Vol.2019-OS-147 No.7
2019/7/25

Apache Sparkにおけるデータ依存グラフ解析に基づくメモリ内キャッシュ置換手法

那須 敦也[1,a)]　置田 真生[1,b)]　伊野 文彦[1,c)]

概要：Apache Spark におけるプログラムの高速化を目的として，実行時間の予測に基づくメモリ内キャッシュの置換手法を提案する．提案手法は，メモリ内キャッシュの作成指示を自動挿入する既存システムを拡張する．まず小規模な事前実行から得たデータ依存グラフを解析して，キャッシュの組み合わせによる実行時間の増大を予測する．次に大規模な本実行において，予測結果に基づいて，高速化に対する寄与が大きいデータの組み合わせを貪欲に選択し，それらを優先的に残す置換手法を実現する．実験の結果，メモリ内キャッシュの置換が頻繁に発生する状況において，提案手法は既存の置換手法と比較して 1.0 倍〜1.5 倍の高速化を得た．さらに，開発者によりキャッシュ指示の最適化を施したプログラムと比較して最大で約 1.1 倍の高速化を実現した．提案手法は，PC クラスタの総メモリ容量を超える大規模なデータ処理の高速化をキャッシュ指示に関する試行錯誤なしに実現できる点で有用である．

キーワード：ユーザ透過性，ソフトウェアキャッシュ，PC クラスタ

1. はじめに

大規模データのための並列分散処理基盤として Apache Spark（以下，Spark）[1] がある．Spark の特長は，処理が主記憶上で完結するインメモリ計算にあり，Hadoop の課題であるストレージへのアクセスを回避する．さらに，Spark は高い耐障害性を実現する．具体的には，Resilient Distribution Dataset（RDD）[2] と呼ばれる不変（immutable）かつ並列処理可能なデータ構造を用い，RDD に対する一連の処理（系譜）を逆探索することで障害によって失われたデータの再計算を可能とする．RDD に対する系譜は，RDD を頂点とする有向非循環グラフとして表現され，遅延評価により処理の最適化を実現する．

さらに，Spark は一部の RDD のみをインメモリキャッシュ（IMC）に格納し，それらを得る過程の中間結果を破棄することによりメモリ使用量を節約する．IMC に格納する RDD の組み合わせはキャッシュ作成指示を用いて明示する．プログラムの高速化のためには，プログラマが試行錯誤してキャッシュ作成指示を与える最適な RDD の組み合わせを選択する必要がある．したがって，再計算を最小化できる自動的なキャッシュ手法の開発が望まれている．

米尾ら [3] は，Spark の利便性向上を目的として，キャッシュ指示自動化システム（以下，既存システム）を開発した．米尾らは，再計算を排除するためのキャッシュ作成指示の必要十分条件を明らかにした．メモリ容量が十分大きくキャッシュの置換が発生しない場合，再計算を完全に回避できる．一方，メモリ容量と比較して入力データ量が大きい場合，IMC の置換が頻繁に発生することで大量の再計算が必要となり，実行時間が増大する．米尾らはさらに，再計算に要する時間の単純な推定に基づく置換アルゴリズム MD を提案し，問題規模が大きい場合の置換の頻度を削減した．しかし，開発者によりキャッシュ指示の最適化を施したプログラム（以下，手動選択）と比較すると再計算時間が大きく，MD は再計算の最小化に不十分である．

そこで本研究では，プログラムの実行時間の予測に基づく IMC の置換手法を提案する．具体的には，IMC に格納する RDD の組み合せを変更した場合の全体実行時間を予測し，ある RDD の削除により誘発する他の RDD の再計算時間を推定する．提案手法はこの推定をもとに，誘発する再計算時間の小さい RDD を優先的に置換する．全体実行時間の予測には，プログラム全体の依存グラフおよび RDD 処理単位の計算時間が必要となる．そこで既存システム [3] を拡張し，小規模な事前実行で得た依存グラフを解析して，続く大規模な実行の実行時間を削減する．なお，

[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)] a-nasu@ist.osaka-u.ac.jp
[b)] okita@ist.osaka-u.ac.jp
[c)] ino@ist.osaka-u.ac.jp

遅延評価が原因で RDD 処理単位の時間計測が難しいため，提案手法では熟練者が設定した RDD 処理の計算コストで代替する．

提案手法を用いることで，ユーザプログラムの改変なしに，手動選択と同程度の時間でユーザプログラムを実行できる．さらに，キャッシュ作成指示に関する試行錯誤が不要であるため，ユーザはプログラムの開発に専念できる．

以降，まず 2 節で関連研究を紹介し，3 節で Spark の概要を示し，4 節で既存システムについて説明する．5 節で提案手法について説明し，6 節で評価実験を示す．最後に 7 節で本稿をまとめる．

2. 関連研究

Spark におけるプログラムの高速化を目的として，標準の LRU 置換を拡張する研究がある．Duan ら [4] は予測した重みに基づく置換手法 Weight Replacement（WR）Algorithm を提案した．WR は実行中のジョブ（プログラムの一部）に対して，RDD に対する操作（以下，RDD 操作）の計算時間，各 RDD 操作から得た計算結果のデータ量，および計算結果の該当ジョブにおける使用回数の 3 つの観点から，各 RDD 操作の重要性を推定する．そして，重要性が小さい計算結果のキャッシュを優先的に置換する．WR では，該当ジョブ内だけでの情報で重要性を推定するため，局所的な最適化に陥る可能性がある．一方，提案手法は事前実行が必要であるが，プログラム全体の依存グラフからキャッシュの重要性を予測する．そのため，大域的な最適化を期待できる．

