Vol.2017-OS-140 No.13
2017/5/17

Cgroupsを利用したHadoopにおける落ちこぼれタスクのリソース制限による再現

岩井 良成[1,a)]　杉木 章義[2]　棟朝 雅晴[2]

概要：本研究では，Hadoop において著しく実行時間が長いタスク，落ちこぼれタスク（Straggling Tasks）を再現するフレームワークを提案する．本研究では，落ちこぼれタスクのうち，リソース制限により再現可能であるものを対象とする．リソース制限には，Docker コンテナと cgroups を活用し，犠牲となる Map/Reduce タスクを確率的に選択し，これらのリソース使用量を一定以下に制限することで実現した．実験では，HiBench ベンチマークを活用し，再現する落ちこぼれタスクの割合と，リソース制限量を変化させることで，Hadoop ジョブ全体の実行が落ちこぼれタスクにより遅延することを確認した．

キーワード：並列分散処理，OS コンテナ，リソース制限，Hadoop，MapReduce，cgroups

1. はじめに

大規模データの並列分散処理システムとして，Hadoop[1] が広く活用されている．Hadoop は Google の MapReduce システム [2] を参考に開発されており，近年の初期に登場した大規模データ処理システムであるが，現在でも ETL（Extract-Transform-Load）処理などに広く活用されている．

Hadoop に限らず，一般に並列分散処理システムでは，同一フェーズの処理において，他のタスクに比べて著しく実行時間の長いタスク（Straggling Tasks）の存在が知られている．本研究では，これらのタスクを落ちこぼれタスクと呼ぶこととする．落ちこぼれタスクの存在は，同一フェーズの処理，ひいてはジョブ全体の実行時間を大きく遅延させることが知られている．Hadoop においては，Map，Shuffle，Reduce の 3 つのフェーズのタスク間で処理が依存しており，前フェーズでの落ちこぼれタスクの存在が後続のタスク処理に大きく影響を与える．

落ちこぼれタスクが全体の実行時間を大きく遅延させる理由は，タスク間に依存関係が存在するからである．Hadoop では，Map，Shuffle，Reduce の 3 つのフェーズ間で依存する比較的シンプルな構造である．一方で，落ちこぼれタスクには，ロングテール性があることも知られている．落ちこぼれタスクの発生確率は低いものの，その実行時間は他のタスクと比較して極端に大きいものとなる．

Hadoop においても，落ちこぼれタスクに対処したとするさまざまな研究が提案されている [3], [4], [5]．しかしながら，落ちこぼれタスクの発生は動的な環境に大きく依存し，その再現や，正しく対処されているかどうかの確認を行うことが難しい．また，多くの研究で，アプリケーションや Hadoop を改造することで，落ちこぼれタスクをアドホックに再現する試みが行われており，共通性かつ再利用性のある手法とは言い難い．

本研究では，Hadoop を対象とした落ちこぼれタスクの統一的な再現フレームワークを提案する．本研究では，Docker Container Executor（DCE）[6] を活用し，Map および Reduce の各タスクを Docker コンテナとして実行することで，個別のタスク単位でのリソース制限を可能とする．また，Docker が基礎としている cgroups を活用することで，きめ細やかな細粒度でのリソース制限を可能とする．本手法は，既存の DCE の枠組みを活用しているため，アプリケーション及び Hadoop の双方を改変する必要はない．また，将来的には，同一フレームワーク上でさまざまな研究を比較することも可能とする．

実験では，HiBench ベンチマーク [7], [8] を活用し，本フレームワークにより落ちこぼれタスクが再現できるか確認した．その結果，複数のワークロードで CPU およびメモリのリソース制限により，落ちこぼれタスクが発生し，ベンチマークのジョブ全体の実行時間が大きくなること

[1] 北海道大学大学院 情報科学研究科 情報理工学専攻
Graduate School of Information Science and Technology, Hokkaido University
[2] 北海道大学 情報基盤センター
Information Initiative Center, Hokkaido University
[a)] E-mail:iwakmn@eis.hokudai.ac.jp

