Vol.2017-OS-141 No.17
2017/7/27

Hadoop/Sparkにおける第2階層スケジューラ導入による タスク進捗度平準化の検討

杉木 章義[1,a)]　岩井 良成[2]

概要：本研究では，Hadoop および Spark を対象とし，従来からのクラスタ・レベルのスケジューラに加えて，各ノード単位で複数タスクの調停を行うノード・レベルスケジューラを導入し，各 Map/Reduce タスク等の進捗度の平準化を行う．同一処理フェーズのタスク進捗の平準化を行うことで，ジョブ全体の処理が高速化されることを期待する．実装では，YARN の CGroups サポートを活用し，CGroups に基づくリソース監視や制限を行う．本実装は YARN を対象としているため，Hadoop（MapReduce）及び Spark の双方に適用できることが期待される．

キーワード：並列分散，リソース制御，負荷分散，OS コンテナ，Hadoop，CGroups

1. はじめに

近年，ビッグデータ処理基盤として Hadoop [1] および Spark [2] が活用されている．Hadoop は Google 社の MapReduce [3] を参考に開発されたシステムであり，メモリ上に収まりきらないディスク上の大規模データ処理に広く活用されている．一方の Spark はインメモリ型の並列処理基盤であり，分散メモリに収まる程度の大きさのデータの繰り返し処理や対話処理を中心に活用されている．両者の基盤は，最近では機械学習にも用いられており，学習の前処理や学習そのもの [4] にも活用されている．

Hadoop や Spark に代表される並列分散処理基盤では，一般に**タスク**間の処理速度のばらつきによる性能低下が問題となる．これは，HPC（High Performance Computing）における古典的な問題であり，並列分散処理ではリソースを最大限活用し，均一に処理が進んだ場合に最大の性能が得られることを期待する．しかしながら，実際には，多くの場合，処理が早く完了したタスクが処理の遅いタスクの完了を待つことになる．これの極端な場合が **straggling tasks** であり，他のタスクに比べて極端に遅いタスクの存在が知られている．

タスクの処理速度のばらつきは，さまざまな理由で発生し，完全に対処することは難しい．一般に処理速度のばらつきは，不均一（heterogeneous）なシステム構成，リソース競合，入力となるデータの偏り（skew）などによって発生する．さらには近年，Amazon Web Services，Microsoft Azure，Google Cloud Platform などのパブリッククラウドにおいて，バックグラウンドで動作する他の仮想マシンによる影響が問題として知られており [5]，対策が求められている．これらのパブリッククラウドで Hadoop や Spark を動作させせることは，非常によく行われている．

処理速度のばらつきへの対策として，ばらつきを考慮したタスクの割り当て（allocation）や動的な負荷分散（load balancing）が行われる．特にリソース競合については，さまざまな要因で発生する可能性があり，事前に予見して完全に対処することが難しく，動的なリソース監視や調整による対策が必要である．

本論文では，Hadoop および Spark への第 2 階層スケジューラ導入によるタスク**進捗度**（progress）の**平準化**（leveling）について検討する．第 2 階層であるとは，標準的なクラスタ・レベルのスケジューラ（cluster-level scheduler）とは別に，各ノード単位で複数タスクの調停を行うノード・レベルのスケジューラ（node-level scheduler）を新たに導入する．本スケジューラは標準のクラスタ・レベルスケジューラと独立して，干渉することなく動作し，Mesos [6] や Omega [7] の Two-level Scheduling と異なる意味で第 2 階層スケジューラを用いる．本方式は，各タスクの性能指標となるタスクの進捗度を観測し，同一処理グループの他のタスクに比べて処理が遅れているタスク

1 北海道大学 情報基盤センター
Information Initiative Center, Hokkaido University
2 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
a) E-mail:sugiki@iic.hokudai.ac.jp

