データ転送量を考慮した
Hadoop 性能改善方式の提案と評価

筑波大学
図書館情報メディア研究科
2014 年 3 月
渡邊 飛雄馬

目次

図表目次

第1章 序論

近年、膨大なデータを分析し、情報を抽出しようという取り組みが広く行われている。Yahoo, Google, Facebook や Amazon 等に代表されるさまざまな Web サービスは、一日に数十テラバイトから数ペタバイトのログデータを分析し、ユーザの行動分析によるサービス改善や、システムの保守運用に活用している。[1] [2] また、Web サービスにとどまらず、自動車プローブデータに基づく渋滞解析 [2] や、日々の電力利用状況に基づいたサービス提案[3]など、ユビキタスネットワーク社会の生成するデータの活用事例も増加してきている。このような背景を受けて、ビッグデータと呼ばれるこのような巨大なデータを高速に効率よく分析する仕組みの研究が盛んである。

現在広く使われているビッグデータを分析するためのフレームワークに MapReduce [4] がある。 MapReduce は、Map と Reduce の 2 段階でデータを処理する分散処理フレームワークであり、2004 年に Google によって発表された。現在、この MapReduce フレームワークのオープンソース実装である Hadoop [5] が一般的に用いられており、Facebook や Yahoo、New York Times 等の企業においても実際に利用されている [1]。

MapReduce フレームワークでは、与えられた仕事 “Job”を複数の小さな処理単位である “Task” に分割し、Map と Reduce と呼ばれる 2 つの段階に分けて複数台の汎用計算機（ノード）で並列処理をすることによって膨大なデータを高速処理する。Map 時の処理を行う Task を Map task、Reduce 時の処理を行う Task を Reduce task と呼ぶ。Map task は分割された入力データ (Input Data Split) 一つ一つに対して処理を行い、Reduce task は Map task の出力を受け取り、集計や並び替えなどのとりまとめ処理を行う。ユーザは、Map task と Reduce task での処理を関数として定義し、入力データと組み合わせて MapReduce クラスタに投入することで Job を実行させる。投入された Job は、Job の進行を管理する 1 台の JobTracker によって複数の Task に分割され、複数台の TaskTracker に割り当てられる。この Task の割り当てを Task scheduling と呼ぶ。Hadoop の Task scheduling は、TaskTracker が同時実行可能な Task 数に基づいて行われる。TaskTracker が同時実行可能な Map と Reduce task の最大数は、Task slot として予めクラスタ管理者が TaskTracker 毎に設定しておく必要があり、一般的には TaskTracker の性能に応じて決定する。Job を受け取った JobTracker は、各 TaskTracker の Task slot 数を超えないように Task scheduling を行う。

Hadoop をはじめとする、MapReduce フレームワークの性能向上の研究は盛んである。文献 [6] では、MapReduce クラスタ内におけるデータのブロードキャストによるネットワーク負荷を低減するために、BitTorrent に似た方式の利用を提案している。また、同論文では、同時並行で実行されている Job 間のネットワーク帯域割り当て公平性のためのフロー制御方式も提案されている。文献 [7] [8] では、Map task への Input Data Split のネットワーク転送を極力避けるために、Input Data Split をあらかじめ所持している TaskTracker 上で Map task を実行する “Data Local Map task” の割合を向上させる Task 割り当て方式 “Delay Scheduling” が提案されている。そのほか、Hadoop の性能向上を目的とした研究に、“Copy Compute Splitting” [8] がある。これは、ある Job の Map task がすべて終了し、Map task の出力する中間データを受信しない限り Reduce task が

計算処理にはいれず、TaskTracker の Reduce slot を何もしないまま消費してしまう問題“Slot hoarding”を解決するために、Reduce Task を 2 つの段階に分割して Reduce task slot を割り当てる方式である。また、Task の投機的実行の条件を変更、優先度を設定し、無駄な投機的実行生成を防ぎ、効率的な投機的実行 Task の生成を行うことによる Hadoop 性能向上方式 “LATE scheduler” [9] の提案もある。

一般的に、MapReduce は膨大なデータを扱うため、クラスタ内でのデータ転送がボトルネックになるといわれている。文献 [10] では、実際運用されている MapReduce クラスタ内において、ネットワーク輻輳が大きなボトルネックとなっていることが示されている。また、Orchestra の論文では、Hadoop の Map task と Reduce task 間に発生するデータ転送時間が Job 実行時間の 50%を占めている状況が報告されている。また、私の予備実験により、ネットワーク輻輳だけでなく、TaskTracker の HDD（二次記憶装置）I/O 輻輳が Job 実行時間に大きく影響していることが判明した。しかし、上述したように、データ転送時間の低減に着目した研究の数はあまり多くない。また、Hadoop のような分散処理システムは、クラスタ内の負荷が偏ることによってその性能が著しく低下するにもかかわらず、TaskTracker のネットワークや HDD 負荷状況を考慮した Task scheduling 方式は未だ行われていない。

そこで本論文では、Hadoop の Job 実行時間の短縮を目的として、TaskTracker のデータ転送量を考慮した Task scheduling 方式を 3 種類提案する。“Receive Rate Scheduling” は、TaskTracker のネットワークインターフェースの受信帯域が閾値を超えていた場合に Data Local Map task と Reduce task を割り当てない Task scheduling 方式である。“Disk Rate Scheduling” は、TaskTracker の HDD I/O 待ち時間の平均が閾値を超えていた場合に Map task の割り当てを避ける。そして、“Potential Reception Scheduling” は、Reduce タスクが、TaskTracker に割り当てられている Reduce task がまだ受信完了していない Map 中間データ量の合計が閾値を超えた場合、その Task Tracker に Reduce Task を割り当てない Task scheduling 方式である。

そして、これらの Task scheduling 方式を組み合わせ、研究室内で構築したクラスタと、Amazon EC2 [11] のクラウド環境上に構築したクラスタの 2 種類の Hadoop クラスタ上で評価実験を行い、提案方式が既存の Task scheduling 方式と比較して平均して約 5%の性能向上を実現したことを示す。また、予備実験により判明した Job 実行速度が低下する仕組みの解説を行う。

以降、本論文は以下のように構成される。第 2 章では、Hadoop の動作概要と、ネットワーク I/O と HDD I/O の発生タイミングを説明する。第 3 章では、Hadoop の Job 進行過程の分析と、TaskTracker の各リソース使用率の分析を行う目的で作成したログ解析アプリケーション “HadoopProcessAnalyzer” を紹介する。予備実験と評価実験に用いた Hadoop クラスタと Hadoop Job の解説を第 4 章で行い、Job 実行速度低下メカニズムの解説を第 5 章にて行う。第 6 章では 3 種類の提案方式の詳細を解説する。そして提案方式の性能評価は第 7 章で行い、結果の考察と残る課題の検討を第 8 章で行う。最後に結論を第 9 章で述べる。

第2章 Hadoop の概要

2.1. Hadoop の構成

Hadoop クラスタは、Hadoop をインストールした汎用的なコンピュータをネットワークで相互接続して構成する。Hadoop は大きく分けて、分散ファイルシステムである HDFS と、その上で動作する MapReduce エンジンの 2 要素から構成される。Hadoop の構成を図 1 に示す。

HDFS は 1 台の NameNode と複数の DataNode の Master-Slave 構造となっている。NameNode はファイルシステムの名前空間を管理し、HDFS 上にあるすべてのファイルブロックの場所を管理している。ユーザや MapReduce エンジンが HDFS とファイル入出力を行う際には、まず NameNode に問い合わせを行うことで実際のファイルの位置を得る。DataNode は実際にデータを分散保存している。データの到達性と耐障害性のために、HDFS はデフォルトで 3 つの DataNode に同じブロックを複製している。

MapReduce エンジンも同様に、1 台の JobTracker と複数の TaskTracker の Master-Slave 構造を持っている。JobTracker はユーザから Job を受付、Job を Task に分解、TaskTracker に Task の割り当てを行い、Job の進行を管理する。そして TaskTracker は割り当てられた Task を実際に処理する。

一般的な環境では、HDFS の DataNode と MapReduce エンジンの TaskTracker は同一のノードに配置される。これは、TaskTracker 自身が入力データを持った Map task (Data Local Map task) が発生し、ネットワーク経由のデータ転送を減らすことができる、Reduce task の結果出力時のデータ転送を減らすことができる、などの利点を持つ。
Hadoop の MapReduce エンジンは、HDFS 以外のファイルシステムを使用可能ではあるが、一般的な用途では HDFS を使用することが多く、Data Local Map task の優先割り当て機能など、MapReduce エンジンの機能の一部は HDFS を使用することを前提としている。

