Vol.2018-OS-142 No.15
2018/2/28

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

米尾 謙史[1,a)]　置田 真生[1,b)]　伊野 文彦[1]

概要：本稿は Apache Spark の利便性向上を目的として，キャッシュ指示の自動化およびキャッシュ置換アルゴリズムの改善を提案する．Spark プログラムを高速化するためには，明示的なメモリ内キャッシュ指示による RDD の再計算の回避が必要である．しかし，明確な指針が存在せず，キャッシュ指示には試行錯誤が必要である．さらに，問題規模が増大すると LRU に基づいたキャッシュ置換が原因で，再計算が頻発し実行時間が増大する．そこで我々はまずデータ依存グラフの解析によるキャッシュ指示の自動挿入手法を提案する．本手法は再計算を回避するために必要最低限のキャッシュ指示を与える．次に再計算の頻発を防ぐためのキャッシュ置換アルゴリズムを提案する．本手法はデータ依存グラフをもとに再計算のコストが最小である RDD のキャッシュを優先的に破棄する．提案手法を機械学習ライブラリ MLlib に適用した．キャッシュ置換が発生しない場合，キャッシュ指示を手動で挿入する場合と自動挿入した場合の実行時間の差が高々 4 %であり，透過的なキャッシュ指示を実現した．また，キャッシュ指示の自動挿入手法と提案した置換アルゴリズムを組み合わせることで，問題規模の増大に伴う実行時間の増大を約 34 倍から約 1.2 倍に抑制した．

キーワード：RDD，ソフトウェアキャッシュ，ユーザ透過性，PC クラスタ

1. はじめに

Apache Spark（Spark）[1] は大規模データの高速処理を目的とした並列分散処理基盤である．Spark の特徴の 1 つとして，分散データセット Resilient Distribution Dataset（RDD）[2] を介した透過的な並列プログラミングがある．プログラマは RDD に対する変換操作の連なりとして処理を記述する．実際には Spark は変換操作を最適化し，ジョブと呼ばれる実行単位ごとに実行する．ジョブはタスクに分割され，PC クラスタ内の各ノードに分散される．処理が主記憶上で完結するため，大規模データの高速処理が可能である．

大規模データ処理の最適化のために RDD は実体データを持たず，操作内容と依存関係のみを保持する．実体データは RDD が必要になるときに依存関係を辿って計算される．原則として実体データは計算後に破棄されるため，同一の RDD が必要になるたびに同じ計算が発生する（再計算）．再計算を避けるためにはユーザプログラム上でプログラマが RDD に対してキャッシュ指示を与え，RDD の実体データを主記憶内のキャッシュ領域に格納する必要がある．プログラマは再計算の総時間を最小化するキャッシュ指示を与える RDD の組合せを選択する．

しかし，キャッシュ指示の明確な指針は明らかでない．Spark のプログラミングガイド [3] によれば，プログラム中で複数回必要になる RDD に対してキャッシュ指示をすればよい．しかし，実際にはキャッシュ指示が過剰となり最適化が阻害され，さらに置換が頻繁に発生するため性能が低下する．また，RDD が必要となる回数および再計算に要する時間を実行前に正確に見積もることは難しい．そのため，プログラマは再計算の総計算時間が最小となるキャッシュ指示の組み合わせを試行錯誤して決める必要がある．

さらに，問題の規模の増大により最低限のキャッシュ指示をしても置換の発生が回避できない場合，性能が低下する．キャッシュ領域を超えて RDD にキャッシュ指示を与えた場合，Last Recently Used（LRU）により実行時間削減の効果に関係なく，最も古い時刻に使用された RDD の実体データから置換される．その結果，実行時間が約 5.5 倍になる場合もある [4]．実行時間の増大を回避するために問題規模に応じてキャッシュ指示を削減する必要があるが，プログラマの試行錯誤を要する．

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) k-yoneo@ist.osaka-u.ac.jp
b) okita@ist.osaka-u.ac.jp

ⓒ 2018 Information Processing Society of Japan

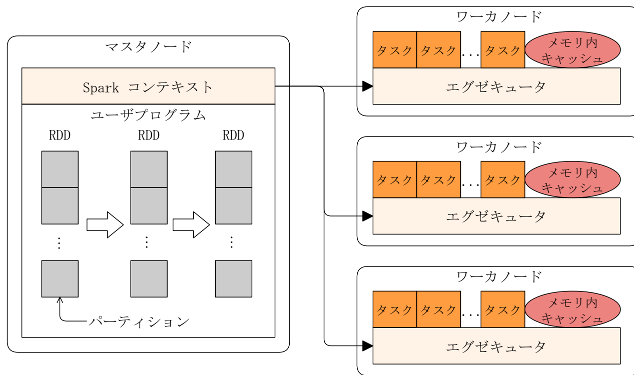

図 1 Spark における並列分散実行の概略

Duan ら [5] はメモリ内キャッシュの利便性を向上するために，キャッシュ指示の対象となる RDD を自動で決定するキャッシュ指示アルゴリズムを提案した．この手法は，1 つのジョブの実行中に 2 回以上必要となる RDD を特定し，その RDD の実体データを初回計算時に主記憶内のキャッシュ領域に格納する．しかし，異なるジョブ間で発生する再計算を回避できない．

そこで本手法では RDD の依存グラフを解析することで，キャッシュ指示の対象となる RDD の組み合わせを自動で求める手法を提案する．異なるジョブ間で発生する再計算を含むすべての再計算を回避するために最低限必要なキャッシュ指示を行う．キャッシュ置換が発生しない場合，すべての再計算の回避を保証する．さらに依存グラフにおける入力データからの距離が最も小さい RDD の実体データを優先的に置換するアルゴリズムを提案する．結果的に再計算に要する時間が大きい RDD を優先してキャッシュ領域に保存することで，実行時間を削減する．これらの手法をもとにキャッシュ指示自動化システムを開発する．提案システムは解析する依存グラフを得るために事前実行が必要だが，ユーザプログラムを改変する必要はない．