ⓒ 2017 Information Processing Society of Japan

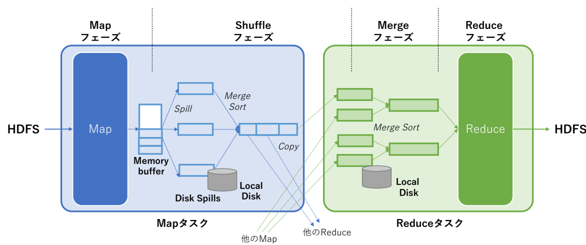

図 1　MapReduce 処理の分解

に，より多くのリソースを割り当てる．以上により，全体の進捗度を揃えることで，クラスタ全体にわたるジョブの完了時間が短縮されることを期待する．各タスクが使用するリソースの監視や制御には，CGroups を活用する．本論文では，Hadoop の MapReduce モデルを対象に研究を進め，将来的に Spark も研究を対象とする．本論文の方式は，YARN を対象に実装されているため，容易に Spark にも対応可能であることが期待される．

2. 前提

2.1 MapReduce 処理の分解

本論文の手法は，Hadoop および Spark 双方への適用を目指しているが，まずは Hadoop の MapReduce モデルを研究の対象とする．これは，MapReduce は処理が単純で，モデル化や動作の把握が容易であるからである．

MapReduce では，下記の 2 つの関数を利用者に記述させ，分散された多数のデータチャンクに対して，並列にこれらの関数を適用する並列分散処理基盤となっている．

$$\begin{aligned} &\mathrm{map} \quad (k_1, v_1) &&\rightarrow \mathit{list}(k_2, v_2) \\ &\mathrm{reduce}\ (k_2, \mathit{list}(v_2)) &&\rightarrow \mathit{list}(v_2) \end{aligned}$$

MapReduce の処理過程では，上記の 2 つの関数に対応する **Map タスク**，**Reduce タスク**の 2 つのステップが存在する．しかしながら，実際には，各タスクの処理過程において，より詳細な複数の**フェーズ**（phase）に分解することができる（**図 1**）．フェーズへの分割方法は研究により異なるが，本研究では，文献 [8] を参考に分割した．特に，Map と Reduce の中間のフェーズでは，かなり複雑な処理が行われていることが分かる．

図 2 に実際の MapReduce 処理のチャートを示す．縦軸はタスク番号，横軸は処理開始時刻からの経過時間となっている．実験環境は 5 章と同一であり，`wordcount` ワークロード（`small` セット），Hadoop のカウンタを用いて結果を取得した．なお，結果を分かりやすくするため，YARN のノードは 1 台に制限した．

図 2 の結果から，いくつかのことを確認することができる．まず，Map タスクおよび Reduce タスクともに処理完了時刻がばらついている．特に，Map タスクにおいては，ほぼ同時刻に処理を開始しているが，このようにばらつい

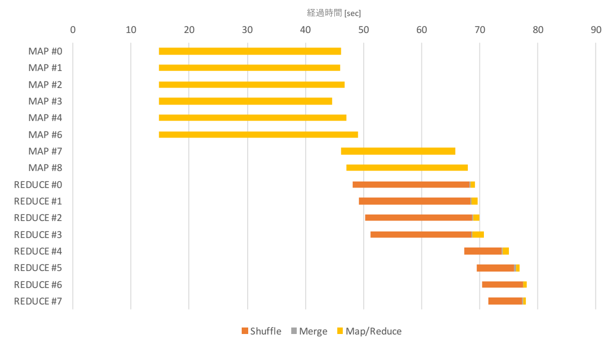

図 2　MapReduce 処理の例

ている．本結果は straggling tasks が存在しない場合であり，もし存在する場合には，より大きな差となる．

次に，同一の Map あるいは Reduce タスク処理であるが，複数の**ウェーブ**（wave）で構成されることがあることが分かる．これは今回の実験環境の YARN の空きスロットが最大 6 であり，同時処理タスクが 6 タスクを超える場合に，処理が遅延されていることが分かる．