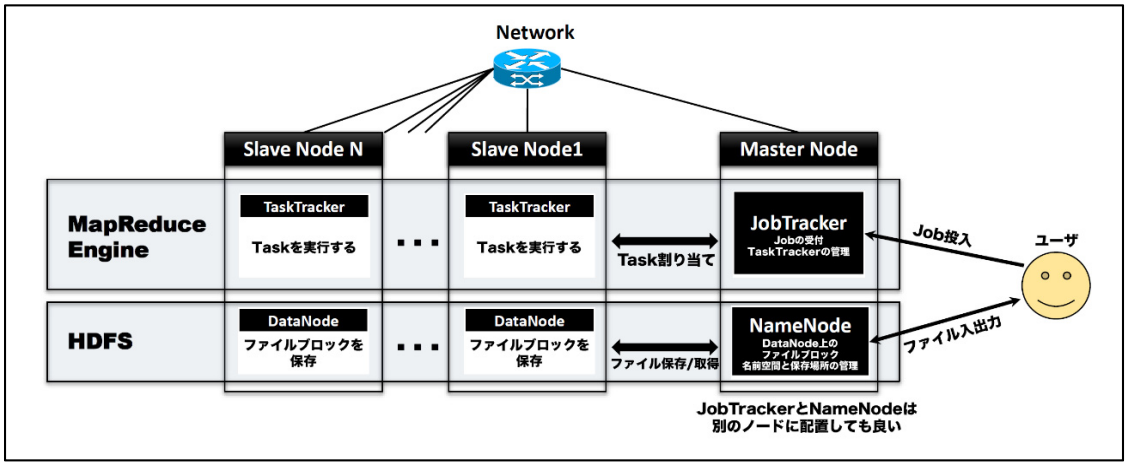


図 1, Hadoop 構成図

2.2. MapReduce 処理過程

MapReduce は巨大なデータに対して分散処理を行うためのフレームワークであり、ユーザにより定義される Map と Reduce の 2 つの関数に基づいて処理を行う。MapReduce の処理の流れを図 2 に示す。

ユーザによって Job が投入されるとまず JobTracker が入力データを Input Data Split に分割し、その Input Data Split を処理する Map task を生成、そして TaskTracker に割り当てる。このとき、Map task が Data Local でなかった場合、TaskTracker は Map Input Data Split のファイルブロックを所持している他のノードからネットワーク経由でデータを受信する。処理を終えた Map task は、その出力を、TaskTracker のローカルディスク上に書き出す。(注: HDFS 上ではない)

その Job の Map task の一部（初期設定では 5%）が終了すると、JobTracker は Reduce task の割り当てを開始する。Reduce task は、Job の全 Map task の出力データ（Map 中間データ）から自分の担当分を受信した後に、Reduce 処理（Reduce 関数）を実行する。つまり、Reduce task は全ての Map task が終了するまで Reduce 処理を開始できず、Job 実行時間に大きく影響しやすい。この現象は 5 章で詳しく示す。Reduce 処理を終えた Reduce task は、その出力データを HDFS 上に保存して終了する。

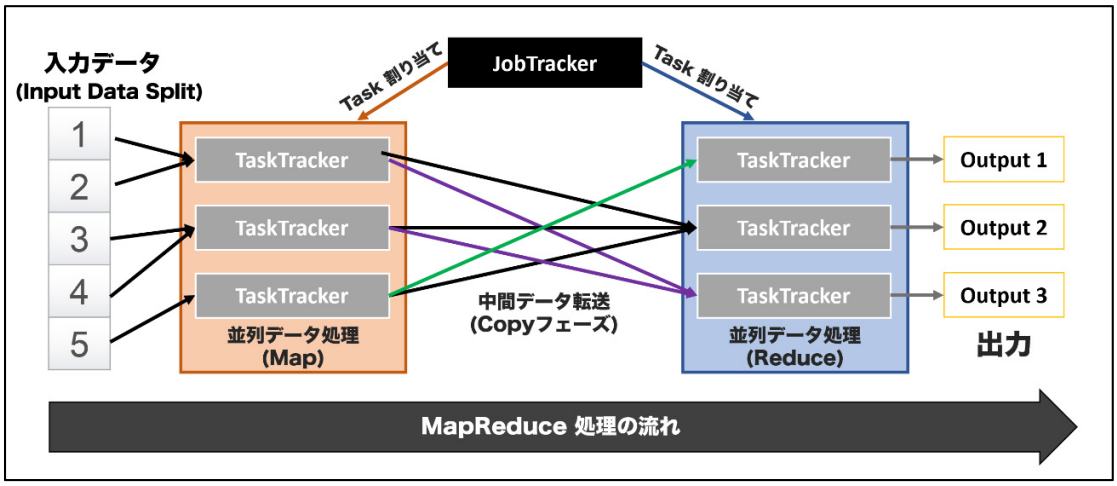


図 2, MapReduce 処理の流れ

2.3. Hadoop の Task scheduling とその問題点

JobTracker が TaskTracker に Task 割り当てを行うことを Task scheduling という。JobTracker と TaskTracker は、HeartBeat と呼ばれる機構で通信しあっており、Task scheduling もこの仕組みを用いて行う。まず、TaskTracker が HeartBeat を用いて JobTracker に Task 要求を行い、JobTracker はこれに対するレスポンスを用いて Task の割り当てを行う。初期設定では、TaskTracker は 5 秒ごとに JobTracker に HeartBeat を送信し続け、これは TaskTracker の生存確認としても利用されている。Hadoop の Task scheduling は、TaskTracker が同時実行可能な Task 数に基づいて行われる。ある TaskTracker が同時実行可能な Map と Reduce task の数は Task slot として予めクラスタ

管理者が TaskTracker 毎に設定しておく必要があり、一般的には TaskTracker の性能に応じて決定する。TaskTracker は、Map と Reduce それぞれの Task slot 数と、現在実行中の Task 数を HeartBeat に含めて送信する。JobTracker はこれを受けて、各 TaskTracker の Task slot 数を超えないように割り当てを行う。図 3 にその仕組みを示す。

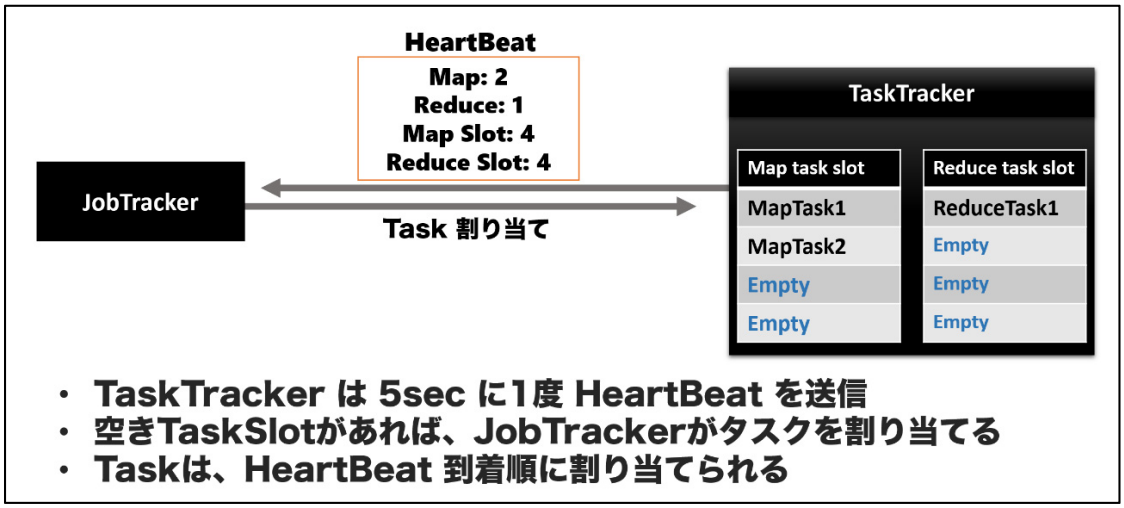


図 3, HeartBeat の仕組み

現在の Hadoop には以下の 3 つの Task Scheduler が実装されている。

- FIFO Scheduler
  Hadoop デフォルトのスケジューラであり、Job が投入された順に処理を実行する。Map と Reduce それぞれの Task 種別において、先に投入された Job の Task の割り当てが終わるまで、次の Job の Task が割り当てられることは無い。また、Job に対して優先度を設定し、それに応じて Task scheduling を行うことができるが、マルチユーザに対応した Job 実行の優先付けには対応していない。