を確認した．一方で，さまざまな将来的な課題も明らかになった．

2. 落ちこぼれタスクの分類

落ちこぼれタスクとは，同一の処理フェーズにおいて，他のタスクと比べて著しく処理時間の長いタスクである．Hadoop においては，Map，Shuffle，Reduce の各フェーズにおいて，他のタスクと比較して処理時間が著しく長いタスクである．落ちこぼれタスクが発生する原因にはさまざまなものがあり，また発生するかどうかは，実行してみるまでわからないことも多く，その再現や対処も難しいものとなる．

本研究では，落ちこぼれタスクの発生要因をシナリオとして分類し，下記にいくつかのシナリオを示す．

- **処理性能の差によるもの：** Hadoop のクラスタでは，Map，Shuffle，Reduce の各フェーズにて，ほぼ均一の速度で処理が進行することを期待する．実際，多くのスケジューラで，均一（homogeneous）なハードウェアであることを仮定している．しかしながら，近年，メニィコア CPU，GPGPU，FPGA，ASIC，DSP などのさまざまなデバイスが活用され，ハードウェアも段階的に導入されるなど，不均一（heterogeneous）なハードウェアが活用される傾向にある．また，同一性能のハードウェアであっても，近年の CPU の省電力制御などにより，処理速度が異なることがある．
- **リソース競合によるもの：**Hadoop のタスク間のリソース競合により，落ちこぼれタスクとなることがある．リソース競合は 2 つの理由により，起こりうる．一つ目は，同一ノード上での複数タスクの実行により，CPU やメモリなどの共用リソースを奪い合い，リソース競合が発生することがある．また，複数ノード間でストレージを共有している場合など，共用ストレージやネットワーク上での競合によっても起こりうる．二つ目は，バックグラウンドで実行されている仮想マシンによる影響である．近年，Amazon EC2，Microsoft Azure，Google Cloud Platform などのパブリッククラウドにおいても大規模データ処理が広く行われており，バックグラウンドで実行されている仮想マシンによるリソース競合が大きな問題となる．
- **データの偏りによるもの：**Map フェーズは，入力データをほぼ等サイズに分割するため，データの偏りは生じにくいが，Reduce フェーズの入力となる Key と対応する複数の Value によるタプルの量は key の偏りに依存するため，特定のタスクの処理量が膨大となり，落ちこぼれタスクが発生しうる．また，Hadoop では HDFS（Hadoop Distributed File System）が使われることが多いが，前述の Map フェーズの場合においても，データがロードされているノードとタスク処理を行うノードが異なることがあり，落ちこぼれタスクが発生しうる．

前半の二つはリソース制限の工夫により，再現可能であり，本研究の対象とする．しかしながら，最後のデータの偏りによるものはリソース制限により再現できず，本研究の対象外とする．

3. システムの概要

本研究では，Hadoop における落ちこぼれタスクの再現フレームワークを提案する．Docker Container Executor（DCE）[6] を活用し，Map/Reduce の各タスクを Docker コンテナとして実行することでリソース制限を可能とする．cgroups を活用したリソース制限の工夫により，落ちこぼれタスクを再現する．

3.1 Docker Container Executor

DCE は，YARN[9] の NodeManager が YARN コンテナを Docker コンテナとして実行する機能を提供する．YARN のコンテナ間の独立性を高め，ライブラリなどの依存関係を解決することを目的としている．MapReduce と組み合わせて使用した場合には，Map タスクおよび Reduce タスクが Docker コンテナとして実行され，Shuffle タスクは Docker コンテナとして実行されない．

本研究では，落ちこぼれタスクの再現を目的として，DCE を Map/Reduce タスク単位での細粒度なリソース制限のために活用する．

3.2 Cgroups

近年の Linux が提供する cgroups は，細粒度なリソース制限のための手段を提供し，Docker 自身も活用している．cgroups では，アプリケーションに対応する複数のプロセスを一つのグループとしてまとめ，リソースの課金や制限を行う機能を提供する．