最後に，各フェーズの処理であるが，Map および Reduce フェーズ以外であっても，比較的大きな割合を占めていることが分かる．図 2 はワークロードの大きさを `small` にした場合の結果であるが，無視できない大きさの時間であることが分かる．また，図 2 には存在しないが，**バックアップタスク**が実行されることがあり，また，複数の利用者によって複数の MapReduce **ジョブ**が同一のクラスタ環境で同時に実行されることがある．さらには，Oozie などの**ワークフロー**を用いた場合には，各回に MapReduce 処理を行う複数の**ステージ**で構成されることがある．

2.2 リソース競合の仮定

バージョン 2 以降の Hadoop（MapReduce2）では，各 Map/Reduce タスクへのリソースの割り当てに YARN を使用する．YARN の標準スケジューラでは，各ノードの仮想コア数および仮想メモリ量をもとに，空きスロットに Map/Reduce タスクを割り当てることで，スケジューリングを行う．

標準の YARN スケジューラはよく機能するものの，いくつかの点で問題がある．まず，標準のスケジューラでは，ディスクやネットワークといった共有リソースが考慮されていない．多くの Map/Reduce タスクは I/O インテンシブのため，これらのリソースがボトルネックとなることがある．また，CPU とメモリの既に考慮されているリソースであっても，リソース競合が発生することがある．例えば，CPU に関しては仮想コア数のみを考慮しているため，同一物理コアを共有する場合には，リソース競合が発生しうる．

関連研究として，MapReduce を対象としたさまざまな

ⓒ 2017 Information Processing Society of Japan

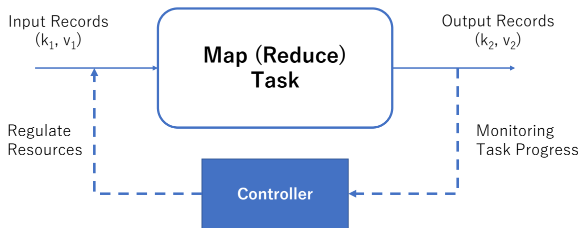

図 3 本スケジューリングの概要

改良スケジューラが提案されているが [9], [10], [11]，本研究は YARN の標準スケジューラを改良せず，ノード・レベルのスケジューラを追加することでリソース競合の改善を試みる．

3. 提案

本論文のアイデアは非常にシンプルである．Hadoop の各タスクの**進捗度**（progress）を取得し，進捗の遅れているタスクに対して，より多くのリソースを割り当てる．以上により，**進捗度に基づくスケジューリング**（progress-based scheduling）[12], [13] を実施する（**図 3**）．

本スケジューリングにおいて中心的話題となるのは，(1) タスク進捗度の取得と，(2) リソース制御である．3.1 節で本論文で仮定する性能モデルを説明したのち，3.2 節と 3.3 節でそれぞれ解説する．

3.1 性能モデル

本論文では，文献 [14] を参考に，下記の性能モデルを仮定する．本モデルを説明のために使用する．

タスク k の時刻 t における進捗度を $progress_k(t)$ とした場合に，各タスクの進捗度は下記で与えられる．

$$progress_k(t) = \frac{t - t_{start}}{\tilde{e}_k(t) - t_{start}} \times 100 \quad (1)$$

なお，t_{start} はジョブ全体の開始時刻，$\tilde{e}_k(t)$ はタスク k の予想に基づく終了時刻である．

予想終了時刻 $\tilde{e}_k(t)$ は，下記の 2 つの要素に分解することが可能である．

$$\tilde{e}_k(t) = p_k(t) + \tilde{r}_k(t) \quad (2)$$

$p_k(t)$ はこれまでの処理時間の実測値，$\tilde{r}_k(t)$ は予想残り処理時間である．$p_k(t)$ に実測値を用いることができる点で，より正確な予測であることが期待される．