- Fair Scheduler [8]
  マルチユーザ向けの Task Scheduler であり、全てのユーザに対して重み付け公平な Task slot 割り当てを実現する。また、同一ユーザの Job であっても、FIFO Scheduler とは異なり、投入時期に拘わらずそれぞれの Task が公平に割り当てられる。また、ユーザ間の公平性を維持するための Task プリエンプションをサポートする。

- Capacity Scheduler
  マルチユーザ向けの Task Scheduler であり、ユーザ毎にタスクキューを持つ。キュー毎に使用できる Task slot 数を決定することで、ユーザ間での Hadoop クラスタリソースの共有を行う。また、各キュー内では FIFO Scheduling が行われている。

これらの Scheduler は Job の優先度制御やユーザ間の公平性維持制御を除いて、いずれも予め静的に決定された TaskTracker の Task slot 数を超えないように Task scheduling

を行っているだけである。一方、Job の種類や Task の進行状況、処理対象データ量によって、Task が消費する TaskTracker のリソース（CPU、HDD I/O、ネットワーク I/O 等）の種類や量は異なる。例えば、Data Local Map task は HDD I/O を消費するだけだが、そうでない Map task は受信ネットワーク帯域も消費する。また、Word count や Grep のような Job は、Map task にて単語の抽出を行うため、Map 中間データ転送量が非常に少ない。それに対して Sort や Aggregate 等のような Job は大量の Map 中間データ伝送を発生させる[12] 。このように、Task slot 数を超えないようにするだけの Task scheduling 方式では、実際に各 TaskTracker が消費しているリソースを反映できず、クラスタ内の TaskTracker 間において、大きな負荷の偏りを発生させる可能性がある。一般的に分散処理システムにおいては、クラスタに所属する処理ノードの負荷の偏りを減らすことがスループット向上につながるため、TaskTracker の実際の負荷に基づいた Task scheduling が Hadoop クラスタの性能向上をもたらすと考えられる。

2.4. Task の投機的実行

何らかの理由で一部の Task が、同じ Job の他の Task と比較して非常に時間がかかっている場合、JobTracker は その Task と同じ Task を別の TaskTracker でバックアップとして実行する。これを Task の投機的実行という。元の Task か投機的実行 Task のどちらか一方が先に終了した場合、もう一方の Task は直ちに終了される。Task の投機的実行が行われる条件は以下の通りである。

1. その Job の Task 全体の平均進行状況から 20%以上遅れている
2. Task が開始してから 60 秒以上経っている
3. 他にその Task の投機的実行 Task がない（投機的実行 Task は同時に 1 つまで）

ただし、この投機的実行条件にはいくつか問題があり、最適でない Task の投機的実行や、不必要で過剰な投機的実行による Task slot やリソースの消費などが指摘されている[9]。

2.5. Hadoop のネットワーク I/O

第 1 章で述べたように、Hadoop の Job におけるネットワーク転送の占める割合は大きいといわれている。Hadoop の Job 実行において、大規模なネットワーク I/O が発生する時期は以下の 2 つである。

1. Data Local でない Map task 開始時の Map Input Data Split の受信
2. Reduce task による、全 Map task からの Map 中間データの受信

特に 2 の Map 中間データの受信は、1 対多の通信が同時発生するためにネットワーク輻輳を引き起こしやすいといわれている。また、これらに加え、終了した Reduce task が HDFS 上に書き出した Job の最終出力データの HDFS による複製もネットワークトラヒックを発生させる。

2.6. Hadoop の HDD I/O

また、ネットワーク転送だけでなく HDD I/O も大きなボトルネックとなることが実験によって確認できた。Hadoop の Job 実行において、HDD I/O が発生する時期とその内容は以下の通りである。

1. Map Input Data Split の HDFS からの受信: 読み出し/書き込み
2. Map 関数: 読み出し/書き込み
3. Map 中間データの書き出し: 書き込み
4. Map 中間データの送受信: 読み出し/書き込み
5. Reduce 関数: 読み出し/書き込み
6. Reduce 出力データの書き出し: 書き込み

また、ネットワーク I/O と同様に、HDFS の複製処理は HDD I/O を発生させる。

第3章 HadoopProcessAnalyzer

HadoopProcessAnalyzer は、Hadoop のログと TaskTracker のリソース使用率を可視化するアプリケーションであり、本研究を実施する過程で開発した。

3.1. 開発動機

本研究を実施するにあたって、Hadoop の Job 進行過程の分析と、その過程における TaskTracker のリソース使用率の分析を行う必要があった。Hadoop にデフォルトで実装されている Web インターフェースでは、Job や各 Task の開始・終了時間、実行時間、処理したデータサイズ、Data Local の有無などの情報を得ることができるが、Job の実行時間に対して悪影響を及ぼしている Task の判別や、その Task が遅延した原因を分析することができない。そこで、Job と Task の進行過程、そして TaskTracker のリソース使用率を可視化できるアプリケーションである HadoopProcessAnalyzer を開発した。

3.2. 構造

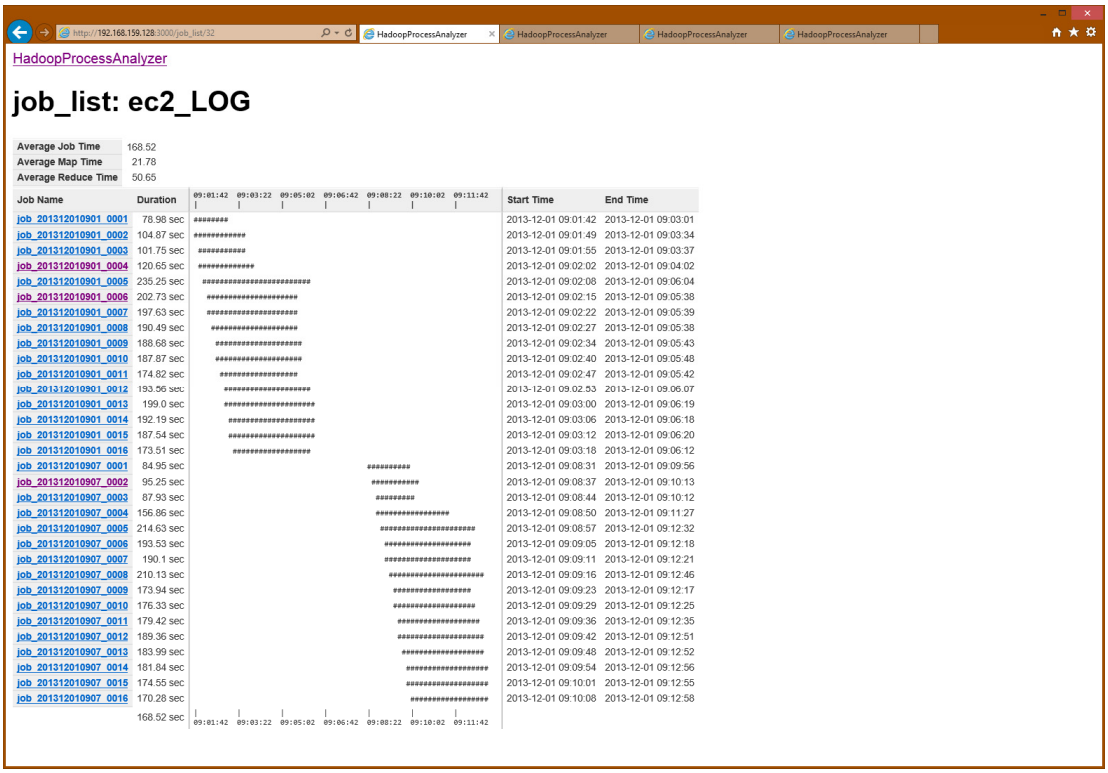
HadoopProcessAnalyzer は、TaskTracker でリソース使用率を監視・ログ生成を行うスクリプトと、Hadoop とリソース使用率のログ解析・可視化を行う Web アプリケーションの 2 つから構成されている。

前者は、Linux 環境で一般的に用いられているリソース監視アプリケーションである iperf を呼び出し、CPU、RAM、HDD I/O、ネットワーク I/O をテキストファイルに書き出すシェルスクリプトである。後者は Ruby on Rails 4.0.2 で構築されており、Hadoop のログファイルとリソース使用率監視スクリプトのログファイルを合わせて読み込ませることで、ブラウザ経由で解析結果を閲覧できる。

3.3. 機能

図 4-7 が Web インターフェースのスクリーンショットである。クラスタで実行された Job 実行時間のガントチャートの表示（図 4)、Job の各 Task 実行時間のガントチャート、Data Local の真偽、Task 完了の真偽、投機的実行 Task の真偽の表示（図 5)、クラスタの各 TaskTracker にて実行された Task の一覧ガントチャート表示と、リソース使用率のグラフ表示（図 6)、そして各 Task の情報と、Task 実行時間内の TaskTracker リソース使用率のグラフ表示（図 7）を行うことができる。