また，キャッシュ作成指示を与えるべき最適な RDD の組み合わせを決定する研究がある．Gottin[5] らは，依存グラフにおける RDD 操作の計算時間をすべて推定したコストモデルを入力として，実行時間が最小となる RDD のキャッシュを決定する S-CACHE を提案した．S-CACHE はキャッシュの置換が発生しないことを前提として，キャッシュ作成指示を与える RDD を決める．したがって，IMC 容量が十分大きい場合に有用である．一方，提案手法はすべての再計算を回避可能な必要最低限のキャッシュのうち，実行時間を削減するキャッシュを優先的に残す置換手法である．提案手法を用いることで，キャッシュの置換が発生する大規模なデータ処理を高速化できる．

3. Apache Spark

Spark はマスタ・ワーカ型のフレームワークである．ユーザはマスタ上で動作するドライバ・プログラムのみを記述する．ワーカ上の分散実行，すなわちデータの分散，タスク分割，スケジューリングおよび通信は Spark が暗黙的に実行する．

ドライバ・プログラムは 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．

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:
　$\widehat{a}$: 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 ← ∅;                                   ▷ a の計算に必要なデータ
2: for all v ∈ P(a) do
3:     if v̂ ∈ C then
4:         Q ← Q + {v̂}
5:     else
6:         Q ← Q + {X(v)}
7: â ← (Q を入力として a を実行 )
8: if a にキャッシュ指示があれば then
9:     C ← C + {â}                          ▷ 必要に応じて置換
10: return â
```

Spark は変換の呼び出し時，計算の実行を保留し依存関係のみを記憶する．アクションの呼び出し時，依存関係を辿って必要なすべての操作を実行する．1 つのアクションの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ワーカ間通信を必要に応じて行う．この通信をシャッフルと呼ぶ．

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 $A \subset V$，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) は頂点間の依存関係を表し，RDD 操作 v の入力として RDD 操作 u の計算結果が必要であることを意味する．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 は各操作の計算に要した時間を表し，データ量 D は各操作の計算結果の大きさを表す．以降では $v \in V$ の計算結果を $\widehat{v}$ で表現する．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．

ジョブ a_i 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．また，$X(a_i)$ の計算に必要な G の部分グラフを a_i の系譜 $L(a_i)$ と呼ぶ．すなわち $L(a_i)$ は a_i に到達可能な頂点からなる部分グラフである．

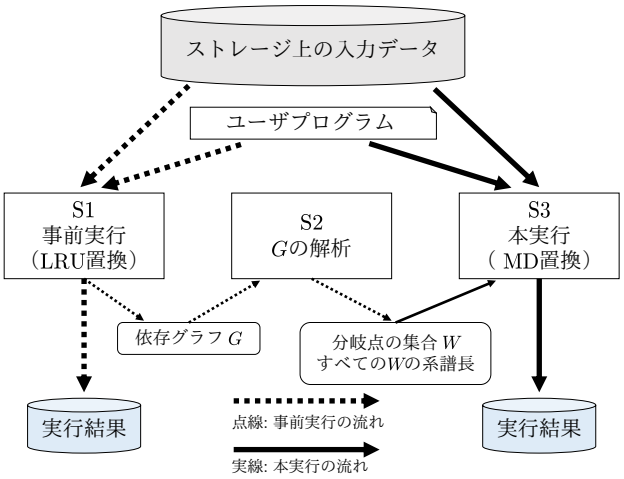

図 1　既存システムの概要

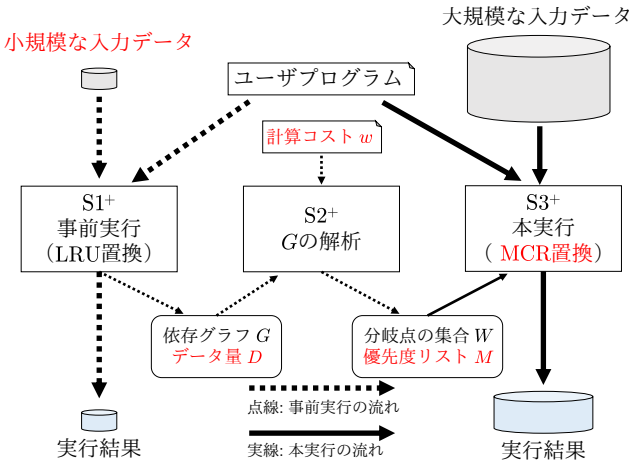

図 2　提案システムの概要

4. 既存システム

図 1 に既存システム [3] の概要を示す．既存システムは以下の 3 段階で構成される．ただし，依存グラフ G における出次数が 2 以上である頂点 v を分岐点として，分岐点の集合 $W \subset V$ を定義する．また，系譜 $L(v)$ の直径を系譜長と呼び $|L(v)|$ と表す．

（S1） 依存グラフ G を取得する事前実行
（S2） W を決定するための G の解析
（S3） キャッシュ作成指示を自動挿入する本実行

W の全要素の計算結果を IMC に格納可能ならば，すべての再計算の発生を回避できる [3]．そこで，既存システムは IMC の容量が十分大きいことを仮定して，W の全要素に対してキャッシュ作成指示を挿入する．実際に IMC の容量が不足する場合は，MD に基づいて系譜長 $|L(v)|$ が小さい v の計算結果 $\hat{v}$ を優先的に置換する．

既存システムは，まず S1 から G を取得する．次に S2 において，G 全体を解析することで W および $|L(v \in W)|$ を決定する．最後に S3 において，$v \in W$ の実行時に自動的にキャッシュ作成指示を挿入する．ただし，S1 および S3 で G が同一であることを保証するために，S1 において S3 と同一の入力データを用いる制約がある．

4.1 既存システムにおける課題

既存システムに以下に示す 2 つの課題がある．

