社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Spark RDDの入出力性能の高速化に関する検討

張　凱輝[†,††]　谷村　勇輔[††,†]　中田　秀基[††,†]　小川　宏高[††,†]

† 筑波大学　〒305–8577 茨城県つくば市天王台 1-1-1
†† 産業技術総合研究所　〒305–8560 茨城県つくば市梅園 1-1-1
E-mail: †{neilyo.chou,yusuke.tanimura,hide-nakada,h-ogawa}@aist.go.jp

あらまし　Spark は機械学習やデータマイニングなどの反復計算を高速に実行できる並列データ処理フレームワークである．RDD（Resilient Distributed Dataset）と呼ばれる仕組みを利用してインメモリの並列処理や耐障害性の確保を実現したり，中間データをキャッシュして再利用可能にしている点に特徴がある．扱うデータが大きくメモリ容量が不足する場合には，一部または全部のデータを処理ノードのディスクに置いて処理を行うことも可能である．しかし，ディスクを用いることにより，Spark アプリケーションの実行性能が低下する可能性がある上，このディスク利用の有無をユーザが指示しないといけない問題がある．そこで本研究では，ディスク利用時の Spark アプリケーションの実行性能を調査し，RDD キャッシュのストレージレベルや性能の異なるディスクを用いた場合について比較した．その結果をもとに，ディスク利用時の RDD 入出力性能の高速化について検討を行うとともに，RDD キャッシュにディスクを利用する際の指針を明らかにした．
キーワード　ビッグデータ，Spark，RDD キャッシュ，ディスク I/O，性能解析

Toward Improving I/O Performance of Spark RDD

Kaihui ZHANG[†,††], Yusuke TANIMURA[††,†], Hidemoto NAKADA[††,†], and Hirotaka OGAWA[††,†]

† University of Tsukuba　Tennoudai 1–1–1, Tsukuba, Ibaraki, 305–8577 Japan
†† National Institute of Advanced Industrial Science and Technology　Umezono 1–1–1, Tsukuba, Ibaraki, 305–8568 Japan
E-mail: †{neilyo.chou,yusuke.tanimura,hide-nakada,h-ogawa}@aist.go.jp

Abstract　Spark is a parallel data processing framework that can perform iterative calculation, such as machine learning and data mining, in high speed. Spark achieves parallel processing and fault tolerance by using a mechanism called RDD (Resilient Distributed Dataset) and RDD caching allows the Spark programs to reuse intermediate data. When data to be cached is too large to hold in memory of the Spark nodes, some or all of the data can be placed on disks of the nodes. However, use of the disks might degrade execution performance of the Spark applications and it is also a problem that the application users must instruct whether or not the disks are used for caching. In this study, execution performance of the Spark application using disks for caching was investigated by comparing the cases of using different storage levels and disks of different performance. This report summarizes insights form the results for improving I/O performance of the RDD caching using disks and guidelines of when to use disks.
Key words　Big Data, Spark, RDD Caching, Disk I/O, Performance Analysis

1. はじめに

Apache Spark（以下，Spark） [1] は機械学習やデータマイニングを高速に実行できるオープンソースの並列データ処理フレームワークとして，ビッグデータ解析や人工知能分野において高い注目を集めている．Spark は Hadoop [2] が実装する MapReduce [3] と同様の利点を持つ一方，中間データを HDFS（Hadoop Distributed File System）に置くのではなく，ワーカノードのメモリやディスクに保持するなどの工夫により，反復操作や対話処理において Hadoop より優れた性能を提供する．Spark において各データは RDD（Resilient Distributed Dataset）として抽象化され，それぞれの中間データも RDD の 1 つとして扱われる．Spark の主な処理はインメモリで行われるが，シャッフル操作の一部やユーザが指定した場合にワー

カノードのディスクが用いられる．例えば，RDD を再利用や障害対策のためにキャッシュしておく場合に，メモリではなくディスクを選択することが可能である．しかし，ディスクの利用は Spark アプリケーションの実行性能の低下を招く恐れがあり，慎重に行う必要がある．その一方で，ディスク利用の選択はユーザに委ねられており，ディスク利用の是非を適切に判断することが容易ではないという問題がある．