予想残り時間 $\tilde{r}_k(t)$ は，下記の式で表現することができる．

$$\tilde{r}_k(t) = \tilde{f}_k(D_k(t), R_k(t)) \quad (3)$$

$D_k(t)$ は時刻 t の残りの入力となるレコードの集合，$R_k(t)$ はタスク k が時刻 t で使用できるリソースの集合である．なお，Map と Reduce のタスクの種類により，$D_{k,Map} = \{(k, v)\}$ または $D_{k,Reduce} = \{(k, V_k)\}$ となる．リソース集合が追加されている点で，文献 [14] の成果を拡張している．

また，$\tilde{f}_k$ が与えられた場合に，$D_k(t)$ にデータの偏りがあっても本方式では調整することができず，$R_k(t)$ の制御のみで残りの処理時間を調整することとなる．

一般に，予測の根拠となる $\tilde{f}_k$ を決定することは難しい．非線形性を有しており，決定するためには十分な事前プロファイル情報が必要なこともある．そのため，近似的に処理時間は key-value ペアの value 集合の大きさ $|V_k|$ で決まるとしている関連研究 [14] もある．

$$\tilde{f}_k(\{(k, v)\}, R_k) \approx \tilde{f}_k(|V_k|, R_k) \quad (4)$$

3.2 タスク進捗度の取得

本論文の手法は，正確な**進捗指標**（progress indicator）に依存している．不正確な進捗指標はリソース使用の過多や不足（over-/under-utilization）となり，かえって性能低下を招く．

また，進捗度は原則として，アプリケーションに依存した指標であるため，進捗度の取得には Hadoop との連携が必要である．

3.2.1 進捗度の計算

Hadoop の進捗指標として，さまざまなものが提案されている [14], [15], [16]．標準の Hadoop でも jobs，tasks，attempts それぞれの進捗指標が提供されているが，あまり正確ではないことが知られている．

現在の先端的な進捗指標に関する研究成果が文献 [14] である．残り処理時間の予測を，具体的な key ごとに追跡しており，また処理時間が非線形となる場合にも対処できるとしている．本手法は非常に正確な進捗指標を提供するものの，key 単位で処理時間を追跡するため，内部管理情報が大きく，複雑な実装となる．実際，頻度の高い key 集合のみの情報追跡に制限するなど，データ量削減のための最適化も行われている．

より簡易的な進捗指標として，文献 [15], [16] が上記の研究に先立って提案されている．元々，進捗指標はデータベース分野で長時間 SQL クエリの完了時間予測のために研究されてきたが，Hadoop にこれらの成果を応用している．

これらの研究では，タスク k の残りの処理時間の予測 $\tilde{r}_k(t)$ を次の式で計算する．

$$\tilde{r}_k(t) = s_k \alpha_k |D_k(t)| \quad (5)$$

$|D_k(t)|$ は入力レコードのサイズである．α_k は事前プロファイルにより決定し，入力レコード数に対する処理時間の線形性を仮定する．また，一定の実行時のリソース変動を考慮するため，s_k を導入している．

本研究では，進捗指標の正確さと実装の容易さのトレードオフを考慮し，後者の方式を採用する．本研究の目的は，

ⓒ 2017 Information Processing Society of Japan

進捗指標そのものではないことから，進捗指標を簡易的に実装し，進捗度に基づくスケジューリングが有効であることを，まずは確認する．

上記に基づき進捗指標を計算するためには，入力となるレコード数の取得が必要である．本研究では，Hadoop の REST API [17] を使用し，Hadoop が提供するカウンタ情報からレコード数を取得することをを試みた．しかしながら，REST API で提供される情報の周期は非常に粗く，Hadoop のデフォルト進捗度も 0%から 100%に急に変化することも多い．この周期を変更する `mapreduce.task.{merge,combine}.progress.records`, `mapreduce.client.{progressmonitor,completion}.pollinterval` などのパラメータが提供されているが，効果が得られなかった．よって，Hadoop 自体の修正を行い，Hadoop 内部の情報を取得する工夫が必要であると思われる．