提案システムは機械学習のパラメータチューニングのように同一プログラムを繰り返し実行する場合に有用である．提案システムでは事前実行を 1 回行うことでキャッシュ指示を決定するため，プログラマはパラメータチューニングに専念できる．

以降，まず 2 節で Spark，3 節で問題点，および 4 節で提案手法について説明して，5 節で実験結果を示す．6 節で関連研究を紹介し，最後に 7 節で本稿をまとめる．

2. Apache Spark

Spark はマスタ・ワーカ型のフレームワークである．図 1 に Spark の構造を示す．プログラマが記述する部分はユーザプログラムのみであり，データの分散，タスク分割，並列実行および通信は Spark が暗黙的に行う．Spark コンテキスト（SC）は Spark のシェルとなるオブジェクトである．ユーザは SC を介して PC クラスタ上での並列分散実行を実現する．Spark はワーカノードにエグゼキュータを作成する．エグゼキュータはプロセスであり，タスクの実行およびキャッシュの管理を行う．

```
val input = sc.textFile("inputData.csv").map(入力ファイルを
        変換).cache()
val norm = input.map(ノルムを計算).cache()
val data = input.zip(norm).map(...)

val centers = init(data)                    /* 下部で定義 */
while(収束するまでループ) {
  val newCenters = data.mapPartitions(新中心座標を計算).
        reduceByKey(...).collectAsMap()
  ...                  /* newCentersに基づきcentersを更新 */
}
val WSSSE = data.map(centersに基づき二乗和誤差を計算).sum()

private def init(data) {
  val centers = Seq[...]()
  val costs = data.map(座標のコストを設定)
  val sample = data.takeSample(初期座標をランダム抽出).toSeq
  val newCenters = sample.toDense
  centers ++= newCenters
  while(数ステップ繰り返す) {              /* 今回の例では2回 */
    val preCosts = costs
    costs = data.zip(preCosts).map(座標のコストを再設定).
        cache()
    val sumCost = costs.aggregate(...)
    preCosts.unpersist()
    val chosen = data.zip(costs).mapPartitionsWithIndex(新
        中心座標を計算).collect()
    newCenters = chosen.map(_.toDense)
    centers ++= newCenters
  }
  costs.unpersist()
  val weightMap = data.flatMap(各中心座標の候補を重み付け).
        reduceByKey(_ + _).collectAsMap()

  return ...    /* weightMapからk個の中心座標を決定して返す */
}
```

図 2 MLlib[6] 収録の k 平均法の擬似コード（Scala 言語．赤色は RDD，水色は変換，青色はアクション）

2.1 RDD（Resilient Distribution Dataset）

RDD はパーティションに区分された集合を表す Spark 固有のデータ構造である．Spark では，RDD R を抽象的な形式 (f, S)，すなわち R を返す関数 f とその入力 S の組として扱う．また，以降では R の実体，すなわち R が表す集合の全要素データを $/R/$ と表す．R は原則として $/R/$ を保持せず，Spark は $/R/$ を必要に応じて計算する．

RDD に対する計算操作は変換およびアクションの 2 つに分類できる．変換 $R \mapsto S$ は RDD R を操作して新たな RDD S を得る．なお，変換はストレージ上のデータおよび変数から RDD を生成する操作を含む．また，入力は最大 2 つの RDD をとりうる．一方，アクション $R \mapsto x$ は

ⓒ 2018 Information Processing Society of Japan

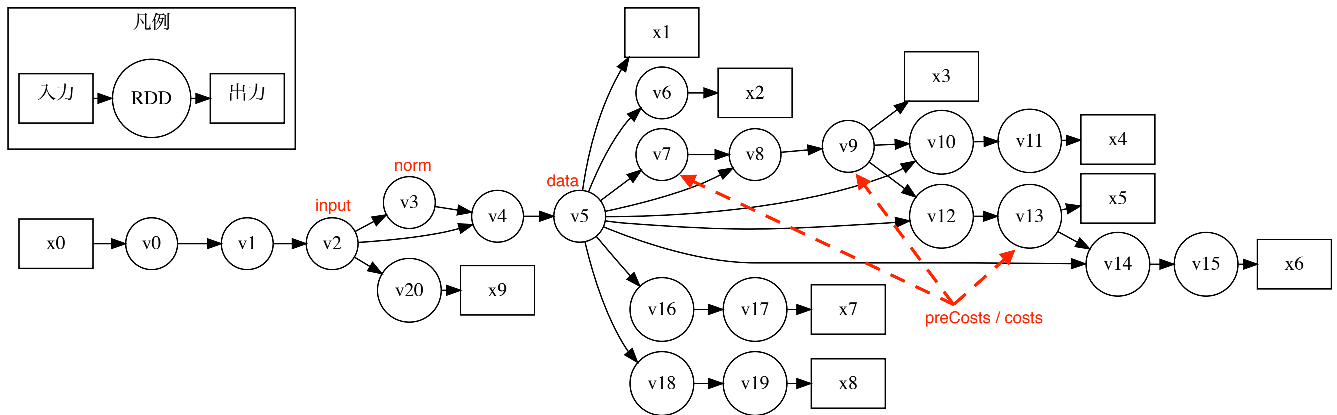

図 3　図 2 の実行を表す依存グラフ（赤字は対応する変数名）

RDD R の実体データを計算し，結果 x を RDD 以外の形式で返すかストレージ上に保存する．RDD 操作はパーティションごとに独立したタスクとなり，Spark が各タスクをエグゼキュータに分散配置する．

ユーザプログラムの実行は，RDD の依存グラフとして表現できる．依存グラフ $G = (V, E)$ は有向非循環グラフであり，頂点集合 V の要素は RDD，ストレージ上のデータおよび変数である．図 2 の依存グラフを図 3 に示す．入次数が 0 の頂点（入力点）あるいは出次数が 0 の頂点（出力点）はストレージ上のデータもしくは変数であり，それ以外の頂点はすべて RDD である．辺 $u \to v$ は頂点 v の計算が頂点 u の計算に依存することを意味する．辺のうち出力点の入次辺はアクションの実行に基づく依存であり，それ以外の辺はすべて変換の実行に基づく依存である．G 上で出次数が 2 以上の頂点を分岐点，入次数が 2 以上の頂点を合流点と呼ぶ．