我々はこれまでの研究 [4, 5] において，RDD のキャッシュ（Persist）や Checkpoint におけるストレージへの書き込み性能の調査を行ってきた．本稿ではそれをさらに発展させ，機械学習のプログラムをベンチマークとして用い，RDD キャッシュの効果や利用するディスクの性能と合わせてディスク利用が Spark アプリケーションの実行性能に与える影響について調査を行った結果を報告する．具体的には，RDD キャッシュにメモリまたはディスクを用いた場合の比較，メモリとディスクを併用する場合のキャッシュの挙動調査，4 種類のストレージデバイスを用いた場合の Persist 性能の比較である．これらの結果をもとに RDD キャッシュにディスクを利用する際の指針，およびディスクを利用した Persist の高速化の可能性についてまとめた．

2. Spark の RDD キャッシュの仕組み

2.1 Spark の概要

Spark [1] は University of California, Berkeley の AMPLab（Algorithms, Machines, and People Lab）の研究プロジェクトにより開発が始まったオープンソースの並列データ処理フレームワークである．Hadoop と同様に MapReduce のアルゴリズムに基づいた分散処理が可能であり，コンピューティングとストレージの両機能を持つノードからなるクラスタで実行可能である．Spark は Hadoop MapReduce の利点に加えて，中間データをメモリに保持することで機械学習とデータマイニングなど特定のデータセットに対して複数の反復操作を必要とするアプリケーションを効率良く実行できる特徴を持つ．反復計算の負荷が大きいほど，Spark を利用するメリットはより大きくなる．

Spark は Scala や Java，Python など複数のプログラミング言語をサポートし，Spark-Shell を用いた対話型計算，YARN などの資源管理フレームワークを用いた一括実行が可能である．また，Hadoop エコシステムとの親和性が高く，Hive，HBase，HDFS などと組み合わせて利用することもできる．

2.2 RDD と Persist の仕組み

Spark では RDD（Resilient Distributed Datasets）と呼ばれる，読み取り専用の分散データ構造が用いられる．RDD は内部的にパーティションに分割され，各パーティションをデータ処理の単位として分散並列実行が可能である．各 RDD に対しては map，flatMap，filter，join などのデータ操作を適用できる．ただし，変換操作（map，filter など）はすぐに処理が実行されるわけでなく，アクション操作（count，first など）が呼ばれてからジョブとして投入され，データのロードおよび処理が開始される．これは遅延処理と呼ばれる．

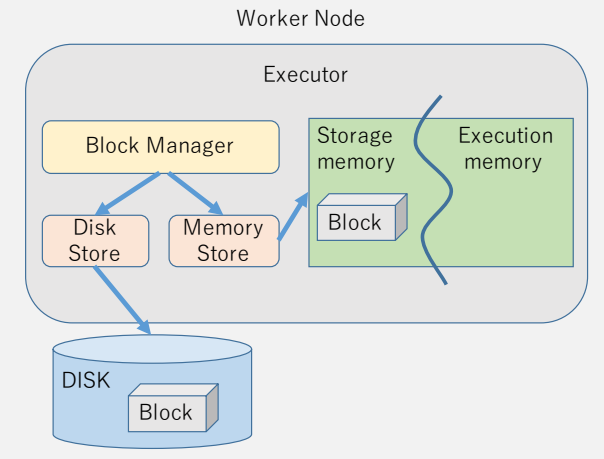

図 1　RDD キャッシュの内部処理

Spark における耐障害性の実現方法は RDD の保存（Persist または Checkpoint）と再構築の 2 つからなる．RDD を生成したノードは，自身が生成したパーティションを明示的にメモリあるいはディスクに保存することができる．Spark ではこれを Persist と呼ぶ．Persist が実行された RDD を保持するノードに障害が発生した場合は，Spark は RDD の生成手順を記録した「Lineage」を参照し，失われたデータのパーティションのみを再度実体化するようにタスクをスケジューリングすることで耐障害性を実現している．またノードに障害があっても全体の処理速度の低下を引き起こさないように，データを複数のノードに複製（レプリケーション）しておくこともできる．