3.2.2 進捗度の推測

Hadoop の標準的なカウンタ情報から，進捗度のヒントとなる情報が十分に得られないのであれば，CGroups で取得できる情報から，各タスクの進捗度を推測するアプローチもあり得る．

一つの方法として，ディスクへの I/O 量をもとに進捗度を推測するアプローチがあり得る．事前にプロファイル情報を取得しておき，それと比較して，現在の進捗度を予測する．このアプローチの欠点として，ディスク I/O の量は必ずしも入出力レコードに対する進捗に比例しないため，不確かな推測となる可能性がある．しかしながら，CGroups の情報を直接利用するため，Hadoop やアプリケーションを修正する必要がなく，また高い頻度で情報を取得することも可能である．

3.3 リソース制御

CGroups は，**表 1** を例として，CPU，メモリ，ディスク，ネットワークを含む，さまざまなリソースに対する制御能力を提供する．しかしながら，複数のリソースに対して制御できる場合に，どのリソースをどの程度，制御するのか，決定することは難しい．

例えば，時刻 t におけるジョブ全体の基準となる進捗度 $\overline{progress(t)}$（タスク全体の平均進捗度など）が与えられたとして，下記の差分 $\Delta_k(t)$ から，

$$\Delta_k(t) = progress_k(t) - \overline{progress(t)} \quad (6)$$

各リソースの調整量 $\delta_{k,CPU}(t)$，$\delta_{k,mem}(t)$，$\delta_{k,disk}(t)$，$\delta_{k,net}(t)$ を決定することは難しい．

そこで本研究では，単一のボトルネックとなるリソースを仮定する．ボトルネックとなるリソースが存在すれば，タスクの進捗はそこで制御できるはずである．本論文では，ディスク I/O がボトルネックであると仮定する．これは必ずしも真ではないが，多くの場合，Map/Reduce タスクの入出力はディスクに依存しており，そこでタスク進捗の制御が可能であると推測される．

ディスク I/O の制御は，図 1 のローカルディスクに対する I/O に対してのみ行われる．一般的に，HDFS が Hadoop と組み合わせて用いられているが，HDFS へのアクセスは Map フェーズの入力，Reduce フェーズの出力で行われ，中間データはローカルディスク上に配置される．このローカルディスクのアクセス先は通常，`hadoop.tmp.dir` で指定されている．

3.4 その他の事項

本研究は，現在，同時に実行されるジョブは一つと仮定している．これは同一の利用者あるいは他の利用者によって，並行して実行されるジョブは存在しないとしており，実際の Hadoop 利用環境と比較すると，強い仮定である．また，各タスクに対して，単一のウェーブのみを仮定している．これらを考慮する場合，タスクのより詳細な属性情報の把握が必要であり．今後の課題とする．

また，本研究では，均一な速度で全てのタスクが実行された場合が最も高速であると仮定している．これは必ずしも正しいとは限らない．本研究のアプローチが有効な範囲の確認については，今後の課題とする．

4. 実装

本実装では，YARN の CGroups サポート [18] を利用する．CGroups サポートは，YARN のスケジュール単位である YARN コンテナを CGroups のコンテナとして実行する機能である．本機能により，Map および Reduce の各タスクを CGroups のコンテナ内で実行し，リソース使用量の監視や制御を行うことができるようになる．しかしながら，標準の CGroups サポートでは，CPU リソースに対する制限のみにしか対応していないため，著者らでメモリ，ディスク，およびネットワークに対しても制限できるように拡張している．

具体的には，`CgroupsLCEResourcesHandler` を修正した．本変更は，プラグインとして交換可能な部分のみの修正であり，Hadoop 本体は依然として変更していない．よって，アプリケーションおよび Hadoop は不可侵（non-intrusive）であるという性質は保たれる．

本実装は先行研究 [19] でも使用しており，今回はタスク処理を遅らせるという目的ではなく，高速化を目的として使用する．また，予備実験として Spark の YARN モードに対しても本実装を適用し，タスク単位のリソース監視や制限が可能であることを確認している．