Spark はユーザプログラムを実行しながら動的に G を構築する．G の初期状態は空であり，変換 $R \mapsto S$ が呼び出されるたびに，頂点 R，S および辺 $R \to S$ を G に追加する．このとき，変換の計算そのものは実行されず，その実行はのちのアクション呼び出しまで遅延される．

2.2　ジョブ

アクション $R \mapsto x$ が呼び出されると，SC は結果 x の計算に必要な情報を RDD R をもとに導出してジョブ $J(R \mapsto x)$ を作成する．Spark はジョブをプログラムの実行単位とする．すなわち，プログラムの実行はすべてのアクションに対するジョブの集合である．

ジョブ $J(R \mapsto x)$ の実行には，R の系譜 $L(R)$ が必要である．系譜 $L(R)$ は R の計算手順，すなわち G 上で R を終端とする経路であり，辺の列として定義できる．系譜 $L(R)$ は G を逆探索することで導出する．R から入次辺を逆向きに辿り，入力点に到達するまで深さ優先順で再帰的に繰り返す．例えば，図 3 におけるジョブ $J(v_5 \mapsto x_1)$ に必要な系譜 $L(v_5)$ は以下となる．

$$
\begin{aligned}
L(v_5) = (&v_4 \to v_5, v_3 \to v_4, \\
&v_2 \to v_3, v_1 \to v_2, v_0 \to v_1, x_0 \to v_0, \\
&v_2 \to v_4, v_1 \to v_2, v_0 \to v_1, x_0 \to v_0)
\end{aligned}
$$

ここで，系譜 $L(R)$ は経路の部分的な重複を含む可能性がある点に注意が必要である．G に分岐があると，逆探索の過程で一度探索した経路を再度探索する．Spark はこの重複を自動的に排除しない．上記の $L(v_5)$ の例では，$(v_1 \to v_2, v_0 \to v_1, x_0 \to v_0)$ の部分，すなわち $L(v_2)$ が重複する．

ジョブ $J(R \mapsto x)$ を作成したのち，SC は $L(R)$ をもとにタスク生成およびスケジューリングを行い，エグゼキュータにタスクを割り当てる [2]．このとき，Spark は高速化のために処理を最適化する．例えば，複数の変換をパーティション単位で一括化することによる局所性の向上や，最終的に利用されないと判断したパーティションに対する処理の省略がある．エグゼキュータは，スケジュールにしたがって RDD の実体データを順に計算し，必要に応じて互いに通信を行い，結果 x を得る．

2.2.1　再計算

Spark はジョブを計算する資源を確保するために，RDD の実体データを計算後に破棄する．そのため一度実体データを計算した RDD R に関わらず，必要となるときに $/R/$ の計算を要する．これを R の再計算と呼び，その再計算に要する時間を再計算コストと呼ぶ．

再計算は同一ジョブ内で発生する場合と異なるジョブ間で発生する場合がある．同一ジョブ内で発生する理由は，2.2 節で挙げた $L(v_5)$ の例のように，系譜内の経路の重複による．一方，異なるジョブ間で発生する理由は，異なるアクションが同じ RDD を必要とするためである．例えば，図 3 ではアクション $v_{20} \mapsto x_9$ を除くすべてのアクションが v_5 を必要とし，そのたびに v_5 の計算が発生する．

2.3　RDD へのキャッシュ指示

再計算を明示的に回避する手法として，Spark には RDD

のメモリ内キャッシュ機能がある．RDD R の実体データ $/R/$ をエグゼキュータのメモリ内キャッシュ領域（IMC）に保存することで，$/R/$ の再利用を可能にする．具体的には，ジョブの実行中に R を計算するとき，$/R/$ のすべての要素が IMC 上に存在すれば，Spark は $L(R)$ の計算を省略する．結果，R の再計算コストは 0 となる．

RDD R をキャッシュするためには，プログラム中で R に対して明示的なキャッシュ指示（`persist()` および `cache()` メソッド）を与える必要がある．R にキャッシュ指示があることを述語 $C(R)$ で表す．エグゼキュータは $/R/$ を計算したのち，$C(R)$ が真であれば $/R/$ を IMC に保存する．

以降では，$C(R)$ が真となる G 上の頂点をキャッシュ点と呼ぶ．例えば，図 3 の例では v_2，v_3，v_9 および v_{13} がキャッシュ点である．

2.3.1 キャッシュの置換

IMC の容量を超えて RDD の実体データを保存する場合，LRU に基づいた置換が発生する．Spark は RDD の実体データを各エグゼキュータに分散配置する．IMC の容量の違いや処理の過程により各エグゼキュータに保存するデータに偏りが発生するため，Spark は置換をエグゼキュータごとに行う．

また，プログラマはキャッシュした実体データを明示的に破棄できる．ユーザプログラムが RDD R に対するキャッシュ破棄命令（`unpersist()` メソッド）を呼び出すときに，エグゼキュータは LRU の順番に関係なく IMC から $/R/$ を破棄する．例えば，図 2 の例では v_9 および v_{13} に対して，それぞれ不要となる箇所にキャッシュ破棄命令が記述されている．

3. キャッシュ指示の問題点

本稿では，メモリ内キャッシュ機構における問題のうち，次の 2 点に注目する．

(P1) 試行錯誤の必要性
(P2) 問題の大規模化に伴うキャッシュヒット率の低下

3.1 試行錯誤の必要性

過剰なキャッシュ指示は実行時間の増大を引き起こす．理由の 1 つは，IMC の圧迫に伴うキャッシュ置換の頻発である．その結果，メモリ内キャッシュのヒット率が低下し，再計算を回避できない．さらに，実体データの格納による最適化の阻害がある．省略されるべき中間結果の RDD R にキャッシュ指示が存在すると，$/R/$ を計算する必要が生じ，最適化を阻害する．