それに加えて，RDD はアクションを実行されるたびに計算し直されるのが基本的な動作であるため，同じ RDD に対して複数の操作を適用する場合に速やかに次の結果を得るためにも Persist が活用される．Persist の実行は，RDD に対して persist() メソッドを呼び出し，引数にストレージレベルを指定することで行う．例えば，ストレージレベルとしてメモリあるいはディスク，またはそれらを併用する指定を選択可能である．メモリを選択する場合にはシリアライズの有無も指定可能である．

2.3 Persist の動作と内部アルゴリズム

RDD はパーティションに分割されているため，一度に RDD の全てがメモリにロードされるわけではない．メモリにロードされたパーティションに対しては Least-Recently-Used（LRU）規則が適用され，メモリに収まらない RDD の Persist が必要になった際，アクセス頻度が最低のパーティションはドロップされる．メモリのみを使用するストレージレベルが設定されている場合，このパーティションは次にアクセスされた時に計算し直されることになる．ストレージレベルがメモリとディスクの併用であれば，当該パーティションはディスクに書き出される．どちらの場合でも，ユーザがメモリ容量を超えて Persist を実行してもジョブの失敗を危惧する必要はないと考えられるが，Executor のメモリやディスクが消費されるので，利用されていない RDD は unpersist() メソッドによりキャッシュ先から削除されることが望ましい．

図 1 に示すように Executor のメモリは，MemoryManager によって内部的に ExecutionMemory と StorageMemory の 2 つに動的に分割されて管理される．ExecutionMemory は RDD を処理するのに用いられる一方，StorageMemory は BlockManager により管理され，RDD パーティションを Block という形で保存するのに用いられる．Persist 実行時のストレージレベルに関して，Spark ではメモリのみを用いる方式がデフォルトであるが，他にもいくつの選択肢がある．以下では方式毎に Persist の挙動を説明する．

MEMORY_ONLY: Persist をメモリのみで行う方式である．Persist が実行されると，BlockManager は MemoryStore のインタフェースを通して StorageMemory に必要分のメモリを確保し，RDD パーティションを Block という形で保存する．性能は最も高速である．

DISK_ONLY: Persist をディスクのみで行う方式である．Persist が実行されると，BlockManager はメモリ上のパーティションをシリアライズし，DiskStore のインタフェースを通してローカルディスクに保存する．性能は最も低速であるが，メモリを消費しない．

MEMORY_AND_DISK: メモリとディスクを併用して Persist を行う方式である．メモリを優先的に利用し，まずは StorageMemory 内に Block として保存することを試みるが，メモリに収まりきらない場合は LRU に基づき，新しい Block が必要とするメモリ量を賄うのに必要なだけの古い Block をシリアライズしてディスクに移動する．ディスクに保存された RDD を参照する際は，BlockManager が利用可能な StorageMemory と StorageMemory 上に該当データを展開するのに必要な容量を確認し，ディスクから Block を読み込み，StorageMemory に戻す操作を行う．

3. 評価実験

3.1 実験方法と実験環境

本研究では先にも述べたように，ディスク利用時の Spark アプリケーションの実行性能を調査し，キャッシュの効果やディスク I/O 性能を考慮して適切にディスク利用を行うための指針を得ることとディスク利用時の高速化を図ることを目的としている．これらを達成するため，以下に述べる評価実験を行った．最初の評価実験では，Spark の機械学習ライブラリ（MLlib）[6] に含まれる SparseNaiveBayes と DenseKMeans のベンチマークを実行し，Persist の有無とストレージレベルの違いによる性能を評価した．さらに，メモリとディスクを併用する場合の効率について評価実験を行った．各入力データは HiBench [7] を用いて生成し，データサイズには Tiny を指定した．また，I/O 性能の異なる複数のストレージデバイスを用いて Persist に要する時間を測定した．この実験では，Spark のワーカ処理において任意のサイズの RDD を生成し，Persist を実行するプログラム（RDDTest）を用いた．