以降で述べる実験結果の分析は、この HadoopProcessAnalyzer を用いて行った。



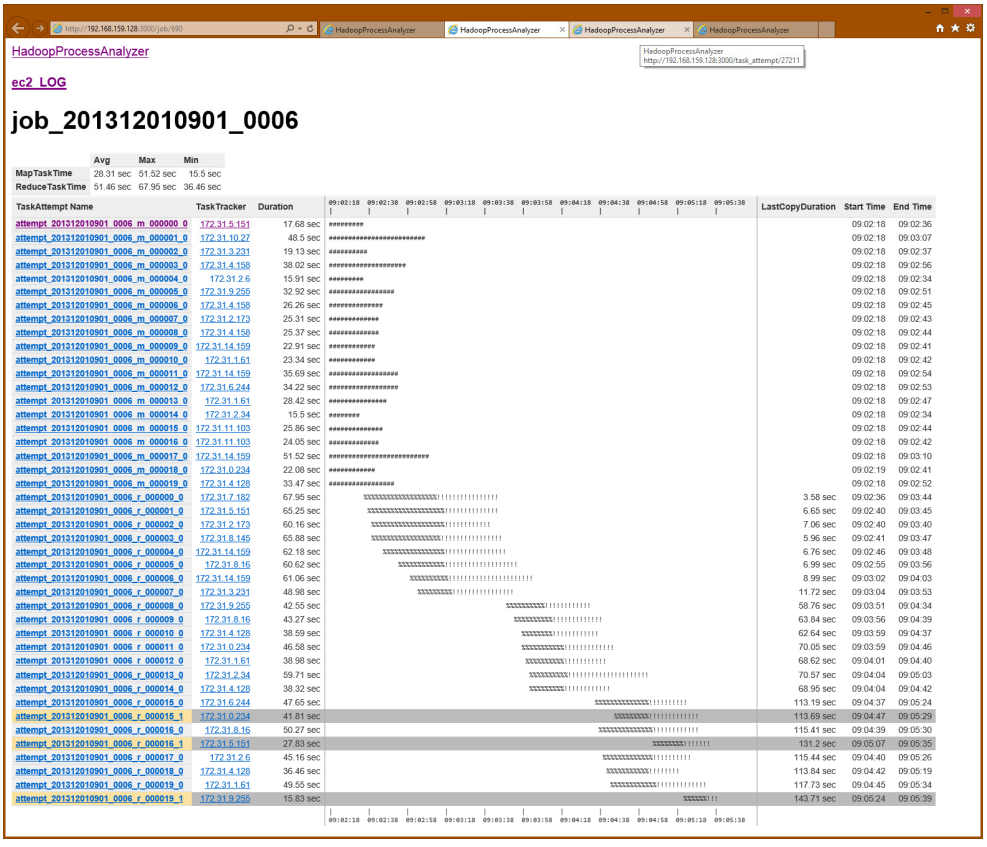
HadoopProcessAnalyzer

job_list: ec2_LOG

Average Job Time	168.52
Average Map Time	21.78
Average Reduce Time	50.65

Job Name	Duration	Start Time	End Time
job_201312010901_0001	78.98 sec	2013-12-01 09:01:42	2013-12-01 09:03:01
job_201312010901_0002	104.87 sec	2013-12-01 09:01:49	2013-12-01 09:03:34
job_201312010901_0003	101.75 sec	2013-12-01 09:01:55	2013-12-01 09:03:37
job_201312010901_0004	120.65 sec	2013-12-01 09:02:02	2013-12-01 09:04:02
job_201312010901_0005	235.25 sec	2013-12-01 09:02:08	2013-12-01 09:06:04
job_201312010901_0006	202.73 sec	2013-12-01 09:02:15	2013-12-01 09:05:38
job_201312010901_0007	197.63 sec	2013-12-01 09:02:22	2013-12-01 09:05:39
job_201312010901_0008	190.49 sec	2013-12-01 09:02:27	2013-12-01 09:05:38
job_201312010901_0009	188.68 sec	2013-12-01 09:02:34	2013-12-01 09:05:43
job_201312010901_0010	187.87 sec	2013-12-01 09:02:40	2013-12-01 09:05:48
job_201312010901_0011	174.82 sec	2013-12-01 09:02:47	2013-12-01 09:05:42
job_201312010901_0012	193.56 sec	2013-12-01 09:02:53	2013-12-01 09:06:07
job_201312010901_0013	199.0 sec	2013-12-01 09:03:00	2013-12-01 09:06:19
job_201312010901_0014	192.19 sec	2013-12-01 09:03:06	2013-12-01 09:06:18
job_201312010901_0015	187.54 sec	2013-12-01 09:03:12	2013-12-01 09:06:20
job_201312010901_0016	173.51 sec	2013-12-01 09:03:18	2013-12-01 09:06:12
job_201312010907_0001	84.95 sec	2013-12-01 09:08:31	2013-12-01 09:09:56
job_201312010907_0002	95.25 sec	2013-12-01 09:08:37	2013-12-01 09:10:13
job_201312010907_0003	87.93 sec	2013-12-01 09:08:44	2013-12-01 09:10:12
job_201312010907_0004	156.86 sec	2013-12-01 09:08:50	2013-12-01 09:11:27
job_201312010907_0005	214.63 sec	2013-12-01 09:08:57	2013-12-01 09:12:32
job_201312010907_0006	193.53 sec	2013-12-01 09:09:05	2013-12-01 09:12:18
job_201312010907_0007	190.1 sec	2013-12-01 09:09:11	2013-12-01 09:12:21
job_201312010907_0008	210.13 sec	2013-12-01 09:09:16	2013-12-01 09:12:46
job_201312010907_0009	173.94 sec	2013-12-01 09:09:23	2013-12-01 09:12:17
job_201312010907_0010	176.33 sec	2013-12-01 09:09:29	2013-12-01 09:12:25
job_201312010907_0011	179.42 sec	2013-12-01 09:09:36	2013-12-01 09:12:35
job_201312010907_0012	189.36 sec	2013-12-01 09:09:42	2013-12-01 09:12:51
job_201312010907_0013	183.99 sec	2013-12-01 09:09:48	2013-12-01 09:12:52
job_201312010907_0014	181.84 sec	2013-12-01 09:09:54	2013-12-01 09:12:56
job_201312010907_0015	174.55 sec	2013-12-01 09:10:01	2013-12-01 09:12:55
job_201312010907_0016	170.28 sec	2013-12-01 09:10:08	2013-12-01 09:12:58
	168.52 sec		

図 4, クラスタで実行された Job の一覧



HadoopProcessAnalyzer

ec2_LOG

job_201312010901_0006

	Avg	Max	Min
MapTaskTime	28.31 sec	51.52 sec	15.5 sec
ReduceTaskTime	51.46 sec	67.95 sec	36.46 sec