したがって，キャッシュ指示を与える RDD をプログラマが取捨選択する必要がある．一般に，一定の IMC 容量下での実行時間が最短となるキャッシュ指示の組み合わせ（最適なキャッシュ指示）を求めることは難しい．この問題は再計算コストを価値とみなしたナップサック問題に帰着できる．しかし，キャッシュ指示の組み合わせによって再計算コストが変化するため，問題はより複雑である．例えば図 3 において，何もキャッシュしない場合の v_3 の再計算コストは $L(v_3)$ の処理時間だが，v_2 をキャッシュした場合のそれは $v_2 \mapsto v_3$ の処理時間のみである．

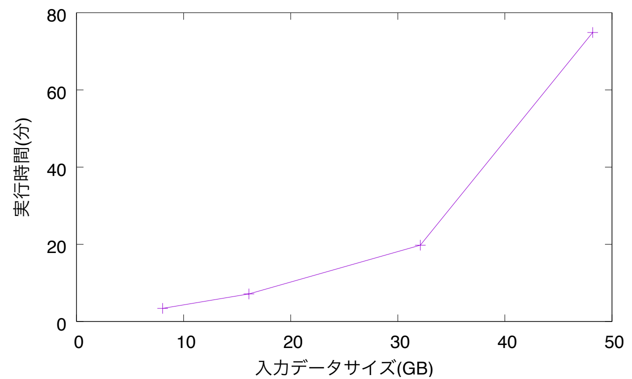

図 4 図 2 の実行時間（分）

さらに，最適なキャッシュ指示を求めるために必要な情報をプログラマは正確に把握できない．必要な情報は，(1)IMC の容量，(2) 各 RDD の実体データを計算する回数，(3) 各 RDD の実体データの容量および (4) 各変換に要する時間である．このうち (1) 以外の情報はプログラムを実行するまで計測できない．さらに，(3) および (4) は，最適化のための省略が原因で正確な計測が難しい．

したがって，プログラマは経験的あるいは実験的にキャッシュ指示の組み合わせを選択する．プログラムの実行とキャッシュ指示の変更を繰り返して，実行時間の比較的短い組み合わせを探索する方法が一般的である．

3.2 問題の大規模化によるヒット率の低下

特定のプログラムに対して問題規模を増大するとき，一定規模を超えた場合に実行時間が非線形に増大する．図 2 に対して入力データ量を変化したときの実行時間を図 4 に示す．入力データ量が 48 GB のとき，32 GB の場合と比較して 3 倍以上の処理時間を要する．

この原因の 1 つは，キャッシュの交互追い出しが発生し，ヒット率が低下するためである．交互追い出しとは，キャッシュ指示が与えられた RDD の組に対して，一方をキャッシュするためにもう一方を破棄する状況を繰り返すことである．交互追い出しが発生し得る条件は，頻繁に必要となる系譜上にキャッシュ点が複数存在し，それらの実体データの総データ量が IMC の容量を超過することである．例えば，図 3 の例では，前述のように v_2，v_3 および v_9 にキャッシュ指示がある．ジョブ $J(v_{13} \mapsto x_5)$ を実行する場合，系譜 $L(v_{13})$ 上には v_2 のみを含む経路，(v_2, v_3) を含む経路，(v_2, v_9) を含む経路および (v_2, v_3, v_9) を含む経路が 1 つずつ存在する．IMC の容量が $/v_2/$，$/v_3/$ およ

ⓒ 2018 Information Processing Society of Japan

び $/v_9/$ の合計量より小さいと，計算順によっては v_2 および v_9 が相互追い出しの関係となり，それらの再計算が複数回発生する．

4. 提案手法

本手法の目的は実行時間削減のためのキャッシュ指示という観点における，Spark のユーザ透過性向上である．まず (P1) を解決するために，依存グラフ G 上でキャッシュ指示を与えるべき頂点集合 W を明らかにする．IMC 容量が RDD の実体データ量に対して十分大きい場合には，W のすべてにキャッシュ指示を与えることで再計算の発生を回避できる．次に (P2) を解決するために，キャッシュの置換が発生する場合に，交互追い出しを避けるための置換アルゴリズムを提案する．最後に，これらの手法に基づいて Spark を拡張したキャッシュ指示自動化システムを提案する．

4.1 再計算を回避するための必要十分条件

キャッシュの置換を考慮しない場合，すべての再計算を回避するための必要十分条件は，依存グラフ G 上の分岐点すべてにキャッシュ指示を与えることである．ここで，頂点 v の出次隣接頂点の集合および出次数をそれぞれ $N(v)$ および $|N(v)|$ で表す．すなわち，

$$N(v) = \{u \in V \mid (v \to u) \in E\}$$

である．また，v の計算回数を返す関数を $F(v)$ とする．次の 2 つの命題について，P と Q が同値であることを以下に示す．

$$P : \forall v \in V \ (|N(v)| \geq 2 \Rightarrow C(v) = \text{true})$$

$$Q : \forall v \in V \ F(v) = 1$$

証明． まず $P \Rightarrow Q$ を示す．$C(v)$ が偽の場合，$F(v)$ はプログラム実行中に G を逆探索するときの v を通過する回数に等しい．すなわち，次の式 (1) が成り立つ．

$$F(v) = \begin{cases} \sum_{u \in N(v)} F(u), & \text{if } |N(v)| > 0, \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

一方 $C(v)$ が真であれば，Spark は $/v/$ を 1 回目のみ計算し，以降は計算を省略するため，実行中は常に $F(v) = 1$ となる．ここで P が成り立つ場合，$|N(v)| \geq 2$ であるすべての $v \in V$ において $F(v) = 1$．また式 (1) より，$|N(v)| = 1$ となるすべての $v \in V$ において $F(v) = F(u)$ $(N(v) = \{u\})$ である．したがって，すべての $v \in V$ について $F(v)$ は 1 あるいは隣接頂点と同じ値となるため，Q が成立し，$P \Rightarrow Q$ である．

