容量が大きく，他の中間結果を IMC に格納する目的で $\widehat{v_c}$ の一部を破棄する．その結果 a_9 を実行中に IMC 内の $\widehat{v_c}$ が 3 分の 1 に減少したので，a_{14} および a_{15} において削減できた実行時間は小さい．

6. まとめ

本稿では，シャッフルに伴う暗黙的なキャッシュを再現した実行時間予測手法を提案した．提案手法は（1）シャッフル依存関係の検出および（2）シャッフルを意識した実行時間予測の 2 段階で構成される．（1）は小規模な事前実行の実行時情報からシャッフル依存関係を検出する．（2）は RDD の依存グラフを辿って Spark プログラムの実行を再現し，実行時間を予測する．（2）においてシャッフル依存関係を辿る回数を高々 1 にすることで，シャッフルに伴う中間結果の再利用を再現する．

実験において，予測に要する時間，予測の正確度およびキャッシュ置換アルゴリズムへの応用の観点から提案手法を評価した．シャッフルを含むプログラムを対象とした場合，既存手法と比較して予測に要する時間を約 8 倍高速化し，予測の相対誤差を 58%削減した．また，予測実行時間に基づくメモリ内キャッシュ置換アルゴリズムに提案手法を適用すると，既存手法と比較して Spark プログラムを最大 1.1 倍高速化した．

今後の課題は，アクション操作の種類による実行時間の変動調査と各 RDD 操作に要する時間およびシャッフル通信に要する時間を自動的に予測することである．

謝辞 本研究の一部は，JSPS 科研費 JP15H01687 および JP16H02801 の補助による．

参考文献

[1] The Apache Software Foundation: Apache Spark - Unified Analytics Engine for Big Data, https://spark.apache.org/ (Accessed 2020-06-25).

[2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing (HotCloud '10)*, pp. 1–7 (2010).

[3] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *Proc. 9th USENIX Symp. Networked Systems Design and Implementation (NSDI '12)*, pp. 15–28 (2012).

[4] Gupta, G. P. and Kulariya, M.: A Framework for Fast and Efficient Cyber Security Network Intrusion Detection using Apache Spark, *Procedia Computer Science*, Vol. 93, pp. 824–831 (2016).

[5] Harnie, D., Saey, M., Vapirev, A. E., Wegner, J. K., Gedich, A., Steijaert, M., Ceulemans, H., Wuyts, R. and De Meuter, W.: Scaling machine learning for target pre-

ⓒ 2020 Information Processing Society of Japan

diction in drug discovery using Apache Spark, *Future Generation Computer Systems*, Vol. 67, pp. 409–417 (2017).

[6] Capuccini, M., Ahmed, L., Schaal, W., Laure, E. and Spjuth, O.: Large-scale virtual screening on public cloud resources with Apache Spark, *J. Cheminformatics*, Vol. 9, No. 15, pp. 1–6 (2017).

[7] Jiang, M., Gallagher, B., Chu, A., Abdulla, G. and Bender, T.: Exploiting Spark for HPC Simulation Data: Taming the Ephemeral Data Explosion, *Proc. Int'l Conf. High Performance Computing in Asia-Pacific Region (HPCAsia '20)*, pp. 150–160 (2020).

[8] Rana, N. and Deshmukh, S.: Shuffle Performance in Apache Spark, *Int'l J. Engineering Research & Technology (IJERT)*, Vol. 4, pp. 177–180 (2015).

[9] Davidson, A. and Or, A.: Optimizing Shuffle Performance in Spark, *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep.* (2013).

[10] Yang, Z., Jia, D., Ioannidis, S., Mi, N. and Sheng, B.: Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks, *Proc. 11th Int'l Conf. Cloud Computing (CLOUD '18)*, pp. 277–284 (2018).

[11] Nasu, A., Yoneo, K., Okita, M. and Ino, F.: Transparent In-memory Cache Management in Apache Spark based on Post-Mortem Analysis, *Proc. 7th IEEE Int'l Conf. Big Data (Big Data '19)*, pp. 3388–3396 (2019).

[12] Gottin, V. M., Pacheco, E., Dias, J., Ciarlini, A. E. M., Costa, B., Vieira, W., Souto, Y. M., Pires, P., Porto, F. and Rittmeyer, J. G.: Automatic Caching Decision for Scientific Dataflow Execution in Apache Spark, *Proc. 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR '18)*, pp. 1–10 (2018).

[13] Duan, M., Li, K., Tang, Z., Xiao, G. and Li, K.: Selection and replacement algorithms for memory performance improvement in Spark, *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 8, pp. 2473–2486 (2016).

[14] Lijie Xu: GitHub - JerryLead/SparkInternals, `https://github.com/JerryLead/SparkInternals/tree/e52b94080ce75aa83a6fa73c0cab86669cf89c5e` (Accessed 2020-06-25).

[15] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S. et al.: Mllib: Machine Learning in Apache Spark, *J. Machine Learning Research*, Vol. 17, No. 1, pp. 1235–1241 (2016).

[16] GroupLens Research: MovieLens, `https://grouplens.org/datasets/movielens/` (accessed 2020-06-25).

ⓒ 2020 Information Processing Society of Japan