TaskAttempt Name	TaskTracker	Duration	LastCopyDuration	Start Time	End Time
attempt_201312010901_0006_m_000000_0	172.31.5.151	17.68 sec		09:02:18	09:02:36
attempt_201312010901_0006_m_000001_0	172.31.10.27	48.5 sec		09:02:18	09:03:07
attempt_201312010901_0006_m_000002_0	172.31.3.231	19.13 sec		09:02:18	09:02:37
attempt_201312010901_0006_m_000003_0	172.31.4.158	38.02 sec		09:02:18	09:02:56
attempt_201312010901_0006_m_000004_0	172.31.2.6	15.91 sec		09:02:18	09:02:34
attempt_201312010901_0006_m_000005_0	172.31.9.255	32.92 sec		09:02:18	09:02:51
attempt_201312010901_0006_m_000006_0	172.31.4.158	26.26 sec		09:02:18	09:02:45
attempt_201312010901_0006_m_000007_0	172.31.2.173	25.31 sec		09:02:18	09:02:43
attempt_201312010901_0006_m_000008_0	172.31.4.158	25.37 sec		09:02:18	09:02:44
attempt_201312010901_0006_m_000009_0	172.31.14.159	22.91 sec		09:02:18	09:02:41
attempt_201312010901_0006_m_000010_0	172.31.1.61	23.34 sec		09:02:18	09:02:42
attempt_201312010901_0006_m_000011_0	172.31.14.159	35.69 sec		09:02:18	09:02:54
attempt_201312010901_0006_m_000012_0	172.31.6.244	34.22 sec		09:02:18	09:02:53
attempt_201312010901_0006_m_000013_0	172.31.1.61	28.42 sec		09:02:18	09:02:47
attempt_201312010901_0006_m_000014_0	172.31.2.34	15.5 sec		09:02:18	09:02:34
attempt_201312010901_0006_m_000015_0	172.31.11.103	25.86 sec		09:02:18	09:02:44
attempt_201312010901_0006_m_000016_0	172.31.11.103	24.05 sec		09:02:18	09:02:42
attempt_201312010901_0006_m_000017_0	172.31.14.159	51.52 sec		09:02:18	09:03:10
attempt_201312010901_0006_m_000018_0	172.31.0.234	22.08 sec		09:02:19	09:02:41
attempt_201312010901_0006_m_000019_0	172.31.4.128	33.47 sec		09:02:18	09:02:52
attempt_201312010901_0006_r_000000_0	172.31.7.182	67.95 sec	3.58 sec	09:02:36	09:03:44
attempt_201312010901_0006_r_000001_0	172.31.5.151	65.25 sec	6.65 sec	09:02:40	09:03:45
attempt_201312010901_0006_r_000002_0	172.31.2.173	60.16 sec	7.06 sec	09:02:40	09:03:40
attempt_201312010901_0006_r_000003_0	172.31.8.145	65.88 sec	5.96 sec	09:02:41	09:03:47
attempt_201312010901_0006_r_000004_0	172.31.14.159	62.18 sec	6.75 sec	09:02:46	09:03:48
attempt_201312010901_0006_r_000005_0	172.31.8.16	60.62 sec	6.99 sec	09:02:55	09:03:56
attempt_201312010901_0006_r_000006_0	172.31.14.159	61.06 sec	8.99 sec	09:03:02	09:04:03
attempt_201312010901_0006_r_000007_0	172.31.3.231	48.98 sec	11.72 sec	09:03:04	09:03:53
attempt_201312010901_0006_r_000008_0	172.31.9.255	42.55 sec	58.76 sec	09:03:51	09:04:34
attempt_201312010901_0006_r_000009_0	172.31.8.16	43.27 sec	63.84 sec	09:03:56	09:04:39
attempt_201312010901_0006_r_000010_0	172.31.4.128	38.59 sec	62.64 sec	09:03:59	09:04:37
attempt_201312010901_0006_r_000011_0	172.31.0.234	46.58 sec	70.05 sec	09:03:59	09:04:46
attempt_201312010901_0006_r_000012_0	172.31.1.61	38.98 sec	68.62 sec	09:04:01	09:04:40
attempt_201312010901_0006_r_000013_0	172.31.2.34	59.71 sec	70.57 sec	09:04:04	09:05:03
attempt_201312010901_0006_r_000014_0	172.31.4.128	38.32 sec	68.95 sec	09:04:04	09:04:42
attempt_201312010901_0006_r_000015_0	172.31.6.244	47.65 sec	113.19 sec	09:04:37	09:05:24
attempt_201312010901_0006_r_000016_1	172.31.0.234	41.81 sec	113.69 sec	09:04:47	09:05:29
attempt_201312010901_0006_r_000016_0	172.31.8.16	50.27 sec	115.41 sec	09:04:39	09:05:30
attempt_201312010901_0006_r_000016_1	172.31.5.151	27.83 sec	131.2 sec	09:05:07	09:05:35
attempt_201312010901_0006_r_000017_0	172.31.2.6	45.16 sec	115.44 sec	09:04:40	09:05:26
attempt_201312010901_0006_r_000018_0	172.31.4.128	36.46 sec	113.84 sec	09:04:42	09:05:19
attempt_201312010901_0006_r_000019_0	172.31.1.61	49.55 sec	117.73 sec	09:04:45	09:05:34
attempt_201312010901_0006_r_000019_1	172.31.9.255	15.83 sec	143.71 sec	09:05:24	09:05:39

図 5, Job のタスク一覧

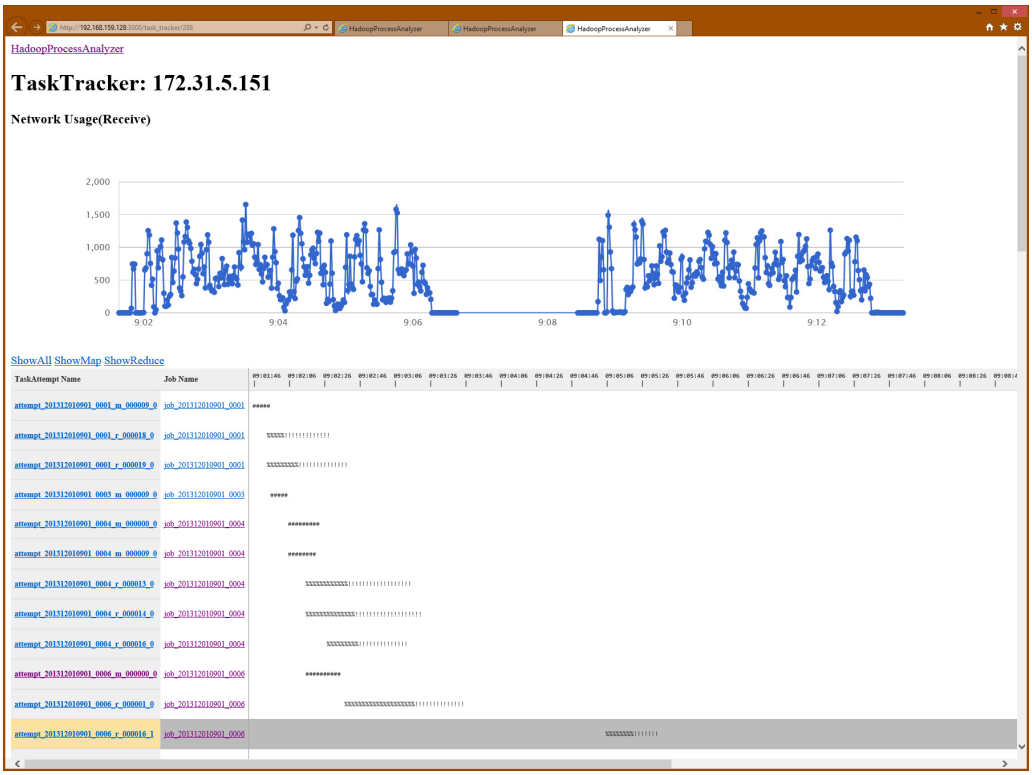


図 6, TaskTracker で実行された Task 一覧

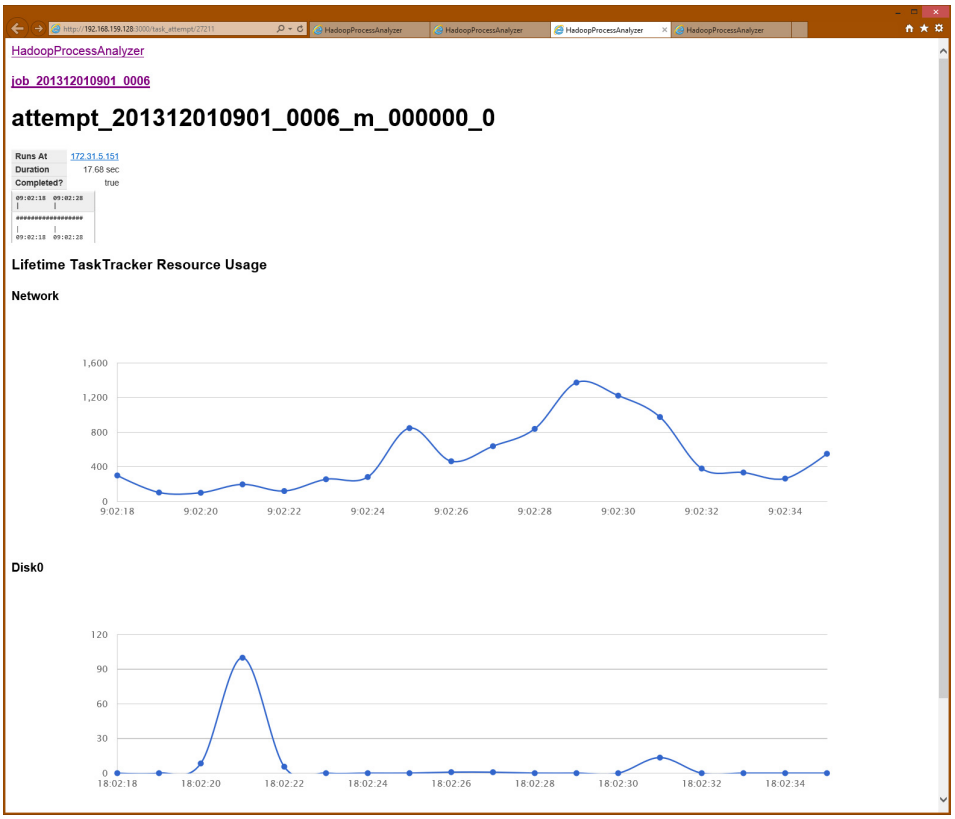


図 7, Job のタスク一覧

第4章 実験環境

本論文の以降の章には、実際の Hadoop クラスタを用いた実験結果が含まれる。そこで、本章では本研究実施に際して用いた実験環境の解説を行う。

実験は、研究室内に構築したクラスタ（Lab cluster）と Amazon EC2 上に構築したクラスタ（EC2 cluster）の 2 つのクラスタ上で Job を実行し、HadoopProcessAnalyzer を用いてその結果を分析することによって行った。