上記以外の事前のプロファイラ取得，スケジューリングのためのコードはすべて Python で記述している．Hadoop のジョブ実行は一般的に長時間にわたるため，記述言語に

ⓒ 2017 Information Processing Society of Japan

表 1　CGroups で指定可能なパラメータの例

リソース	パラメータ	説明
CPU	cpu.shares	CPU 時間の相対配分を指定する．
	`cpu.cfs_period_us`，`cpu.cfs_quota_us`	CPU を period 時間のうち quota 時間だけ，使用可能とする．
メモリ	`memory.limit_in_bytes`	メモリ使用量の上限を指定する．
ディスク	`blkio.weight`	ディスク I/O の重みを指定する．
	`blkio.throttle.{read,write}_bps_device`	ディスク I/O レートの上限を bytes/sec で指定する．
	`blkio.throttle.{read,write}_iops_device`	ディスク I/O レートの上限を IOPS で指定する．
ネットワーク	`net_cls.classid`	クラス ID を割り当て，Linux `tc` などの外部の機構で制御可能とする．

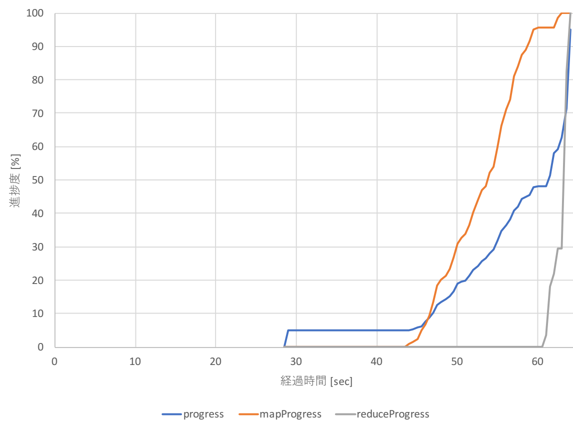

図 4　ジョブ全体の進捗度（`wordcount`，`large`）

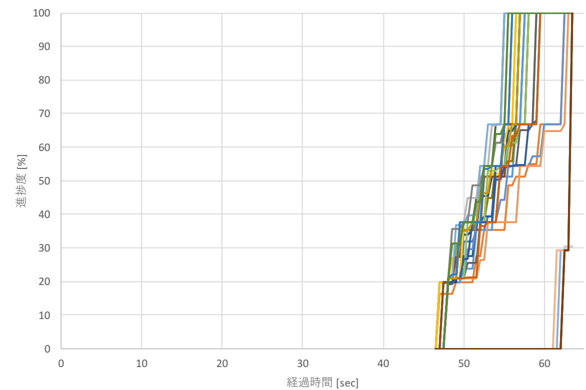

図 5　各タスクの進捗度（`wordcount`，`large`）

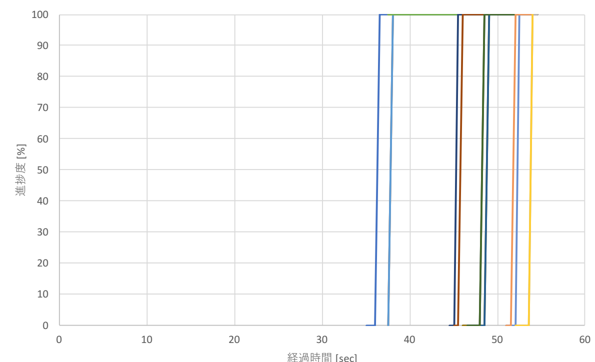

図 6　各タスクの進捗度（`sort`，`large`）

よるオーバヘッドは大きな問題とはならない．

5.　予備実験

5.1　実験環境

実験では，北海道大学アカデミッククラウドの M サーバを 4 台使用した．M サーバは XenServer 上の仮想マシンとして提供されており，Xeon E7-8820（2.4GHz）の CPU から 4 コア，8GB のメモリ，1TB のディスク領域が FibreChannel 接続ストレージから割り当てられている．ネットワーク接続は，ハードウェア筐体あたり 10Gbps Ethernet が 2 本となっている．