（P1） IMC 不足時における実行時間の増大
（P2） 事前実行のオーバヘッドが大きい

P1 の原因は RDD 操作 v の計算結果 $\hat{v}$ のキャッシュミスにより生じる再計算時間と系譜長として予測した再計算時間の不一致である．MD は系譜長が小さい v の $\hat{v}$ を優先的に置換する．しかし，IMC に存在する計算結果の組み合わせに依存して，$\hat{v}$ の置換が他の RDD 操作の再計算を誘発するため，系譜長は発生する再計算時間の予測として不十分である．したがって，実際に発生する再計算時間により近い時間を予測するために，$\hat{v}$ の置換が誘発する再計算時間の推定が必要である．

P2 の原因は，オーバヘッドすなわち S1 および S2 の合計実行時間が，原則として S3 の実行時間と同じかそれ以上を要するためである．これは S1 および S3 で同一の入力データを用いる制約に起因する．したがって，既存システムが有用となる状況は，同一データに対する実行を繰り返す用途に限られる．

5. 提案手法

提案手法の目的は，再計算時間の予測による置換アルゴリズムの改善および既存システムのオーバヘッド削減である．P1 を解決するために，メモリ使用量あたりの高速化に対する寄与の小さな計算結果 $\hat{v}$ を優先的に置換する MCR を提案する．MCR は以下 2 段階からなる．1 つは，実行時間を予測して v に対する置換の優先度（以下，置換優先度）を導出する pMCR である．もう 1 つは，置換優先度に基づいて IMC に存在する $\hat{v}$ を置換する mMCR である．MCR の高速化のために，pMCR は本実行前に処理する．

pMCR では，依存グラフ G を入力として，$\hat{v}$ のキャッシュミスが誘発する再計算時間を予測する．また，再計算時間の予測には，小規模な入力データを用いた事前実行で得たデータ量の列 D を用いる．さらに，本実行における $\hat{v}$ のデータ量が不明であるため，pMCR では実行時間の予測から v の置換優先度を決定する．

mMCR では，pMCR から得た置換優先度を参照して，IMC 不足時に置換優先度が高い v の $\hat{v}$ を破棄することで IMC の空き容量を確保する．

また，P2 を解決するために，事前実行および本実行で生成する依存グラフ G が異なる場合，参照する置換優先度を選択するアルゴリズム JCP を提案する．

5.1 提案システム

拡張した提案システムの概要を図 2 に示す．提案システムは，既存システムの S1，S2 および S3 を以下のように拡張する．まず小規模な事前実行から依存グラフ G およびデータ量の列 D を得る（S1$^+$）．次に，pMCR において，G，計算コスト w および D を入力として，分岐点集合 W を決定する．その結果，置換優先度リストの列 M を得る（S2$^+$）．最後に，大規模な本実行において，W に対して自動的にキャッシュ作成指示を与え，IMC 不足時は mMCR に基づいてキャッシュを置換する（S3$^+$）．ただし，Spark では，遅延評価による最適化によって v を一括実行し，正確な計算コスト $w(v)$ の測定は難しい．そのため，$w(v)$ はユーザの指定値を用いる．

5.2 置換優先度の導出法

RDD 操作 v に対する最適なキャッシュの組合せを求める問題は，IMC 容量を容量とし，v をキャッシュすることで削減可能な再計算の時間を価値としたナップサック問題に帰着できる．したがって，最適な v の置換優先度を得るためには，組み合わせ最適化問題を解く必要がある．ただし，キャッシュ作成指示を与える v の組み合わせによって価値が増減するため，動的計画法を適用できない．本研究では，貪欲法を用いたヒューリスティック解法を提案する．

5.2.1 貪欲法を用いた置換優先度の導出

まず，アクション a_k の実行時に IMC C へ $\widehat{v} \notin C$ を追加することで，削減可能な再計算の総時間 $R(a_k, v, C)$ の推定方法を式 (1) に示す．

$$R(a_k, v, C) = T(a_k, C) - T(a_k, C + \{\widehat{v}\}) \qquad (1)$$

ここで $T(a_k, C)$ は，IMC C のもとで a_k 以降の全アクションの実行に要する時間の合計を表す．IMC に新たなキャッシュを追加すると，a_k だけでなく以降の全アクションにおいて再計算を削減する可能性がある．再計算以外の要因で実行時間が変化しないと仮定すると，$R(a_k, v, C)$ は式 (1) のように実行時間の差分で計算できる．

アクション a 単体の実行時間 $t(a, C)$ は，アルゴリズム 1 より，式 (2) のように再帰的に計算する．

$$t(a, C) = \sum_{p \in P(a)} \begin{cases} 0 & (\widehat{p} \in C) \\ w(p) + t(p, C) & (\text{otherwise}) \end{cases} \qquad (2)$$

$T(a_k, C)$ は各アクションの実行時間の総和として式 (3) で表現する．

$$T(a_k, C) = \sum_{i=k}^{|J|} t(a_i, C) \qquad (3)$$