実験に用いた Hadoop のバージョンは 1.1.2 である。この Hadoop に、時間取得のコードを数行追加し、Reduce task がそれぞれの Map 中間データ取得にかかった時間をログに出力する機能を追加した。また、Hadoop の実行に伴って、3 章で述べた HadoopProcessAnalyzer の TaskTracker リソース監視スクリプトを各 TaskTracker 上で実行した。これらの監視機能がもたらすオーバーヘッドは非常に小さく、Hadoop の Job 処理時間には全く影響しない。

4.1. Lab cluster

研究室内クラスタ（Lab cluster）は、1 つのギガビットイーサネットスイッチと 7 台のノードによって構成される、スター型トポロジのクラスタである。ノードは HDD 数や CPU 数、RAM 容量において大きくばらつきがあり、Task slot 数は CPU 性能に応じて差異を持たせている。表 1 にその詳細を示す。

4.2. EC2 cluster

Amazon EC2 クラスタ（EC2 cluster）は、IAAS 型クラウドサービスである Amazon EC2 上に構築したクラスタで、51 台のノードから構成されている。第 5 章で行う予備実験では、20 台のノードからなるクラスタを用いた。EC2 cluster の性能詳細を表 2 に示す。各ノードには、EC2 のインスタンスプランにおける“m1.xlarge” [12] を用いた。EC2 のインスタンスプラン決定の根拠は、CPU がクアッドコアで、計算に用いることの出来る HDD が 4 つ付属している、そして RAM が十分な量確保されている事が、[1] で解説されている一般的な Hadoop のノード性能に近しいからである。また、各ノードのネットワーク I/O における入出力帯域がほぼ 1Gbps 確保されていると [8] で報告されており、専用の Hadoop クラスタを構築した際と比較して遜色ないからである。ただし、iperf を用いてノード間ネットワーク帯域を実際に計測したところ、各ノード間の実効ネットワーク帯域は 1~1.5Gbps であり、ギガビットイーサネットで相互接続された一般的な Hadoop クラスタよりも若干帯域幅が広かった。さらに、文献[6]で述べられているように、Amazon EC2 は全てのノードが 1 つのラックに所属しているような、スター型のトポロジとみなすことのできるネットワーク帯域を持っている。MapReduce クラスタは、ラック間スイッチで接続された複数のラックに設置されたノード群によって構成される事が多いといわれており、そのラック間スイッチでのネットワーク輻輳が多く観測されている [10] 。このようなネットワーク輻輳は EC2 上で構築された Hadoop クラスタでは発生しないことに留意すべきである。

表 1, Lab cluster 性能詳細

Name	Role	CPU	Core	RAM	HDD	Task slot
N0	JobTracker NameNode	Phenom II X4 920	4	4GB	1	N/A
N1	TaskTracker DataNode	Core i7 960	4	6GB	4	Map: 4 Reduce: 4
N2		Xeon E3 1240	4	4GB	3	
N3						
N4						
N5		Celeron G550	2	2GB	2	Map: 2 Reduce: 2
N6						

表 2, EC2 cluster 性能詳細

Name	Role	CPU	Core	RAM	HDD	Task slot
N0	JobTracker NameNode	Xeon E5507 (相当)	4	15GB	5	N/A
N1~ N51	TaskTracker DataNode					Map: 4 Reduce: 4

4.3. Hadoop の環境設定

HDFS のファイルブロックサイズの初期値は 64MB であるが、本研究ではその値を 256MB に拡張した。これは、大規模データを扱う Hadoop クラスタにおける一般的な設定である。[1] また、Lab cluster における HDFS のブロック複製数を、初期値の 3 から 1 に変更した。その理由は、Lab cluster は一般的な Hadoop クラスタと比較して非常に規模が小さく、初期値の複製数では HDFS のファイルブロック複製がもたらすネットワーク負荷が相対的に大きくなってしまうからである。さらに、Data Local Map task の発生確率が通常よりも高くなってしまうことも懸念したからである。

Hadoop の Task scheduler にはユーザ数を 1 人に設定した Fair scheduler を用いた。デフォルトの FIFO scheduler では、複数の Job を並列に動作させることができないからである。また、提案する複数の scheduling 方式も、Fair scheduler を改良して実装したものであり、Job の並列実行をサポートする。

4.4. 実験用 Job

実験には、ある程度の Map 中間データを生成し、データ転送が Hadoop の処理時間に影響する一般的な Job である[13] Sort Job を選択した。Sort Job は Hadoop クラスタのスループットを評価するベンチマークとしてしばしば用いられている。[1][14]

入力データは、Hadoop に付属するランダムファイル生成 Job である “randomwriter” を用いて生成したものを使用し、複数の Job を 5 秒間隔でクラスタに投入した。投入された Job は並列実行される。このとき、各 Job の入力データはそれぞれ別のランダムデータを用いた。同じデータに対して複数の Job を実行する場合、Map task 時に、特定のノードにデータ I/O 負荷がかかってしまうからである。

第5章 予備実験：Job 実行速度低下メカニズムの分析

まず、上述した実験条件にて Job を実行し、Job 実行時間に影響する Hadoop のボトルネックを調査した。各クラスタに投入した Job の詳細は表 3 に示す。Lab cluster には 5 個、EC2 cluster では 16 個の Job を 5 秒間隔で投入し、各 Job の進行過程を調査した。

表 3 予備実験 Job の設定

Target Cluster	Job 種別	入力データ	Map task 数	Reduce task 数	同時実行 Job 数
Lab cluster	Sort	Random Files 5GB	20	10	5
EC2 cluster	Sort	Random Files 5GB	20	20	16

5.1. Lab cluster における性能低下

図 8 に各 Job の実行時間を示す。Job 名末尾の番号はクラスタへの投入順を表している。図 9 と図 10 はそれぞれ Lab cluster の全 Job に所属する Map と Reduce task の処理時間の累積分布のグラフである。図 9, 10 を見ると、Lab cluster における Map task の約半数が 20 秒以内に終了し、約 7 割が 40 秒以内に終了している一方、180 秒以上かかっているものもごく少数存在している。一方で、Reduce task の処理時間はばらつきが大きい。

図 8 を見ると、Job0 と Job1, 2 の実行時間には大きな差がある。ここで、Job 実行速度が低下する仕組みを分析するために、これらの Job における Task 進行状況を示したガントチャート（図 11, 12, 13）を用いる。このチャートにおける“#”は Map task を、“%”は Reduce task の Map 中間データ受信フェイズ（Copy フェイズ）を、“!”は Reduce task の Reduce 処理フェイズ（Reduce フェイズ）、そして“-”は投機的実行などが原因で、終了前に停止した Task を表している。