評価実験には表 1 に示すマシンを用い，ローカルモードで Spark の各ベンチマークプログラムを実行した．Persist にディスクを利用する場合には，RAM ディスク，NVMe 接続の SSD，SATA 接続の SSD および HDD の 4 種類を利用した．Spark は v.1.6.1 を用い，Scala v.2.10.6 および Java v.1.8.0_66 を用いてビルドし，実行した．

表 1 評価実験に用いたマシンのスペック

CPU	Intel Xeon CPU E5-2620v3 2.40GHz, 6 cores x2
Memory	128 GB
Network	10 Gbps (for HDFS connection)
NVMe-SSD	Intel SSD DC P3700
SSD	OCZ Vertex3 (240GB, SATA6G I/F)
HDD	Hitachi Travelstar 7K320 (SATA3G I/F)
OS	Ubuntu 14.04 (Kernel v.3.13)

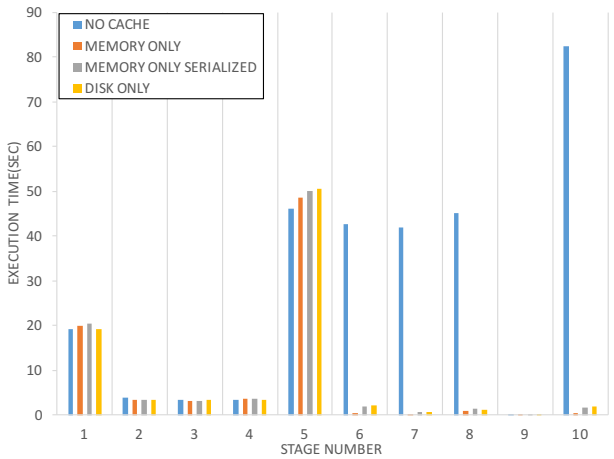

図 2 RDD キャッシュの効果とディスク利用の影響（SparseNaiveBayes）

3.2 ディスク利用による影響

RDD キャッシュの効果とディスク利用による影響を調べるため，スレッド数を 1，Executor に割り当てるメモリを 60GB に設定し，Persist を行わない場合と 3 つのストレージレベル（MEMORY_ONLY，MEMORY_ONLY_SER，DISK_ONLY）を指定して Persist を行う場合の計 4 パターンに対して，MLlib の SparseNaiveBayes と DenseKMeans のベンチマークを実行した．DISK_ONLY の実験では SATA 接続の SSD を用いた．図 2 は SparseNaiveBayes のステージ毎の実行時間をストレージレベルで比較した結果である．Persist を行わない場合，最初は最も高速に実行できるが，ステージ 6 以降は中間データの再生成が必要になるため，実行時間が長くなっているのが分かる．中間データの Persist の重要性を示した結果といえる．MEMORY_ONLY，MEMORY_ONLY_SER と DISK_ONLY の比較では，ステージの移行に対して性能の差が現れたが，全体的な時間差は大きくない．RDD キャッシュにディスクを利用することへの影響は少ないといえる．

同様に，図 3 は DenseKMeans のステージ毎の実行時間を示した結果である．Persist を行わない場合，ステージ 1 では Persist を行う場合よりも短い時間で処理を完了しているが，以降のステージではより長い時間を要しているのが分かる．それとは逆に，Persist を行う場合にはステージ 1 には時間がかかるが，以降のステージでは短い時間で処理が完了している．