次に $Q \Rightarrow P$ を示す．Q が真のとき，$|N(v)| \geq 2$ かつ $C(v)$ が偽となる頂点 v が存在すると仮定する．式 (1) より，$F(u)$ は正の整数であるので，$F(v) \geq 1$ となり Q に矛盾する．したがって，背理法により $Q \Rightarrow P$ である．以上より，P と Q は同値である． □

アルゴリズム 1 キャッシュ置換アルゴリズム MD

```
Input:
    r:IMC に格納を試みる RDD;
    D:IMC に実体データが存在する RDD の集合;
    F:IMC の空き容量;
Output:
    D;
 1: 系譜長の昇順で D をソート;
 2: Δ ← ∅;
 3: for all d ∈ D do
 4:     if |L(r)| ≤ |L(d)| then
 5:         return D;                    ▷ 置換せずに返す
 6:     end if
 7:     Δ ← Δ + d;
 8:     F ← F + sizeof(/d/);
 9:     if F ≥ sizeof(/r/) then
10:         return (D − Δ + r);          ▷ 置換した結果を返す
11:     end if
12: end for
13: return D;
```

図 3 の例では，分岐点は v_2，v_5，v_9 および v_{13} となる．キャッシュの置換を考慮しない場合は，これらすべての実体データを IMC に格納するとき，すべての再計算の発生を回避できる．

4.2 置換アルゴリズムの改善

交互追い出しが発生する原因は，同じ RDD を再度参照する間隔よりキャッシュ置換の間隔が小さく，LRU が FIFO と類似した挙動になるためである．これを解決するためには，参照時間以外の優先度に基づいて置換する RDD の実体データを決定する必要がある．

本手法では，RDD の再計算コストに注目する．再計算コストが大きい RDD の実体データを優先して IMC に残すことで，実行時間を削減する．しかし，変換ごとの実行時間を正確に計測できないので（3.1 節），RDD R の再計算コストの正確な計測は難しい．そこで，系譜 $L(R)$ の長さ $|L(R)|$ を用いて置換する RDD の実体データを決定する．R の再計算コストと $|L(R)|$ は正の相関を持つ．

アルゴリズム 1 に提案するキャッシュ置換アルゴリズム MD を示す．RDD r をキャッシュするため，系譜長の小さい RDD から順に IMC 上の実体データを破棄する．ただし，系譜長が $|L(r)|$ より小さい RDD の実データをすべて破棄しても $/r/$ の格納に必要な容量を確保できない場合は，置換せず r のキャッシュを諦める (5 行目および 13 行目)．例えば，図 3 の例において $/v_2/$ および $/v_5/$ が IMC に存在する状態で置換が発生した場合，系譜長の最も小さい $/v_2/$ を破棄する．なお，実際のキャッシュ置換はパーティション単位で行う．

単純に MD を用いる場合，最長の系譜を持つ RDD の

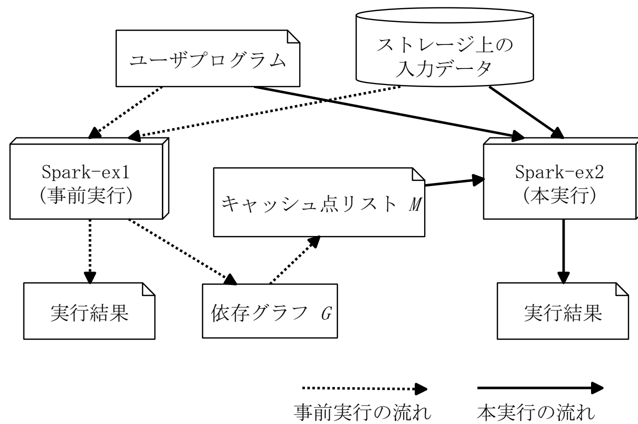

図 5　提案システム利用の流れ

実データを IMC に固定し，プログラム終了まで破棄しない．そこで LRU と組み合わせて，一定時間参照していない RDD の実体データを IMC から破棄する．具体的には，直近 l 回のジョブ実行中に参照しなかった RDD の実体データを，$l+1$ 回目のジョブ実行開始時に破棄する．

4.3　提案システムの概要

提案システムは，既存の Spark を拡張し，$P \Leftrightarrow Q$ を利用してキャッシュ点を自動的に決定する．具体的には，依存グラフ G 上の分岐点に，動的にキャッシュ指示を付与する．そのためには G 全体を必要とする．しかし，あるジョブの実行時にはそれまでに構築された部分グラフしか得られない．そこで本手法では，事前実行により G を取得する．

提案システムの利用の流れを図 5 に示す．事前実行の高速化のためには，実際のデータ処理を省略して依存グラフ G のみを得ることが望ましい．しかし，処理の省略によって分岐および繰り返しの挙動が変化し，G 上の出次数が変化する可能性がある．本手法では正確な G を得るために事前実行においても実際にデータ処理を行う．したがって，事前実行の実行結果もユーザは利用可能である．提案手法は 2 回目以降の本実行に効果がある．

提案システムの構成は 2 つに分類できる．事前実行のための Spark-ex1 および本実行のための Spark-ex2 である．ともに Spark のフレームワークを拡張したもので，提案手法の動作はユーザに隠蔽される．したがって，提案システムを利用するためにユーザプログラムの変更は不要である．なお，これらの拡張は実際には 1 つのシステムに統合されており，実行時のオプションで切り換えて使用する．

Spark-ex1 の役割は，G の取得およびキャッシュ点の特定である．SC の機能を拡張し，実行中に G を記録する．プログラムの実行後，SC は G を全探索して $|N(v)| \geq 2$ となる頂点 v のリスト M を作成し，出力する．

Spark-ex2 の役割は，キャッシュ指示の動的挿入および MD の適用である．前者は SC を拡張して実現する．ジョブ $J(R \mapsto x)$ を実行するとき，$A \in (M \cup L(R))$ となる RDD A に対してキャッシュ指示を付与する．後者はエグゼキュータを拡張し，既存のキャッシュ置換アルゴリズム LRU を MD で代替することで実現する．