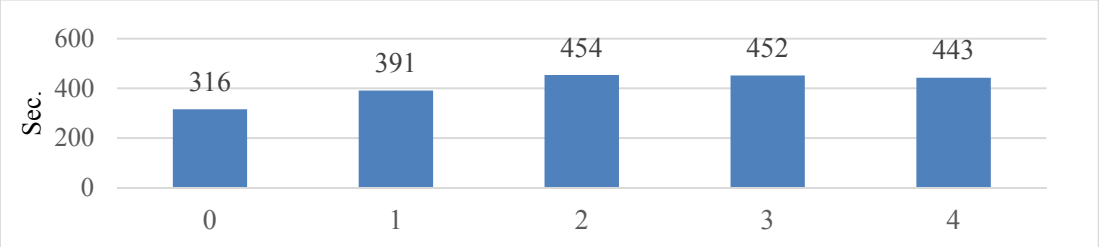


図 8, Job 実行時間（Lab cluster）

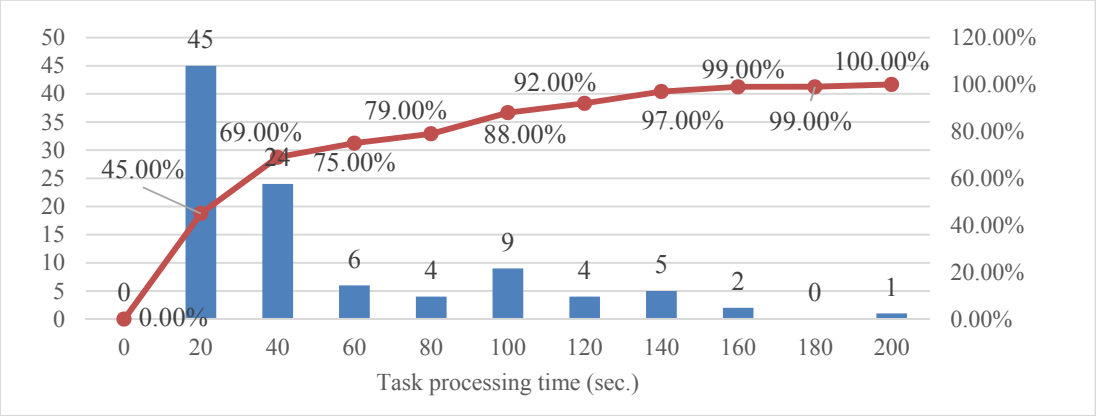


図 9, Map task 実行時間の累積分布 (Lab cluster)

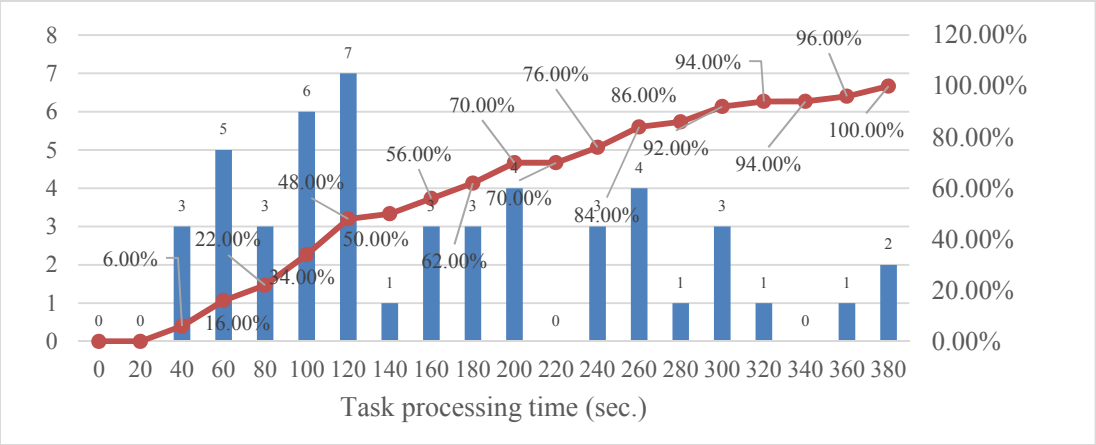


図 10, Reduce task 実行時間の累積分布 (Lab cluster)

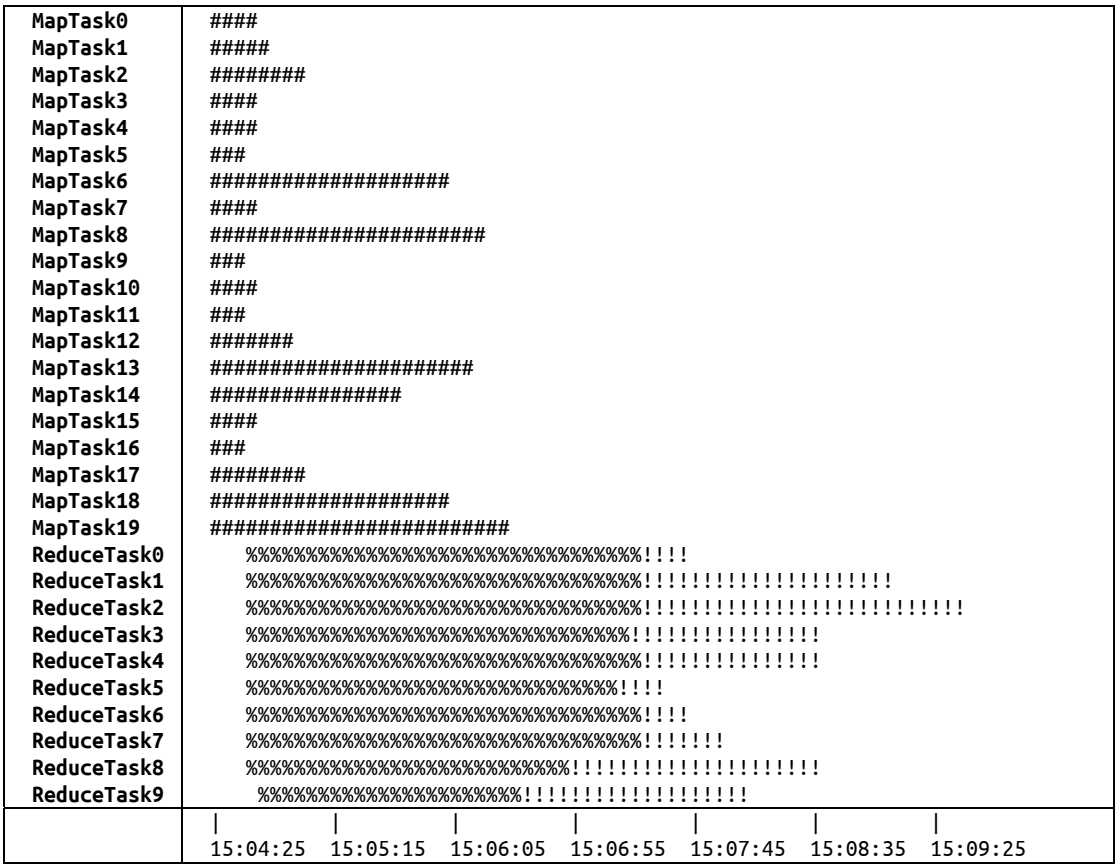


図 11, Job0 の Task 進行ガントチャート (Lab cluster)

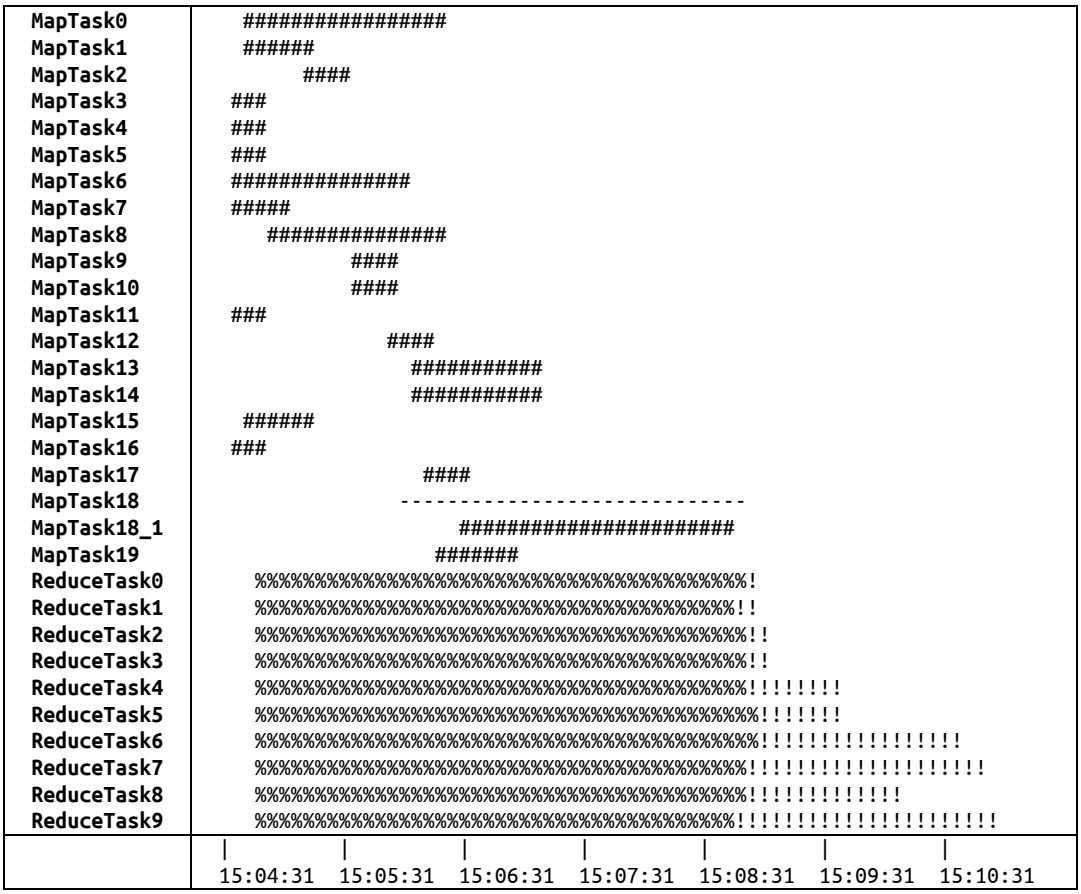


図 12, Job1 の Task 進行ガントチャート (Lab cluster)