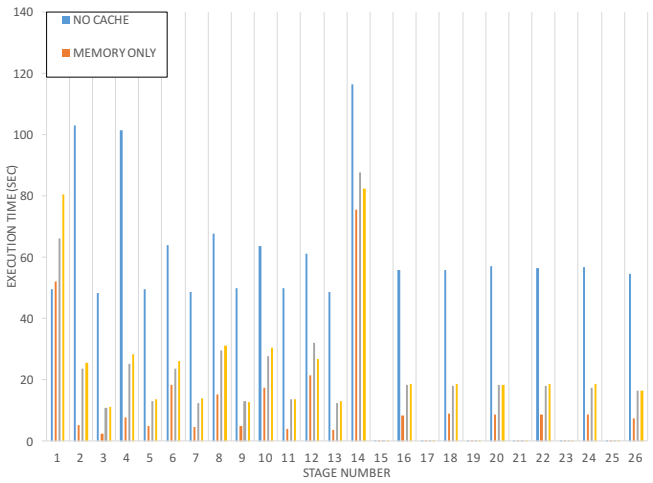

図 3　RDD キャッシュの効果とディスク利用の影響（DenseKMeans）

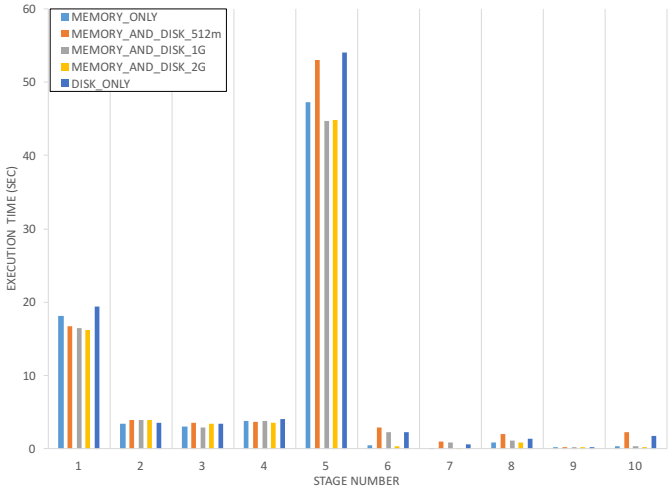

図 4　メモリとディスクの併用による影響（SparseNaiveBays）

MEMORY_ONLY，MEMORY_ONLY_SER と DISK_ONLY の比較では，MEMORY_ONLY が他の 2 つのストレージレベルより良い結果となり，実行時間の差は倍以上となった．MEMORY_ONLY_SER の結果は DISK_ONLY と同等であり，DenseKMeans においては中間データのシリアライズがボトルネックの多くを占めているといえる．

SparseNaiveBayes と DenseKMeans の結果を比べると，後者の方が生成される RDD の合計サイズが 3 倍以上大きく，また各ステージの実行に時間を要しているため，RDD キャッシュの効果がより大きく表れたといえる．

3.3　メモリとディスクの併用による影響

メモリとディスクの併用による影響を調べるため，ストレージレベルを MEMORY_AND_DISK に設定し，Executor に割り当てるメモリ量を 3 つのパターン（512MB，1GB，2GB）に設定して，MEMOLY_ONLY および DISK_ONLY の場合を比較した．MEMORY_ONLY と DISK_ONLY の場合，割り当てるメモリ量は 60BG に設定した．

図 4 は SparseNaiveBayes のステージ毎の実行時間を示した結果であり，図 5 は MEMORY_AND_DISK の各実行において MemoryStore からドロップしたブロック数を示したグラフである．図 5 から分かるように，MEMORY_AND_DISK_2G ではドロップが発生していないため，図 4 の MEMORY_ONLY と同様に，ステージ 6 以降は他の MEMORY_AND_DISK の結果より高速に実行できたと考えられる．MEMORY_AND_DISK_512M と DISK_ONLY はほぼ同等の結果を示しており，MEMORY_AND_DISK におけるドロップの発生による遅延はほとんど見られなかったと考えることができる．

同様に，DenseKMeans の結果を示したのが図 6 および図 7 である．SparseNaiveBays とは異なり，MEMORY_AND_DISK_512M が DISK_ONLY に比べて倍以上の処理時間を要するなど，MEMORY_AND_DISK の実行性能が非常に悪い結果となった．ドロップの状況を見ると，MEMORY_AND_DISK_512M は 1G や 2G に比べて倍以上のブロックがメモリからドロップしており，1 度ドロップされたブロックが再びメモリに格納され，ドロップが繰り返されていることが確認できた．また，メモリが少なくなるほどガベージコレクションの発生率が高くなっており，これらが性能低下の原因であると考えられる．