なお，Spark-ex2 はプログラマによるキャッシュ指示を優先する．すなわち，既にプログラム上で RDD R にキャッシュ指示が存在する場合，R が分岐点でなくともそのキャッシュ指示を有効とする．

表 1　実験環境（PC クラスタ）

ノード数	6 台
CPU	Intel Xeon E3-1230 3.2 GHz
主記憶	8 GB（ノードあたり）
HDD	1 TB SATA HDD
ネットワーク	Gigabit Ethernet
OS	CentOS 5.6
Spark	Spark 2.0.0，Standalone モード
IMC 容量	3 GB（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] (449 MB)
k 平均法	HIGGS[7] (8.03 GB)
交互最小二乗法	MovieLens Latest Datasets[8] (0.7 GB)

5.　評価実験

プログラム実行時間および透過性の観点から，提案手法を評価する．実験には 6 台の計算機で構成する PC クラスタを用いた（表 1）．実験対象には，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（表 2）．なお，MD における破棄の閾値 l は，実験的に求めた値 $l = 2$ を使用した．

アプリケーションごとのキャッシュ点数を表 3 に示す．線形回帰は，他と比較して分岐点と合流点が少ない単純な G をもつ．プログラム中でアクションを 103 回呼ぶ．G 上に 2 個存在する分岐点のうち 1 つは出次数が 100，もう 1 つは 4 である．k 平均法は図 3 のような G をもち，収束までの繰り返し数 n に応じて G が変化する．プログラム中でアクションを 80 回呼ぶ．G 上に 7 個存在する分岐点のうち 1 つは出次数が 80 〜 101，それ以外は 3 である．交互最小二乗法（ALS）は依存グラフにおける合流点が比較的多い．プログラム中でアクションを 12 回呼ぶ．G 上に 10 個存在する分岐点のうち 4 つは出次数が約 22，それ以外は高々 5 である．

5.1　実験結果

提案手法の効果を確認するため，キャッシュ指示の自動挿入および MD を段階的に組み合わせて実行時間を計測した．実験パターンを表 4 に示す．C1 は，提案手法を適用せずに既存の Spark 上でアプリケーションを実行する，オ

ⓒ 2018 Information Processing Society of Japan

表 3　依存グラフの特徴

	$\|V\|$	系譜長	合流点	キャッシュ点		
				$\|C_d\|$	$\|C_b\|$	$\|C_d \cup C_b\|$
線形回帰	609	5 – 10	0	1	2	2
k 平均法	245	5 – 20	11	7	7	8
ALS	430	4 – 384	405	7	10	10

$|C_d|$：開発者が設定したキャッシュ点の集合
$|C_b|$：提案手法が自動設定するキャッシュ点（G の分岐点）の集合

表 4　実験パターン一覧

番号	キャッシュ指示		キャッシュ置換
	開発者の指示	自動挿入	
C1	あり	なし	LRU
C2	なし	あり	LRU
C3	あり	あり	LRU
C1+	あり	なし	MD
C2+	なし	あり	MD
C3+	あり	あり	MD

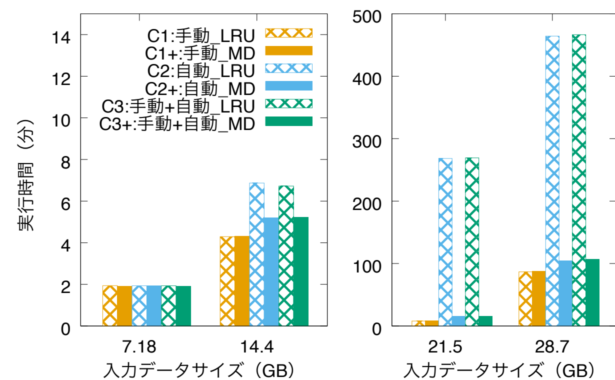

図 6　線形回帰における実行時間（分）

リジナル実行に相当する．開発者の指示とは，MLlib の開発者がプログラムに手作業で挿入したキャッシュ指示である．C2 および C2+は，開発者が挿入したキャッシュ指示を無視し，提案手法により自動挿入されたキャッシュ指示にのみ従う．アプリケーションごとの計測結果を図 6 〜 8 に示す．キャッシュ置換の発生に伴う実行時間の変化を確認するために，入力データ量 B を増大しながら実行時間を計測した．B の変更は，入力データセットの内容を複製して調節した．

5.2　キャッシュ点自動決定手法の評価

まず，自動決定したキャッシュ点の必要性を評価する．入力データ量 B が最小の場合の C1 および C2 を比較すると，すべてのアプリケーションにおいて C2 の実行時間は C1 の実行時間以下である．したがって，IMC 容量が十分大きい場合，提案手法は手動によるキャッシュ指示と同等以上の効果を得られた．

次に，自動決定したキャッシュ点の十分性を評価する．B が最小の場合の C2 および C3 を比較すると，すべてのアプリケーションにおいて実行時間に変化はない．したがって，IMC 容量が十分大きい場合，自動決定したキャッシュ点だけで十分である．