具体的なリソースの制限は下記のように行う．

- **CPU のリソース制限：**cgroups では，CPU のコア数および CPU 実行時間を制限するパラメータを提供している．しかしながら，CPU コア数については，YARN の仮想コア数との複雑な関係があるため，CPU 実行時間の制限のみを使用する．CPU 実行時間の制限は，CPU クロックを変化させた場合と同じような効果を得ることができる．CPU の実行時間は，`cpu.cfs_period_us` と `cpu.cfs_quota_us` の 2 つのパラメータで調整可能であり，あるタスクの CPU 使用率を p に制限する場合，以下で調整可能である．

$$cpu.cfs\_period\_us = p * cpu.cfs\_quota\_us$$

- **メモリのリソース制限：**cgroups では，メモリ使用量の上限を `memory.limit_in_bytes` で制限可能である．

タスクのメモリ使用量を 200MB に制限したい場合，本パラメータを 200MB に設定する．しかしながら，実際には，YARN が各タスクに割り当てるメモリ量，JVM のヒープサイズなど，入れ子の複雑な関係がある．また，cgroups では，メモリの最大スワップ量，OOM（Out-of-Memory）Killer によりメモリ制限を超過した場合の挙動も変更可能である．

3.3 リソース制限の実行

Hadoop の Map/Reduce のタスク数は入力データなど，動的な環境に依存し，実行してみるまで分からない．冗長タスク（Redundant Tasks）の投機的実行やタスクの強制終了などの挙動にも依存し，最終的なタスク数が確定するのは，ジョブ全体の終了時である．

また，Docker コンテナや cgroups の階層では，そもそも Map タスクなのか Reduce タスクなのか判別することも難しい．これらの判別を行うためには，Hadoop 本体との緊密な連携が必要である．

そのため，本研究では実装を簡単にすることを目的に，犠牲とする（Victim）タスクを確率的に決定する．新しいタスクを発見するたびに，乱数を生成し，その乱数が予め定められた一定値以下であれば，犠牲タスクとして決定する．この手法は大域的な情報を必要とせず，簡単に実装できるが，一方で，同一環境の繰り返し実験であっても犠牲となるタスク数が毎回異なるという欠点がある．ただし，期待値としては，与えられた割合で一定となる．より詳細な情報を活用した決定的な手法については，今後の課題とする．

今回，犠牲タスクのリソース制限量を Map/Reduce タスクの実行時間全体にわたって一定にする．タスクの実行中にリソース制限量を上下させ，より落ちこぼれタスクらしい振る舞いをさせることも可能であるが，これらは将来的な課題とする．今回，まずはリソース制限により，落ちこぼれタスクが再現できることを確認することを目的とする．

4. 実装

実装は，下記の Task Watcher と Resource Controller の二つのコンポーネントで実現されている．

- **Task Watcher：** cgroups のサブディレクトリ以下を監視し，Docker コンテナの起動，つまりは Map/Reduce タスクの起動を検知して，下記の Resource Controller に伝える．本コンポーネントは，Map/Reduce タスクが実行される可能性がある全てのノードで起動しておく必要があり，Hadoop のマスタノードで実行される Task Watcher は他のノードから情報を収集する役割も担う．
- **Resource Controller：** Task Watcher から伝えられた情報をもとに，犠牲タスクとするかどうか確率的に判断し，必要な場合に実際にリソース制限を行う．犠牲タスクの割合，リソース制限量ともに調整可能となっており，それぞれ 0%から 100%の割合で調整可能である．

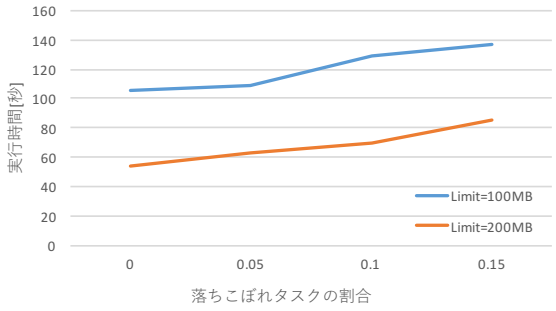

図 1 メモリ制限とジョブ実行時間（wordcount）

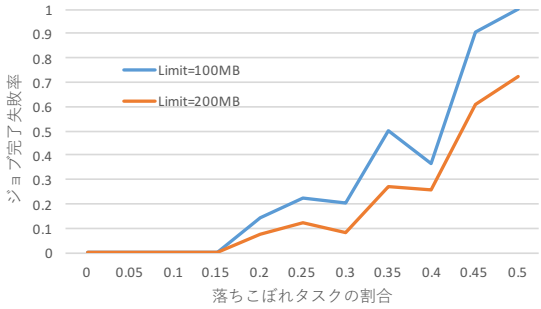

図 2 メモリ制限とジョブ完了失敗率（wordcount）

5. 実験

実験では，落ちこぼれタスクが再現できることを確認する．また，落ちこぼれタスクとなる犠牲タスクの割合やリソース制限の度合いを変化させることで，ジョブ全体の実行時間がどの程度，変化するか確認する．なお，各実行時間については，10 回の平均値を用いている．