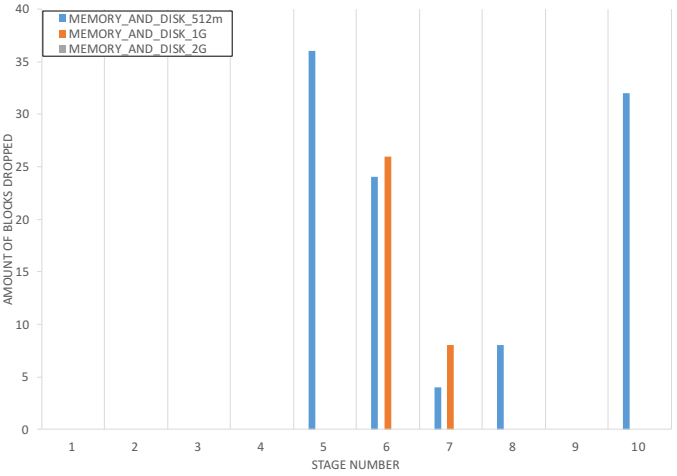

図 5　MemoryStore からのドロップの発生結果（SparseNaiveBays）

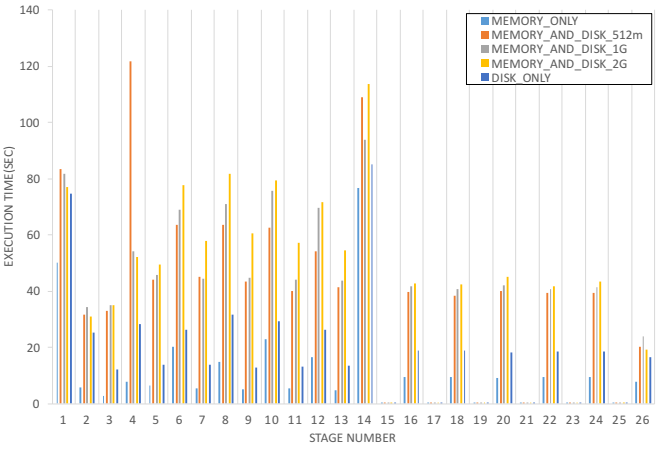

図 6　メモリとディスクの併用による影響（DenseKMeans）

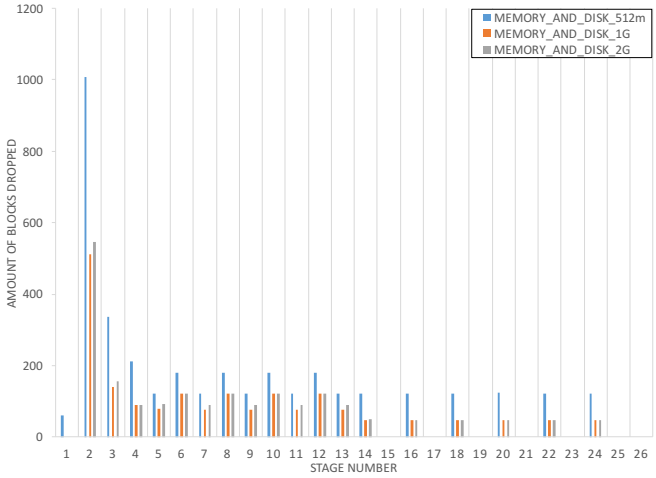

図 7　MemoryStore からのドロップの発生結果（DenseKMeans）

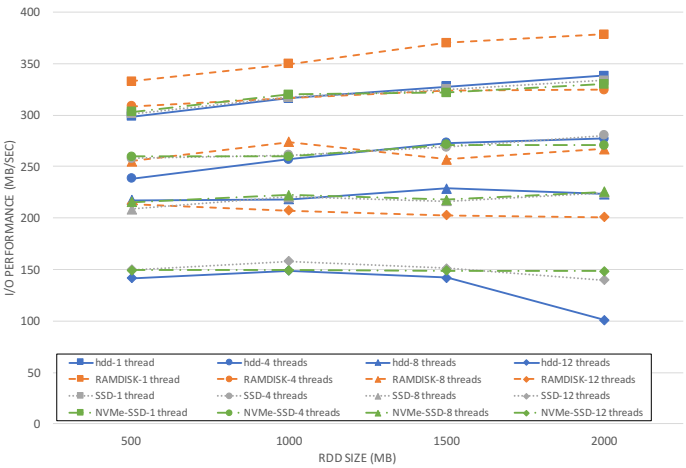

図 9　Persist の実行速度の比較（複数スレッド）

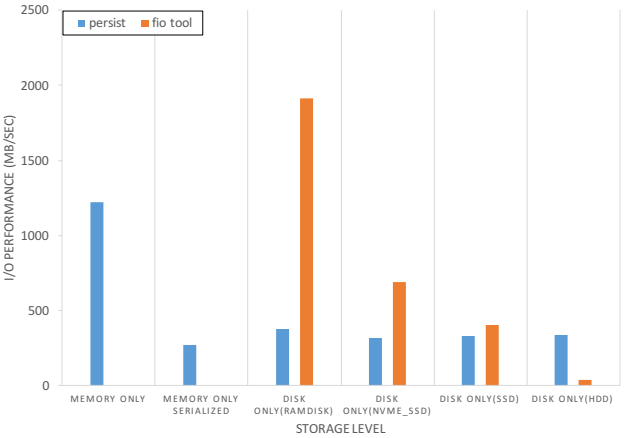

図 8　Persist の実行速度の比較（1 スレッド）

3.4　ストレージデバイスによる違い

ディスクを利用して RDD キャッシュを行う際のストレージデバイスによる性能の違いを調査するため，RDDTest を用いて 1,000MB の RDD を Persist する実験を行った．Persist を実行するスレッド数は 1 である．図 8 は RDDTest の結果と Fio ベンチマーク [8] を用いて各ストレージデバイスの性能を測定した結果を比較したグラフである．この結果より，各ストレージデバイスの性能差が Persist の性能差にあまり影響を与えていないことが分かる．RAM ディスクや NVMe 接続の SSD のようにストレージデバイスが高速である場合にはシリアライズ処理がボトルネックとなり，デバイスの性能を活かすことができていない．また，HDD のようにストレージデバイスが低速であっても，OS のバッファキャッシュの効果により Persist の性能への影響が小さくなっている．