次に，貪欲法を用いてジョブごとの置換優先度リストを決定する pMCR をアルゴリズム 2 に示す．pMCR の出力 M の要素 M_k は，アクション a_k の実行における置換優先度リストである．M_k は a_k 以前の実行において計算対象となるキャッシュ候補を要素とし，先に置換対象となるべき順序に並べた列である．ジョブごとに異なる置換優先度リストを作成する理由は，プログラムの進行にしたがって v の置換優先度が変化するためである．例えば，プログラム前半の計算で $\widehat{v}$ を頻繁に必要とする間は低い置換優先度を保つべきであるが，プログラム後半の計算で $\widehat{v}$ を一切必要としないならば高い置換優先度に変更する必要がある．

アルゴリズム 2 置換優先度の導出アルゴリズム pMCR

Input:
G: 依存グラフ
$J = (a_1, a_2, .., a_{|J|})$: ジョブの列
W: 分岐点（キャッシュ候補）の集合
D: 計算結果のデータ量
Output:
$M = (M_1, M_2, .., M_{|J|})$: 置換優先度リストの列
1: $W_p \leftarrow \emptyset$ ▷ これまでに計算対象となったキャッシュ候補
2: **for** $i = 1$ to $|J|$ **do**
3: $O \leftarrow$ 系譜 $L(a_i)$ に含まれる頂点の集合
4: $W_a \leftarrow W_p \cup (W \cap O)$ ▷ 新たに計算対象となる候補を追加
5: $W_p \leftarrow W_a$
6: $M_i \leftarrow ()$
7: $C \leftarrow \emptyset$
8: **while** $W_a \neq \emptyset$ **do**
9: $W_L = \arg\max_{v \in W_a}(R(a_i, v, C)/D(v))$
10: M_i の先頭から W_L の要素を追加
11: $C \leftarrow C + W_L$ ▷ 貪欲にキャッシュの組み合せを固定
12: $W_a \leftarrow W_a - W_L$
13: **return** M

最後まで IMC 上に残すべき計算結果は，それ単体で高速化への寄与が最大となる計算結果である．そこで，データ量あたりの再計算時間の削減量を高速化の寄与として，キャッシュ候補 $v \in W_a$ の中で $R(a_k, v, \emptyset)/D(v)$ が最大となる計算結果 v_1 について，その置換優先度が最低になるよう M_k の末尾に設定する．そして，v_1 と組み合わせて IMC に格納した場合，高速化の寄与が次に最大となる計算結果 v_2 を M_k の先頭から追加する．これをすべてのキャッシュ候補を M_k に追加するまで繰り返す．

5.3 置換優先度リストに基づく置換アルゴリズム

Spark を拡張し，IMC の置換発生時に用いる置換アルゴリズムを，標準の LRU からアルゴリズム 3 に示す rMCR に差し替える．ここで $\lambda(c)$ は計算結果 c を生成した元の操作 v を表し，$M_i^{-1}(v)$ は M_i における v の添え字，すなわち v の置換優先度を表す．なお，v が M_i^{-1} に含まれない場合は，もっとも置換優先度が高いとみなして置換優先度を -1 とする．また，$D'(v)$ は本実行における計算結果 $\widehat{v}$ の大きさを表す．入力する置換優先度リストはジョブ毎に異なる．そこで後述の JCP を用いて選択した M_k を与える．

$\widehat{v}$ をキャッシュするために，v よりも置換優先度が高い

アルゴリズム 3 置換優先度リストに基づく置換 mMCR

Input:
v: IMC に計算結果の格納を試みる RDD 操作
C: IMC に存在する計算結果の集合
e: IMC の空き容量
M_k: 実行中のジョブに対する置換優先度リスト
Output:
C
1: $(C_1, C_2, \ldots, C_{|C|}) \leftarrow M_k^{-1}(\lambda(c))$ $(c \in C)$ の昇順に C をソート
2: $H \leftarrow \emptyset$ ▷ 追い出す対象の候補集合
3: **for** $i = 1$ **to** $|C|$ **do**
4: **if** $M_k^{-1}(v) \leq M_k^{-1}(\lambda(C_i))$ **then**
5: **return** C ▷ 置換せずに返す
6: $H \leftarrow H + \{C_i\}$
7: $e \leftarrow e + D'(\lambda(C_i))$
8: **if** $e \geq D'(v)$ **then**
9: $C \leftarrow C - H + \{\widehat{v}\}$
10: **return** C
11: **return** C ▷ $\widehat{v}$ のキャッシュを諦める

RDD 操作 u の計算結果 $\widehat{u}$ を破棄することで IMC の空き容量を確保し，$\widehat{v}$ を IMC に格納する．ただし，v より置換優先度が高い u の $\widehat{u}$ をすべて破棄しても IMC の空き容量を確保できない場合，mMCR は $\widehat{v}$ のキャッシュを諦める．

5.4 置換優先度リストの選択