5.1 実験環境

Hadoop クラスタとして，OpenStack 上で構築された 7 台の KVM 仮想マシンを使用した．OpenStack は各サーバあたり Intel Xeon-E5530 の 4 コア CPU，8GB メモリを搭載したハードウェアで構築されている．

Hadoop は 2.7.1 を使用し，Docker は 1.10.3 を使用した．ベンチマークとして HiBench 5.0[7], [8] を用い，データスケールを Large に設定した．今回の実験では，HiBench が提供するワークロードのうち，`wordcount`，`sort`，`terasort`，`join`，`aggregation`，`kmeans` を使用した．

5.2 実験結果：メモリのリソース制限

図 1 にメモリのリソース制限を行なった場合の結果を示

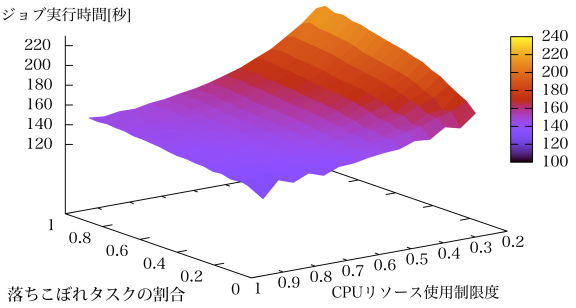

図 3 CPU 制限と実行時間（wordcount）

す．横軸は再現する落ちこぼれタスクの割合であり，縦軸は wordcount ジョブ全体の実行時間である．落ちこぼれタスクの割合が増加し，またメモリ使用量の制限を厳しくするのに従って，実行時間が増加していることが分かる．

しかしながら一方で，**図 2** を見ると，メモリのリソース制限はジョブ全体の実行失敗を引き起こすことが分かる．再現する落ちこぼれタスクの割合が 15%を超えたあたりから，ジョブ全体が失敗し始め，落ちこぼれタスクの割合が 50%および 100MB の制限の場合で，wordcount のジョブ完了失敗率が 100%となる．図 2 については，再現する落ちこぼれタスクの割合が 50%，図 1 については，20%で実験を打ち切っている．

これは，メモリのリソース制限は YARN がタスクに割り当てるメモリ，JVM ヒープサイズに直接関係することから，メモリの制限がタスクの実行失敗に繋がるからである．結果として，Map/Reduce の各タスクの失敗が投機的な冗長タスクの実行で隠蔽できなくなると，ジョブ全体の失敗に繋がり，図 2 のように表面化する．

以上から，メモリのリソース制限は cgroups の活用により可能であるものの，各 Map/Reduce タスクの実行を失敗させ，ジョブ全体の失敗にも繋がるため，落ちこぼれタスクの再現にはあまり適さないことが分かった．

5.3 実験結果：CPU のリソース制限

図 3 に CPU リソースの制限を行なった場合の結果を示す．本実験では，再現する落ちこぼれタスクの割合，落ちこぼれタスクあたりの CPU リソース制限の量を変化させ，wordcount ジョブ全体の実行時間がどのように変化するかを観測している．CPU リソースの制限については，最大クロックを 100%とし，そこから制限を加えた場合の CPU クロック相当が当初の何%であるかを示している．

図 3 から，再現する落ちこぼれタスクの割合が大きくなるに従って，CPU リソースの制限量が大きくなるに従って，wordcount ジョブの実行時間が大きく増加している．本実験では，メモリ制限の場合の実験とは異なり，wordcount ジョブ全体の実行失敗は発生しなかった．

図 4 に HiBench のワークロードごとの CPU のリソース制限による影響を示す．図 4 では，CPU リソースの制限量を一定とし，再現する落ちこぼれタスクの割合を変化させている．当初，ワークロードごとに傾向が異なることを期待したが，大きな違いは得られなかった．