図 9 は同じ RDDTest を用いて，Persist を実行するスレッド数を 1，4，8，12 と変化させた結果である．また，各スレッドが Persist を行う RDD のサイズも 500MB，1,000MB，1,500MB，2,000MB と変化させた．図 9 より RDD サイズおよびスレッド数を増やすことにより，OS のバッファキャッシュの効果が小さくなっていることが確認できる．RDD サイズの増加に対して，RAM ディスクと NVMe 接続の SSD の結果は Persist の I/O 性能を維持しているが，HDD ではスレッド数が 8，RDD サイズが 1,000MB より Persist の I/O 性能の低下が見られる．これらの結果より，複数のタスクを同時に処理して Persist が実行される時に初めてディスク性能が重要になってくることが分かる．

4.　議　　論

本章では 3. 章の結果をもとに，RDD キャッシュ（Persist）におけるディスク利用の指針と Persist の高速化について検討した結果を述べる．

RDD キャッシュの効果は Persist の実行時間と RDD の再生成がトレードオフの関係にあるが，今回試したベンチマークにおいては再生成のコストが圧倒的に高く，キャッシュが非常に有効であった．一方，メモリにキャッシュする方式とディスクにキャッシュする方式では，アプリケーションの全体の実行時間に比べて Persist の時間差はわずかであり，ディスク利用は有効な選択肢の 1 つである．特に，メモリとディスクを併用する方式において，メモリが不足する場合にディスクへのドロップが多発したり，ガベージコレクションの影響を受けたりすることにより，かえってシステムへの負荷が高まり，アプリケーションの実行性能が落ちるケースが見られた．つまり，メモリだけではキャッシュ容量が不足するアプリケーションにおいては最初からディスクのみを利用した方が良い性能を得られる可能性が高い．