本実行 S3$^+$ の実行中，j 番目のアクション a'_j を開始する際に，a'_j の計算中に参照する置換優先度リスト M_k を決定する．具体的には，a'_j に対応する事前実行 S1$^+$ のアクション a_k を特定し，a_k の置換優先度リスト M_k を用いる．本研究の前提として，対象となるプログラムの挙動は同一の入力に対して決定的であると仮定する．したがって，仮に S1$^+$ および S3$^+$ それぞれの依存グラフ G および G' が等しければ，a'_j および a_k のソースコード行が一致し，かつそれぞれの実行における実行順が同じ場合に $a'_j = a_k$ である．以降では，2 つのアクションのソースコード行が一致する場合に真となる述語 $S(a_i, a_j)$ を用いる．

しかし，S1$^+$ および S3$^+$ では入力データが異なるため，$G \neq G'$ となる可能性がある．実行するプログラムは同一であるため，依存グラフが変化する原因は条件分岐および繰り返しの終了判定の 2 つである．これらに起因するグラフの変化は以下の 3 つに分類できる．まず，G に存在した頂点 v が G' に存在しない場合を省略と呼ぶ．次に，逆に G に存在しない頂点 x が G' に存在し，かつ $S(x, y)$ が真となる頂点 y $(y \neq x)$ が G' に存在する場合を反復と呼ぶ．最後に，固有のソースコード行を持つ頂点 u が G' のみに存在する場合を追加と呼ぶ．

本研究では，依存グラフ全体ではなくアクションのみを比較することでアクションの対応を決定する．S1$^+$ および S3$^+$ で実行するソースコードは同一であるため，各アクションの系譜は変化しない．したがって，S1$^+$ および S3$^+$ におけるそれぞれのアクションの系列 J および J' を比較すれば十分である．

アルゴリズム 4 置換優先度リストの動的選択 JCP

Input:
$M = (M_1, M_2, \ldots, M_{|J|})$: 置換優先度リストの列
$J = (a_1, a_2, \ldots, a_{|J|})$: 事前実行におけるジョブの列
a'_j: 本実行における実行中のジョブ
Output:
M_k: 実行中のジョブに割り当てる置換優先度リスト
Global variables:
h: 本実行の進行と J の対応を表すヘッド（0 で初期化）
1: $h \leftarrow h + 1$;
2: **if** $S(a_h, a'_j) = \text{true}$ **then**
3: $M_k \leftarrow M_h$
4: **return** M_k
5: **for** $l = h + 1$ **to** $|J|$ **do** ▷ 時系列の後方を探索
6: **if** $S(a_l, a'_j) = \text{true}$ **then** ▷ 省略の場合
7: $M_k \leftarrow M_l$
8: $h \leftarrow l$ ▷ ヘッドを先に進める
9: **return** M_k
10: **for** $l = h - 1$ **down to** 1 **do** ▷ 時系列の前方を探索
11: **if** $S(a_l, a'_j) = \text{true}$ **then** ▷ 反復の場合
12: $M_k \leftarrow M_l$
13: $h \leftarrow l$ ▷ ヘッドを反復の先頭に戻す
14: **return** M_k
15: $M_k \leftarrow M_{(h-1)}$ ▷ 追加の場合
16: $h \leftarrow h - 1$
17: **return** M_k

a'_j に対応するアクションを J 内で探索し，置換優先度リストを選択するアルゴリズム JCP をアルゴリズム 4 に示す．S3$^+$ の実行中アクションごとに順次 JCP を呼び出すため，JCP は J' 全体を参照できない．そこで S3$^+$ の進行を表すヘッド h を用いて，a'_j の実行直前における J との対応を記憶する．JCP を呼び出すたびにヘッドを 1 つ進め，$S(a_h, a'_j)$ が真であればアクションの系列が局所的に一致すると見なす（2 行目）．$S(a_h, a'_j)$ が偽の場合，まず J の a_h よりも後方を探索する．ソースコード行が同一の a_l が存在すれば，a_h から a_{l-1} までのアクションの省略を意味する（6 行目）．次に J の前方を探索し，ソースコード行が同一の a_l が存在すれば，a'_j をアクション a_l の反復と見なす（11 行目）．JCP の次の呼び出しに備えて，ヘッダは a_l を発見した位置に移動する．J に同一ソースコード行のアクションが存在しない場合，新規アクションの追加を意味する（15 行目）．この場合参照すべき置換優先度リストは存在しないため，直前に選択した置換優先度リストを暫定的に返す．1 つのアクションの追加を反映して，ヘッダを 1 つ戻す．

6. 評価実験

以下の観点から提案手法を評価する．

- 提案手法のオーバヘッド
- 既存手法 [3] との比較
- 手動選択との比較

表 1　実験環境（PC クラスタ）

ノード数	8 台
CPU	Intel Xeon E5-2650 v3 2.30 GHz (8 cores)
OS	CentOS 7.6-1810