図 4 ワークロードごとの落ちこぼれタスクの割合による影響（CPU 制限の場合）

6. 議論と今後の予定

6.1 CPU，メモリ以外のリソース制限

Hadoop は大規模データを対象とした並列分散処理システムであり，CPU やメモリなどのリソースと比較して，ディスクやネットワークなどの I/O リソースも重要ではないかと考えられる．しかしながら，これらに対するリソース制限は限られた時間内に実現できなかった．

cgroups 自身はディスクやネットワーク I/O に関する制限機能を有しているものの，DCE で生成されたコンテナに制限を実施しても，制限が反映されなかった．

ディスク I/O に関して原因を調査したところ，ゲストとなる OS コンテナ内部からホスト OS のファイルシステムがマウントされており，ホスト OS 上のファイルにアクセスする場合には，処理がバイパスされ，リソース制限が行われないことが分かった．これをゲスト OS 内のファイルにアクセスするよう修正すれば，制限が実施されるようになるが，マウントされたファイルを通じてデータのやり取りや設定ファイルの共有を行うなど依存関係があり，修正は容易でないことが分かった．

ネットワーク I/O については，調査が未完となっているが，ネットワーク I/O は Hadoop のライブラリ経由で行われている可能性があり，ゲストとなるコンテナに I/O が課金されず，ホスト OS 側に課金されている可能性がある．

以上により，DCE よりも新しい仕組みである CGroups with YARN の活用を検討している．本機構は Docker コ

ンテナを使用せず，cgroups を直接使用したリソース制限を可能とする．従って，原始的かつ軽量な仕組みであるため，DCE よりも修正が容易であることが期待される．しかしながら，Hadoop の cgroups マウントに関するバグ [YARN-2194] により，検証が行き詰まっている．

また，Spark[10], [11] も YARN の枠組みを使用しており，今後は Spark に関しても適用できることを期待する．Spark はインメモリのデータを対象とした CPU 計算が中心となっており，もし万が一，ディスクやネットワーク I/O に関して制限できなくても，一定の効果が得られる可能性がある．

6.2 落ちこぼれタスクの振る舞いの再現

今回の実験では，落ちこぼれタスクの割合が増加するに従って，リソース制限の度合いを大きくするにつれて，ジョブ全体の実行時間が増加するという，ある意味，当然の結果が得られた．落ちこぼれタスクが現実的に問題となるのは，非常に少数にも関わらず，実行時間に大きな影響を与えるなどのケースである．そのため，より現実の落ちこぼれタスクらしい振る舞いを再現することを検討する．

また，今回は犠牲となる落ちこぼれタスクを確率的に選択するようにしたが，これは落ちこぼれタスクがどこで発生するか分からないという現実に即しているものの，同じ環境でも落ちこぼれタスクの数が固定されないという問題があった．これは，繰り返し実験を行い，結果を比較する際に度々問題となった．Map/Reduce のフェーズも意識した，決定的（deterministic）なタスクの選択方法について，今後検討する．

6.3 スケジューラの性能評価

本フレームワークを活用して，すでに既存のスケジューラ研究の評価や比較は可能である．しかしながら，一般に多くの研究で論文は公開されているもの実装は公開されていない．実際に評価を行うためには，論文をもとにスケジューラの再実装が必要であり，その労力が障害となる．また，スケジューラに関して，Yarn Scheduler Load Simulator（SLS）[12] などのシミュレータによる評価が提案されているが，本研究では，実際の環境に近いエミュレーションによる評価が可能である．

7. 関連研究

落ちこぼれタスクの存在は，Google によるオリジナルの MapReduce 論文 [2] から指摘されており，さまざまな対処法が提案されている．本研究は落ちこぼれタスクについて直接対処するものではないが，落ちこぼれタスクの問題解決に向けた統一的な再現フレームワークを提供する．

Google による MapReduce 論文では，バックアップタスク（Backup Tasks）を導入し，投機的な冗長タスクの実行や，早期のタスク処理の打ち切りにより，落ちこぼれタスクの影響を軽減する試みが行われている．同様の仕組みは Hadoop にも導入されている．