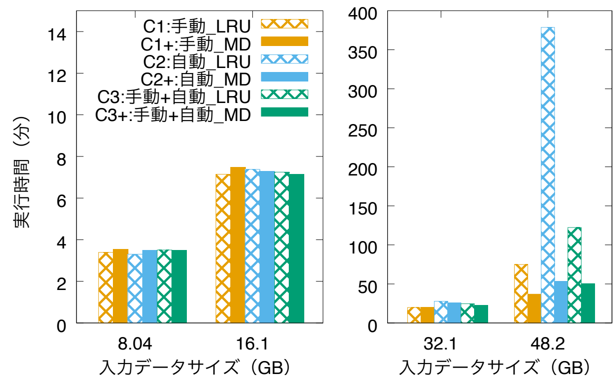

図 7　k 平均法における実行時間（分）

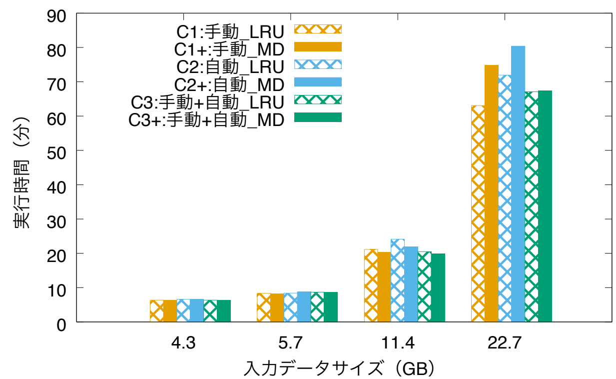

図 8　ALS における実行時間（分）

しかし，B が増大すると，C2 の実行時間は C1 と比較して増大する可能性がある．とくに k 平均法の $B = 48$ GB の場合，約 5.1 倍に増大する．したがって，IMC 容量が B に対して相対的に小さい場合には，キャッシュ指示の自動挿入を単独で用いても有用ではない．

5.2.1　考察

k 平均法では $B \geq 32$ GB のときに，C1 および C3 と比較して C2 の実行時間が増大する．この増大は交互追い出しによるキャッシュミスの発生に起因する．C1 および C3 においてキャッシュヒット率が C2 より高い理由は，図 3 における頂点 v_3 をキャッシュ点に設定するためである．v_3 は C_d に含まれ，かつ C_b に含まれない．$/v_2/$ をすべて IMC に保存できる場合，同一ジョブ内で 2 回発生する v_2 の再計算をすべて回避するため，v_3 をキャッシュしても実行時間は変化しない．しかし，$/v_2/$ をすべて IMC に保存できない場合，v_2 の再計算が 2 回発生する．v_3 をキャッシュすることで，v_2 の再計算のうち 1 回を回避できるため，実行時間を削減できる．$/v_3/$ は $/v_2/$ と比較してデータ量が比較的小さい．$B = 8.04$ GB のとき，$/v_3/$ のデータ量が 83 MB であり，$/v_2/$ のデータ量が 2.7 GB である．そのため，v_3 は v_2 より多くのパーティションを IMC に保存できる．

また，k 平均法の $B = 48$ GB のとき，C3 は C1 よりも実行時間が長い．この理由は v_5 のキャッシュにある．C3 は v_5 をキャッシュすることで IMC の空き容量を減少する．C1 では v_5 をキャッシュしない．C3 の実行時間増大の原因は IMC に保存可能な $/v_3/$ のデータ量が C1 と比較して少ないことである．これらの k 平均法の結果から，キャッシュ置換が発生する状況では RDD の実体データ量を考慮してキャッシュ点を定めることが望ましい．

線形回帰では $B \geq 14$ GB のとき，C1 と比較して C2 および C3 の実行時間が増大する．この理由も k 平均法と同様に交互追い出しにある．C1 はキャッシュ点が 1 箇所であるため，キャッシュ置換は発生しない．

ALS では，C1 と比較して C2 および C3 の実行時間が増大する．実験結果の傾向が同じである線形回帰と比較して，問題規模に関わらず C1 および C2，C3 で実行時間に大きな差が見られない．

提案手法の課題は，キャッシュ指示を行うために事前実行が必要な点である．キャッシュ点を決定するために事前実行が必要だが，事前実行時にもキャッシュ指示を与えることが望ましい．キャッシュ点を設定せずにプログラムを実行すると，すべての再計算が発生し実用的な時間内に実行が完了しない場合がある．例えば，線形回帰で $B \geq 28.7$ GB の場合には実行に 10 時間を要した．これを解決するため，事前実行時にキャッシュ指示を自動で行う既存手法 [5] と組み合わせる方法を検討している．

5.3 置換アルゴリズム MD の評価

置換アルゴリズムだけを変更した実験パターンの組 (C1,C1+)，(C2,C2+) および (C3,C3+) をそれぞれ比較すると，ALS の $B = 0.7$ GB を除くすべての場合で実行時間を削減した．特に，k 平均法の $B = 48$ GB の場合に実行時間の削減率が大きい．したがって，MD は B と比較して IMC 容量が比較的大きい場合に有用である．

また，キャッシュの自動挿入と MD の組合せに注目する．k 平均法の $B = 48$ GB の場合，C1，C2 および C2+ を比較すると，MD と組み合わせることでキャッシュの自動挿入による実行時間の増大を回避できた．したがって，問題規模に関わらず高性能なキャッシュ指示の自動化を実現するために MD が必要である．

5.3.1 考察

k 平均法では交互追い出しによるキャッシュミスをすべて回避し，実行時間を削減した．C1+の実行時間が比較的短いのは，他のパターンと比較して $/v_3/$ を優先的に IMC に保存するためである．C1 および C1+を除き，他のパターンでは $/v_3/$ より $/v_5/$ を優先して IMC に保存する．$/v_3/$ は $/v_5/$ と比較してデータ量が比較的小さい．$B = 8.04$ GB のとき，$|/v_3/| = 83$ MB であり，$|/v_5/| = 2.9$ GB である．そのため $/v_3/$ は $/v_5/$ よりも多くのパーティションを IMC に保存できる．したがって，問題規模が比較的大きい場合は v_3 をキャッシュすることで実行時間を削減できる．

線形回帰では，$B \geq 14.4$ GB のとき，キャッシュの置換が発生する．このとき，置換アルゴリズム MD を適用することで交互追い出しによるキャッシュミスを解消でき，実行時間を削減できた．C2+および C3+よりも C1+の実行時間の方が短い．C1+および C2+，C3+では実質キャッシュする RDD は 1 つのみである．C1+でキャッシュする RDD を v_a，C2+および C3+でキャッシュする RDD を v_b と呼ぶ．v_a より v_b の方が系譜長が大きいが，$/v_a/$ のデータ量が $/v_b/$ よりも大きい．そのため，v_a の方がより多くのパーティションをキャッシュでき，C1+の実行時間の方が短い．