Spark	Spark 2.1.0 (Standalone モード)
主記憶	4 GB（ノードあたり）
IMC 容量	2.1 GB（ノードあたり）
コア数	1 個（ノードあたり）

表 2　対象アプリケーション

アプリケーション	入力データセット
線形回帰	YearPredictionMSD [7] (0.4 MB)
k 平均法	HIGGS [7] (7.5 GB)
交互最小二乗法	MovieLens Latest Datasets [8] (0.7 GB)

表 3　依存グラフの特徴

	頂点数		ジョブ数		キャッシュ点数		
	$\|V\|$	$\|V'\|$	$\|J\|$	$\|J'\|$	$\|C_d\|$	$\|W\|$	$\|C_d \cup W\|$
線形回帰	323	317	55	54	1	2	2
k 平均法	38	38	11	11	4	4	5
ALS	359	359	13	13	12	12	14

$|V|$: 事前実行で生成する頂点数
$|V'|$: 本実行で生成する頂点数
$|J|$: 事前実行におけるジョブ数
$|J'|$: 本実行におけるジョブ数
C_d: 開発者が指示したキャッシュ点の集合

実験には 8 台の計算機で構成する PC クラスタ（表 1）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，コア数は実験の単純化のために 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 [6] から 3 つのアプリケーションを用いた．また，入力データは UCI 機械学習リポジトリおよび MovieLens から得た（表 2）．

アプリケーションにおける依存グラフの特徴を表 3 および表 4 に示す．キャッシュ点とは，キャッシュ作成指示を与える RDD 操作の集合を意味する．キャッシュ点は，事前実行 S1$^+$ では，開発者が指示したキャッシュ点の集合 C_d を表し，本実行 S3$^+$ では，分岐点集合 W を表す．線形回帰は分岐点が 2 つしか存在しない依存グラフ G を持ち，キャッシュ点が比較的少ない．k 平均法はユーザが指定した繰り返し回数に依存して，分岐点の数が増減する G を持つ．本実験では，4 つの分岐点を生成するデフォルトの回数を用いた．交互最小二乗法（ALS）は分岐点の数が 12 箇所存在し，キャッシュなしで実行した場合の再計算を含めたのべ計算回数（以下，計算数）が比較的多い G を持つ．k 平均法および ALS では，実行時間を最適化するためにプログラマが複数の RDD にキャッシュ作成指示を与える必要がある．

アプリケーションごとに S3$^+$ で指定した計算コスト w を表 4 に示す．w の正確な計測は難しいため，同じアプリケーションを用いた予備実験から熟練者が決定した．具体的には，比較的小さい時間で実行可能な RDD 操作 v に対応する $w(v)$ の値を 1 に設定した．また，v と比較して大きい時間を要する RDD 操作 u の $w(u)$ は $w(v)$ と比較して大きい相対値に設定した．

表 4　頂点の計算数および計算コスト

	計算数		RDD 操作の計算コスト w			
	$\|V_c\|$	$\|V'_c\|$	textFile	map	Bykey	others
線形回帰	537	527	277.5	1	1	1
k 平均法	208	208	89.4	1	1	1
ALS	6508	6508	47	2	18.25	1

$|V_c|$: 事前実行における計算数
$|V'_c|$: 本実行における計算数

表 5　事前実行および置換優先度解析の実行時間

	事前実行 S1$^+$（s）	置換優先度解析 S2$^+$ （ms）
線形回帰	7.6	436
k 平均法	4.5	52
ALS	26.8	4496

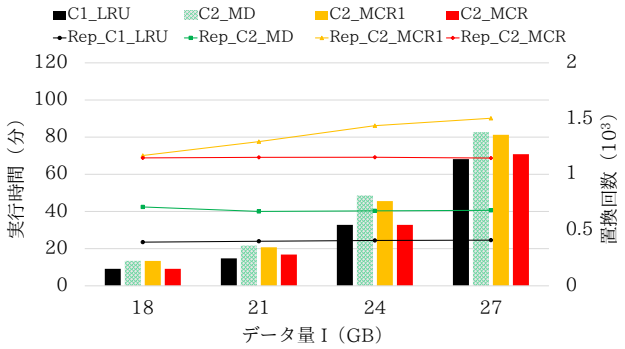

図 3　線形回帰における実行時間および置換回数

比較のために，2 つのキャッシュ指示方針と複数の置換アルゴリズムを組み合わせて評価する．C_d に手動でキャッシュ作成指示を与える実行を C1 とする．また，W を自動的にキャッシュする実行を C2 とする．置換アルゴリズムには MD，MCR1，MCR を用いる．MCR1 は計算コスト w がすべて 1 として実行時間を予測した置換手法である．ただし，本実験では，Spark 標準の振る舞いと比較するため，C1 における置換アルゴリズムは LRU とする（以下，C1_LRU）．

6.1　実験結果

アプリケーションごとの小規模な事前実行 S1$^+$ および優先度解析 S2$^+$ に要した時間を表 5 に示す．S1$^+$ はそれぞれのアプリケーションに小規模な入力データを用いて，local モードで試験的に実行した．また，アプリケーションごとの実行時間およびキャッシュの置換回数を図 3〜5 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節した．

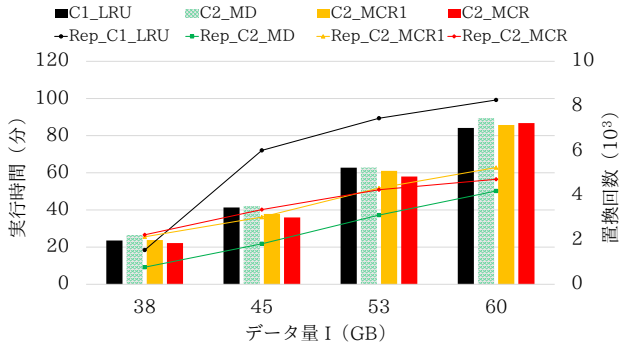

図 4　k 平均法における実行時間および置換回数

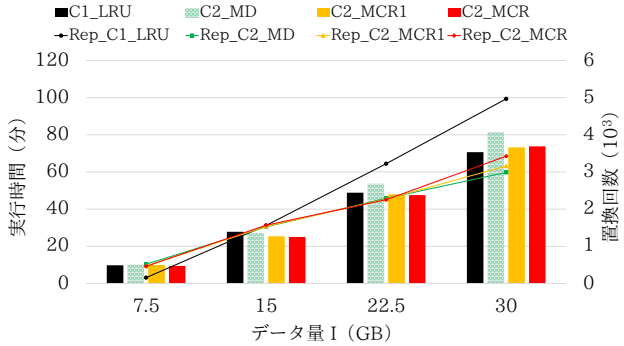

図 5　ALS における実行時間および置換回数

6.2　提案手法のオーバヘッド