Chen ら [4] による研究では，落ちこぼれタスクによるジョブの実行時間およびクラスタのスループット性能への影響を指摘し，Maximum Cost Performance（MCP）を指標とした投機的実行を提案している．本手法は，より正確なタスク処理速度の予測とコスト利得モデルに基づいたバックアップタスクの選択をもととしている．

Zhang ら [3] による PRISM では，Map および Reduce のタスク内部でさらに詳細なフェーズと呼ぶ段階に分割できることを指摘し，フェーズに分割した細粒度のスケジューラを提案している．

Yu ら [5] は，Map/Redude のタスク処理を行うノードと HDFS のデータが配置されているノードが異なることによる落ちこぼれタスク発生の問題を指摘し，投機的なプリフェッチとネットワーク帯域確保による手法を提案している．

本研究の手法は，2 章で指摘したように，処理速度の差やリソース競合に関する落ちこぼれタスクについては，再現可能であると思われる．一方で，データの偏りによる落ちこぼれタスクの再現については難しい．従って，Chen らと Zhang らの提案については，本フレームワークまたはそれの改良により検証可能であると推測されるが，Yu らの手法の検証については難しい．

さらに，本研究は，性能低下を人工的に注入する点において，Fault Injection 技術とも関連する．元々の Fault Injection 技術は，障害を人工的に注入し，検証することで，ソフトウェアの品質を高めることを目的としたものであるが，本研究では，Hadoop 上で落ちこぼれタスクを再現し，影響を軽減することを目的とする．

最後に，Hadoop に限らず，インメモリ計算型の Spark，大規模機械学習の並列分散処理 [13] でも落ちこぼれタスクによる影響が指摘されており，これらの方面へも研究の展開を検討する．

8. まとめ

本研究では，Hadoop を対象とし，落ちこぼれタスクを再現するフレームワークを提案した．Hadoop などの大規模並列分散処理では，落ちこぼれタスクという他のタスクに比べ，極端に実行時間の長いタスクの問題が一般的に知られており，さまざまな解決策が提案されている．本研究では，それらの解決策を統一的に評価するための落ちこぼれタスクの再現フレームワークを作成した．本研究は落ちこぼれタスクを確率的に再現し，その落ちこぼれタスクの遅延具合についても，リソース制限の強さによって再現可能である．本フレームワークは Hadoop やアプリケーションに手を加える必要がなく，実在するいくつかの落ちこぼ

れタスクの発生のシナリオに対応している。

参考文献

[1] The Apache Software Foundation: Apache Hadoop.
[2] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *USENIX OSDI'04*, Berkeley, CA, USA, pp. 10–10 (2004).
[3] Zhang, Q., Zhani, M. F., Yang, Y., Boutaba, R. and Wong, B.: PRISM: Fine-grained Resource-aware Scheduling for MapReduce, *IEEE Trans. on Cloud Computing*, Vol. 3, No. 2, pp. 182–194 (2015).
[4] Chen, Q., Liu, C. and Xiao, Z.: Improving MapReduce Performance using Smart Speculative Execution Strategy, *IEEE Trans. on Computers*, Vol. 63, No. 4, pp. 954–967 (2014).
[5] Yu, Z., Li, M., Yang, X., Zhao, H. and Li, X.: Taming Non-local Stragglers Using Efficient Prefetching in MapReduce, *IEEE Cluster'15*, pp. 52–61 (2015).
[6] The Apache Software Foundation: Docker Container Executor.
[7] Intel: HiBench Suite.
[8] S. Huang et al.: The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis, *IEEE ICDEW 2010*, pp. 41–51 (2010).
[9] V. Kumar et al.: Apache Hadoop YARN: Yet Another Resource Negotiator, *ACM SOCC'13*, New York, NY, USA, pp. 5:1–5:16 (2013).
[10] The Apache Software Foundation: Apache Spark: Lightling-fast Cluster Computing.
[11] M. Zaharia et al.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *USENIX NSDI'12* (2012).
[12] The Apache Software Foundation: Yarn Scheduler Load Simulator (SLS).
[13] A. Harlap et al.: Addressing the Straggler Problem for Iterative Convergent Parallel ML, *ACM SOCC'16*, pp. 98–111 (2016).