ALS では．置換アルゴリズム MD を適用しても実行時間を削減できなかった．特に $B \geq 22.7$ GB のとき，C1 および C2, C3 で実行時間が最大で約 19 %増大する．$B \geq 11.4$ GB のとき，キャッシュの置換が発生する．実行時間の増大の原因に交互追い出し以外の原因があるために，置換アルゴリズム MD では実行時間を削減できなかったと考えられるが，原因は特定できていない．

評価実験の結果，RDD の実体データのデータ量も考慮した置換が望ましいという知見を得た．また，以下の条件を満たす場合，提案手法よりも効果的なキャッシュ点が存在する場合があることが分かった．

- 問題規模が比較的大きい
- 依存グラフに分岐点および合流点が存在する
- 分岐点から合流点までの各経路上に実体データ量が比較的小さい RDD s が存在する

s をキャッシュ点とするキャッシュ点自動決定手法および実体データ量に基づくキャッシュ置換アルゴリズムは今後の課題である．

6. 関連研究

Spark におけるメモリ内キャッシュ指示の自動化に関する研究に Duan ら [5] の手法がある．彼らはキャッシュ指示を与える RDD を動的に決定するキャッシュ指示アルゴリズム Selection Algorithm と，LRU を拡張した置換アルゴリズム Weight Replacement Algorithm を提案した．前者は 1 つのジョブの実行に必要な RDD の系譜を探索して 2 回以上必要となる RDD を特定し，その RDD の実体データを初回計算時に IMC に格納する．後者は同様に 1 つのジョブの系譜と実行時情報をもとに，RDD の使用回数，計算コストおよび実体データ量を考慮して置換する IMC 上の RDD の実体データを決定する．提案手法と比較すると，Duan らの手法は事前実行を必要としないため 1 度しか実行しないプログラムに対しても有用である．しかし，ジョブ単位で処理が完結するため，複数のジョブ間で発生する RDD の再計算を回避できない．一方，提案手法は事前実

行によって得たプログラム全体の G をもとに解析するため，ジョブ間の再計算を回避するキャッシュ指示が可能である．

また，Spark のキャッシュ機構における永続化レベル [3] を利用してプログラムを高速化する研究がある [9], [10]．Koliopoulos ら [9] は実行環境に応じて RDD の実体データを格納する記憶階層（主記憶，ディスク，あるいはその両方）を自動決定するアルゴリズムを提案し，25 %の実行時間削減を実現した．Xu ら [10] は分散メモリ管理システム Neutrino を提案した．既存の Spark では RDD のすべてのパーティションが同じ永続化レベルで保存される．そのため，各エグゼキュータの主記憶の使用状況を考慮できない．Neutrino は依存グラフ G および各ワーカノードの主記憶の使用状況に基づいてパーティションごとに永続化レベルを決定する．Ho らは Neutrino の試作型を Spark に導入し，MLlib に適用した結果最大で 70 %の性能向上を実現した．提案手法の目的がキャッシュの対象となる RDD の選択の自動化にある一方，これら両手法の目的はキャッシュを格納する装置選択の自動化にある．

7. まとめ

本稿は Spark の利便性向上を目的として，Spark を拡張したキャッシュ指示自動化システムを開発した．このシステムは 2 つの機能から成る．まずはデータ依存グラフの解析によるキャッシュ指示の自動挿入である．再計算を回避するために必要最低限のキャッシュ指示を与える．依存グラフを得るために事前実行が必要となるが，ユーザプログラムの改変は必要ない．次に，再計算の頻発を防ぐための置換アルゴリズム MD である．既存の置換アルゴリズムを MD に置き換えることで，問題規模を増大したときのキャッシュヒット率低下を解決する．

本稿ではプログラム実行時間および透過性の観点から提案手法を評価した．Spark が提供する機械学習ライブラリ MLlib で実験した結果，キャッシュ置換が発生しないとき，キャッシュ指示を手動で挿入する場合と自動挿入する場合の実行時間の差が高々 4 %であり，透過的なキャッシュ指示を実現した．問題規模の増大に伴い，キャッシュ指示の自動挿入のみでは実行時間が最大で約 34 倍に増大した．しかし，キャッシュ指示の自動挿入手法と置換アルゴリズム MD を組み合わせることで，実行時間の増大を約 1.2 倍に抑制した．

今後の課題として，事前実行が不要なキャッシュ点自動決定手法およびデータ量を考慮した置換アルゴリズムの開発を目指す．

謝辞 本研究の一部は，科学研究費補助金（基盤研究（A）JP15H01687）の支援による．

参考文献

[1] The Apache Software Foundation: Apache Spark; - Lightning-Fast Cluster Computing, `https://spark.apache.org/` (accessed 2017-12-31).

[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] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M.: *Learning Spark: lightning-fast big data analysis*, O'Reilly Media. (2015).

[4] 米尾謙史，置田真生，萩原兼一：並列分散処理基盤 Apache Spark における系譜つき中間データに対する効果的なメモリ内キャッシュ指示の検討，2016 年ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集 (HPCS'16)，p. 55 (2016).

[5] 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).

[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, *The Journal of 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 2018-01-11).

[8] GroupLens Research: MovieLens, `https://grouplens.org/datasets/movielens/` (accessed 2018-01-09).

[9] Koliopoulos, A.-K., Yiapanis, P., Tekiner, F., Nenadic, G. and Keane, J.: Towards Automatic Memory Tuning for In-Memory Big Data Analytics in Clusters, *Proc. 5th Int'l Congress Big Data (BigData Congress'16)*, pp. 353–356 (2016).

[10] Xu, E., Saxena, M. and Chiu, L.: Neutrino: Revisiting Memory Caching for Iterative Data Analytics., *Proc. 8th USENIX Workshop Hot Topics in Storage and File Systems (HotStorage'16)*, pp. 1–5 (2016).