提案手法を用いるためのオーバヘッドである事前実行 S1$^+$ ならびに置換優先度解析 S2$^+$ の実行時間を本実行 S3$^+$ と比較する．表 5 および図 3〜5 より，S1$^+$ および S2$^+$ の実行時間は S3$^+$ と比較して無視できるほど小さい．したがって，提案手法は 1 度だけ実行するプログラムに対しても有用である．

6.3　既存手法 C2_MD との比較

キャッシュミスにより発生する再計算時間の予測方針が異なる置換アルゴリズムを比較することで，提案手法の妥当性を評価する．比較の結果，C2_MCR1 および C2_MCR は C2_MD と比較してすべての場合で実行時間を削減した．実行時間の削減は計算コストを見積もった C2_MCR の方が大きく，C2_MD と比較して 1.0〜1.5 倍の高速化を得た．C2_MCR1 は C2_MCR と比較して，実行時間が増大する場合があるため，高速化のために RDD 操作に対する計算コストを予測する必要がある．したがって，3 つのアプリケーションについては，提案手法は MD と比較してプログラムの高速化が可能である．

また，提案手法の問題規模に対するスケーラビリティを MD と比較して評価する．スケーラビリティは図 3〜5 における問題規模に対する実行時間の増大から評価する．比較の結果，C2_MCR は C2_MD と比較してすべての場合でスケーラビリティが高いため，問題規模に対する実行時間の増大を抑制する．

6.4　手動選択 C1_LRU との比較

手動選択と提案手法を比較することで，提案手法の実用性を評価する．C2_MCR は C1_LRU と比較して，同程度かそれ以下の時間で実行できた．特に，k 平均法における $I \leq 53$ および ALS における $I \leq 22.5$ では，最大で約 1.1 倍の高速化を得た．また，実行時間の増大は高々 4 %である．さらに，提案手法のオーバヘッドは無視できるほど小さい（6.2 節）．したがって，提案手法はキャッシュの置換が発生する大規模なデータ処理の高速化をキャッシュ指示に関する試行錯誤なしで実現できる．

また，提案手法の問題規模に対するスケーラビリティを手動選択と比較して評価する．比較の結果，線形回帰では，問題規模の増大にともない，提案手法のスケーラビリティは低下する．k 平均法における $I \leq 53$ ではスケーラビリティが高いが，$I \geq 60$ では逆に低下した．また ALS においても，$I \leq 22.5$ ではスケーラビリティが高いが，$I \geq 30$ では低下した．したがって，実行時間の最適化のために複数の RDD 操作にキャッシュ作成指示を与える必要があるアプリケーションでは，高いスケーラビリティを達成できる．

6.5　考察

JCP による置換優先度リストの選択を考察する．また，3 つのアプリケーションに対する MCR1 および MCR による実行時間の増減を考察する．

6.5.1　JCP を用いた置換優先度の参照

本実験では，事前実行および本実行間における依存グラフの違いは，プログラムにおけるジョブの削減のみであった．線形回帰における事前実行では 55 回のジョブのうち，a_3〜a_{54} は同じソースコード行のジョブを繰り返す．一方，本実行では 54 回のジョブのうち，a'_3〜a'_{53} がジョブの繰り返しである．すなわち，本実行において，事前実行で実行した a_{54} を省略したため，a'_{54} 実行時に JCP により $a'_{54} = a_{55}$ と判定できた．

6.5.2　線形回帰

線形回帰では，2 つの分岐点が存在し，RDD の生成順に C2 は (w_1, w_2) にキャッシュ作成指示を与える．C1 では w_1 のみにキャッシュ作成指示を与える．そのため，C2 は置換回数が比較的多い．C2_MCR1 は 54 回のジョブのうち，a'_2〜a'_{51} において w_2 を優先的に IMC に格納し，それ以降のジョブでは，w_1 を格納する．C2_MCR はすべてのジョブにおいて，w_1 を優先的に格納する．すなわち，C1_LRU と同じ RDD を優先的に格納する．

C2_MCR1 はプログラム実行において，a_{52} 以降で w_1 の再計算が発生するため，結果的に実行時間が増大した．C2_MCR は C1_LRU と同じ w_1 を優先的に IMC に残すこ

とで，C2_MD と比較して実行時間の増大を回避した．

6.5.3 k 平均法

k 平均法では，RDD の生成順に C1 は (w_1, w_2, w_4, w_5)，C2 は (w_1, w_3, w_4, w_5) のそれぞれ 4 つの RDD にキャッシュ作成指示を与える．C1_LRU では，削減できる計算時間が長い RDD w_1 を頻繁に使用するため，結果的に w_1 が IMC に残る．一方，w_2，w_4 および w_5 については，IMC への格納後に置換が繰り返される．C2_MCR1 および C2_MCR では，入力する置換優先度リストが異なり，C2_MCR1 では w_3 の置換優先度が低く，C2_MCR では w_1 が低い．すなわち，C2_MCR1 および C2_MCR はそれぞれ w_3 および w_1 を優先的に IMC に格納する．さらに，11 回のジョブのうち，a'_3，a'_4 では w_4 を優先的に格納し，a'_5，a'_6 では w_5 を格納する．