RDD キャッシュにおいては，OS のバッファキャッシュがあるためにディスクの性能はボトルネックになりにくい．1 つのワーカノードにおいて複数のタスクを並列に実行し，各タスクが同時に Persist を実行する場合やメモリの空きが少ない場合においてのみディスク性能の影響がアプリケーションの実行性能に現れると考えられる．それ以外のケースでは，高速なディスクよりも容量の大きい HDD などを用いた方がコストパフォーマンスに優れている．

Persist の実行を高速化するには，高速なディスクを用いて

もあまり効果がないため，Spark 内部の仕組みの改善が必要であると考える．特にメモリとディスクを併用する方式には今回の実験で明らかになった問題があり，同じブロックが何度もドロップされないようにする工夫や新しいブロックを Persist する直前にドロップを実行するのではなく，他の RDD が処理されている間に非同期にドロップを行う案が考えられる．

5. 関 連 研 究

Spark においてメモリと CPU 利用の効率化を図る試みの 1 つに Project Tungsten [9] がある．JVM のオブジェクトモデルやガベージコレクションのオーバヘッドの削減，オフヒープの実装，シリアライズの高速化等により RDD API や DataFrame を用いた Spark プログラム，および Spark SQL の実行速度の改善を実現している．また，RDD キャッシュ用のメモリが不足する際にドロップする RDD を選択する方法については，RDD パーティションの生成コストを考慮した Weight Replacement アルゴリズムの研究が行われている [10]．これらの研究では RDD をメモリにのみキャッシュする方式に焦点が当てられているのに対し，本研究ではディスクを利用した RDD キャッシュの性能解析や高速化を目的としている点に違いがある．

J. Shi らは Spark のタスク実行をプロファイリングして性能解析を行っており，RDD のキャッシュ効果やストレージレベルの影響，ボトルネックの所在について調査を行っている [11]．K-means 実行時の RDD キャッシュの効果に関する調査など本研究と類似する取り組みが見られるが，Hadoop MapReduce と Spark の性能差の理由や耐障害性をアーキテクチャや実装の観点から明らかにすることに主眼が置かれている．

6. おわりに

本稿では Spark においてディスクを利用する RDD キャッシュについて調査を行い，ディスク利用が与える影響やディスクの性能による違いを明らかにし，ユーザがディスク利用を選択する指針やキャッシュの高速化について検討した結果を述べた．今後の課題としては，MLlib のベンチマークだけでなく他のアプリケーションにも適用して，今回得られた指針を一般化することやより大きなデータセット，クラスタ環境を用いての評価実験，メモリとディスクの併用方式における Persist の性能改善などが挙げられる．

謝　　辞

本成果の一部は，国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものである．また，本研究の一部は JSPS 科研費 JP16K00116 の助成を受けたものである．

文　　献

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, and I. Stoica, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI’12), pp.15–28, 2012.

[2] “Apache Hadoop”. http://hadoop.apache.org/

[3] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI’04), pp.1–13, 2004.

[4] 谷村勇輔，小川宏高，“Spark RDD のストレージ出力に関する性能評価，” 情報処理学会研究報告，第 2016-HPC-153 巻，pp.1–5，2016.

[5] 谷村勇輔，小川宏高，“Spark における中間データ用ローカルストレージの構成方式の検討，” 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，p.54，2016.

[6] X. Meng and etal., “MLlib: Machine Learning in Apache Spark,” Machine Learning Research, vol.17, no.1, pp.1235–1241, 2016.

[7] “HiBench Suite”. https://github.com/intel-hadoop/HiBench

[8] “Fio (Flexible I/O Tester)”. https://github.com/axboe/fio

[9] “Project Tungsten (SPARK-7075)”. https://issues.apache.org/jira/browse/SPARK-7075

[10] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, “Selection and Replacement Algorithms for Memory Performance Improvement in Spark,” Concurrency and Computation: Practice and Experience, vol.28, no.8, pp.2473–2486, 2013.

[11] J. Shi, Y. Qiu, U.F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, “Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics,” Proceedings of the VLDB Endowment, vol.8, pp.2110–2121, 2015.