ソフトウェアとしては，CentOS 7（Linux 3.10.0），OpenJDK 1.8.0，Hadoop 2.8.0 を使用した．ベンチマークとして HiBench 6.0 [20] を使用し，ワークロードの大きさは実験に応じて，`small` または `large` を使用した．

5.2　実験結果：進捗度の取得

本論文が目標とする手法は，各タスクの進捗指標の正確さに依存している．まずは，Hadoop が提供する進捗指標や入出力レコード情報の正確さを確認する予備実験を行なった．

図 4 にジョブ全体の進捗度，**図 5** にタスク単位での進捗度を取得した結果を示す．まず，ジョブ全体の進捗度は各タスクの進捗度を平均化した形となっている．図 5 を見ると，各タスクの進捗度はばらつきがあることが分かる．Hadoop が提供する各タスクの進捗度はステップ状となっており，多少，粒度の粗い情報と言える．

図 5 の結果から，Hadoop が標準で提供するタスク進捗度が活用できそうであるが，必ずしも十分な情報が提供されるとは言えない．**図 6** に `sort` ワークロードでの結果を示す．この結果では，各タスクの進捗度は 0%から 100%に急に遷移している．中間のタスク進捗度が提供されるかどうかは，アプリケーションに依存する．

Hadoop が標準で提供する進捗度の精度が十分でないのであれば，入力レコード数の情報を取得し，そこから進捗度を予測する方法も考えられる．**図 7** に Map および Reduce の入力レコードの取得結果を示す．本実験は Hadoop の REST API を使用し，Hadoop の標準カウンタ値を取得している．なお，図 4 および図 5 とは異なる実行となっている．

図 7 を見ると，入力レコード数は各タスクにおいて一度

ⓒ 2017 Information Processing Society of Japan

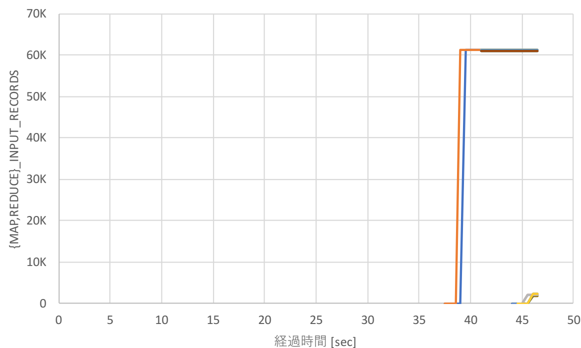

図 7　各タスクの入力レコード数（wordcount，large）

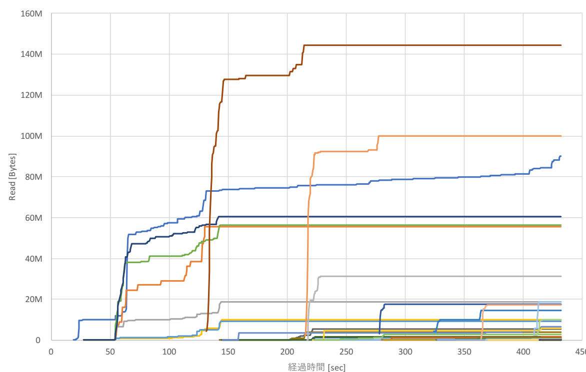

図 8　各タスクの Read アクセス（wordcount，large）

だけ遷移しており，タスク進捗度を予測するには，情報が十分ではないことが分かる．

5.3　実験結果：CGroups による I/O 情報の取得

また，CGroups から取得できる情報を活用し，そこから各タスクの進捗度を推測するアプローチもあり得る．

M サーバにおいて，SAN 接続のストレージシステムの何らかの理由により，CGroups のディスク I/O 情報が正確に取得できなかったことから，本実験では Core i7（2.4GHz）CPU，16GB メモリ，NVMe 接続 SSD の PC を使用した．VMware Fusion 7.1.3 を利用し，Hadoop には 2 コア，4GB メモリの仮想マシン 1 台を割り当てた．