C2_MCR1 および C2_MCR は，一時的に使用メモリ容量の小さい w_4 および w_5 を優先的に格納することで C1 より実行時間を削減する．しかし，$I = 60$ の場合，実行時間は高々 4 %増大したが，原因は特定できていない．

6.5.4 ALS

ALS では，C2_MCR1 および C2_MCR では，13 回のジョブのうち a'_{12} を除いたすべての置換優先度リストが同一である．C2_MCR1 および C2_MCR は C1_LRU と比較して，入力データ量が小さい $I \leq 22.5$ の場合，実行時間を削減し，$I = 30$ では実行時間が増大した．この原因は，a'_4，a'_5 に割り当てた置換優先度が不適切であるためである．具体的には，RDD の生成順に a'_4 では (w_1, w_2) にキャッシュ作成指示を与え，a'_5 では (w_1, w_2, w_3) に与える．C2_MCR1 および C2_MCR では，w_2 を優先的に格納するが，削減できる計算時間が大きい RDD は w_1 である．そのため，IMC 不足時に w_1 を置換することで，a'_4，a'_5 に要する時間が無視できない程増大する．

$I \leq 22.5$ の場合，C2_MCR1 および C2_MCR は a'_{12} に要する時間を削減する．ジョブ実行時に C1_LRU では一部の w_3 の計算結果が IMC に存在するが，C2_MCR1 および C2_MCR では，すべての w_3 の計算結果が IMC に存在するため，結果的に実行時間を削減する．しかし，$I = 30$ では，C2_MCR1 および C2_MCR においても，w_3 のすべての計算結果をキャッシュできないため，a'_{12} における実行時間の削減は小さい．したがって，$I \geq 30$ において，実行時間が増大する．

C2_MCR において，適切な置換優先度が得られない原因は，Spark がシャッフル実行時に計算結果をストレージに保持し，同一のシャッフル実行の一部を省略するためである．pMCR では，省略した処理を含めて実行時間を予測するため，適切な置換優先度が得られない．省略した処理を除いて，実行時間を推定することは今後の課題である．

7. まとめ

本稿では，米尾らが提案したキャッシュ指示自動化システム [3] を拡張し，実行時間の予測に基づく IMC の置換手法 MCR を提案した．拡張したシステムは 3 段階で構成される．まず，小規模な事前実行から RDD の依存グラフを取得する．次に，得た依存グラフを解析し，予測した高速化に対する寄与の大きな RDD を貪欲に選択することで，RDD の置換優先度リストを決定する．最後に，キャッシュの置換が発生する大規模な本実行において，置換優先度に基づいて IMC に存在するデータを置換する．

実験の結果，RDD の置換が頻繁に発生する状況において，提案手法は既存手法 MD と比較して，1.0〜1.5 倍の高速化を得た．さらに，開発者によりキャッシュ指示の最適化を施したプログラムと比較して，実行時間の増大は高々 4 %であり，最大で 1.1 倍の高速化を実現した．したがって，提案手法は IMC が不足する大規模なデータ処理をキャッシュ指示に関する試行錯誤なしで高速化できる．

今後の課題は，各 RDD 操作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; - Lightning-Fast Cluster Computing, `https://spark.apache.org/` (accessed 2019-6-24).
[2] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *Proc. 9th USENIX Conf. Networked Systems Design and Implementation (NSDI '12)*, pp. 1–14 (2012).
[3] 米尾謙史，置田真生，伊野文彦：Apache Spark におけるデータ依存グラフに基づくメモリ内キャッシュの指示および置換，情報処理学会研究報告, 2018-OS-142，p. 9 (2018).
[4] 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).
[5] Gottin, P., Dias, C., Costa, V., Souto, P. and Porto, R.: Automatic Caching Decision for Scientific Dataflow Execution in Apache Spark, *Proc. ACM SIGMOD Workshop on Algorithms and Systems for MapReduce (BeyondMR '18)*, pp. 1–2 (2018).
[6] 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).
[7] UC Irvine: The UCI Machine Learning Repository, `https://archive.ics.uci.edu/ml/` (accessed 2019-06-24).
[8] GroupLens Research: MovieLens, `https://grouplens.org/datasets/movielens/` (accessed 2019-06-24).