図 8 に CGroups による Read アクセスの取得結果を示す．図 5 の結果と比較すると，同様のステップ状の変化となっているが，中間のより詳細な情報が得られている．しかしながら，全体の Read アクセスの総量は分からないため，事前プロファイル取得などの工夫により，推測する必要がある．

6.　まとめと今後の課題

本研究では，Hadoop を対象として，第 2 階層スケジューラの導入によるタスク進捗度の平準化を検討した．本論文の手法では，各タスクの進捗度を進捗指標により監視し，処理が遅れているタスクにより多くのリソースを割り当てることで，進捗の平準化を目指す．各タスクが使用するリソースの監視や制御には，YARN の CGroups サポートを用いる．

今回の実験では，本研究の前提となる精度の高いタスク進捗情報が得られなかった．Hadoop が標準で提供する情報では精度が不足しているため，Hadoop を修正し，内部レコード情報を活用することを検討する．一旦，進捗度が取得できれば，CGroups によるリソース制御は容易であることが期待される．Hadoop の進捗は CGroups を経由したリソース制御により，調整可能であることを確認しており，Spark に対しても適用できることを確認している．よって．今後は本格的な実装や実験に着手するとともに，Spark への適用を目指す．

参考文献

[1] The Apache Software Foundation: Apache Hadoop, `http://hadoop.apache.org/`.
[2] The Apache Software Foundation: Apache Spark: Lightling-fast Cluster Computing, `http://spark.apache.org/`.
[3] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *USENIX OSDI'04*, Berkeley, CA, USA, pp. 10–10 (2004).
[4] X. Meng et al.: MLlib: Machine Learning in Apache Spark, *Journal of Machine Learning Research*, Vol. 17, No. 1, pp. 1235–1241 (2016).
[5] S. Venkataraman et al.: Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics, *USENIX NSDI'16* (2016).
[6] B. Hindman et al.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, *USENIX NSDI'11* (2011).
[7] Schwarzkopf, M. and Konwinski, A.: Omega: Flexible, Scalable Schedulers for Large Compute Clusters, *ACM EuroSys'13* (2013).
[8] White, T.: *Hadoop: The Definitive Guide, 4th Edition*, O'Reilly Media, Inc. (2015).
[9] 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).
[10] 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).
[11] 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).
[12] D. C. Steere et al.: A Feedback-driven Proportion Allocator for Real-Rate Scheduling, *USENIX OSDI'99* (1999).
[13] Douceur, J. R. and Bolosky, W. J.: Progress-based Regulation of Low-Importance Processes, *ACM SOSP'99* (1999).
[14] Coppa, E. and Finocchi, I.: On Data Skewness, Stragglers, and MapReduce Progress Indicators, *ACM SOCC'15* (2015).
[15] K. Morton et al.: Estimating the Progress of MapReduce Pipelines, *IEEE ICDE'10* (2010).

ⓒ 2017 Information Processing Society of Japan

[16] K. Morton et al.: ParaTimer: A Progress Indicator for MapReduce DAGs, *SIGMOD'10* (2010).
[17] The Apache Software Foundation: Hadoop YARN - Introduction to the web services REST API's，https://hadoop.apache.org/docs/r2.8.0/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html.
[18] The Apache Software Foundation: Using CGroups with YARN, https://hadoop.apache.org/docs/r2.8.0/hadoop-yarn/hadoop-yarn-site/NodeManagerCgroups.html.
[19] 岩井良成，杉木章義，棟朝雅晴：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，情報処理学会研究報告 (2017-OS-140(13)) (2017).
[20] Intel: HiBench Suite, https://github.com/intel-hadoop/HiBench.

ⓒ 2017 Information Processing Society of Japan