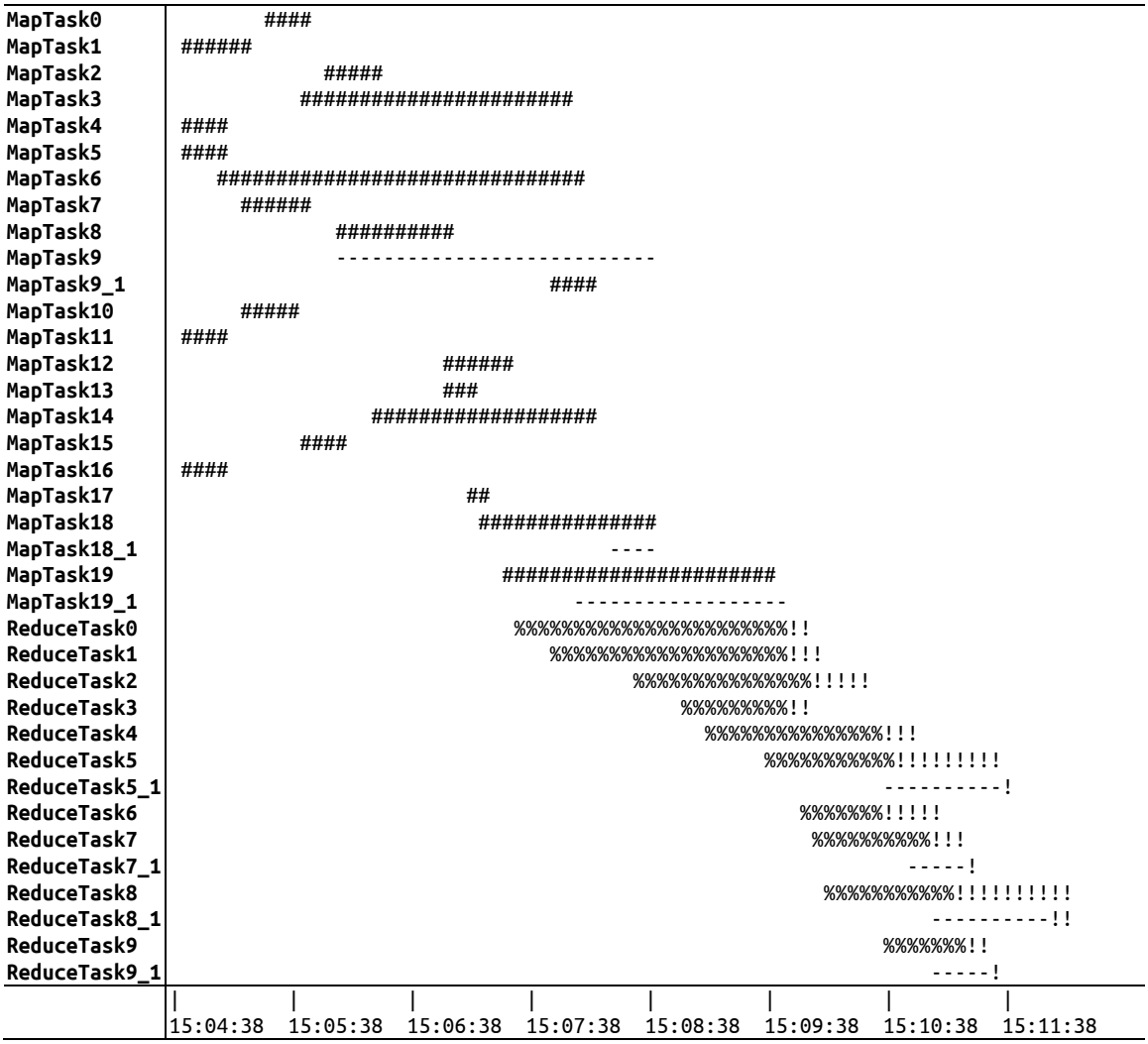


図 13, Job2 の Task 進行ガントチャート (Lab cluster)

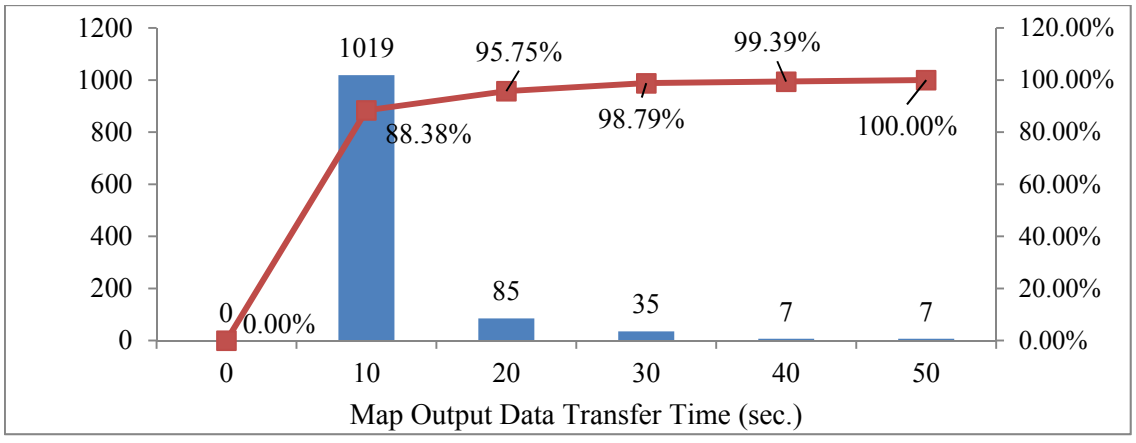


図 14, Map 中間出力データ転送時間の累積分布 (Lab cluster)

5.1.1. Map task の速度低下

Job0（図 11）では、MapTask6, 8, 13, 14, 18 そして 19 が他の Map task と比較して 2~6 倍もの処理時間を要していることがわかる。Job0 の全ての Map task は Data Local であったため、ネットワーク I/O の影響は受けていない。さらに、Sort Job の Map task は入力データを読み込むだけである為、CPU リソースは消費しない。よって、これらの Map task の遅延は HDD I/O の輻輳に起因するものと考えられる。これは、実際に TaskTracker のリソース使用率を解析することにより裏付けられた。

5.1.2. Reduce task の速度低下

Reduce task の Copy フェイズの遅延は、大きく分けて 2 つの原因によって引き起こされている。1 つめは、Map 中間データを所持している送信元 TaskTracker の HDD I/O 輻輳である。図 14 は、Reduce task が 1 つの Map 中間データを受信するのに要した時間の累積分布である。Copy フェイズはネットワーク負荷を発生させるが、ネットワーク輻輳が発生する前に、送信元 TaskTracker がデータ読み出し時に引き起こす HDD I/O がボトルネックとなっていた。この事は、図 14 で表されているような、中間データ転送に非常に長時間を要した TaskTracker ペアのデータ I/O リソース使用率を解析することで判明した。図 11 の ReduceTask0-7 のように、全ての Map task が終了した後しばらく Reduce フェイズに移行出来ていない Reduce task にこの現象が見られた。

2 つめの原因は、Map の遅延に起因する Reduce フェイズへ移行の遅れである。本論文の 2 章 2 節でも触れているように、Reduce task は全ての Map task が終了するまで Reduce フェイズに移行することが出来ない。たった 1 つでも Map task に遅れが生じていた場合でも、その Map task が終了するまで何もせずに待つことになる。（終了済 Map task の中間データの受信は可能） 図 12 にて、MapTask18_1（MapTask18 の投機的実行タスク）の終了を待っている Reduce task 群にこの現象が見られる。

また、Reduce task 群が遅延している Map task の終了を待っている間は、当然 TaskTracker の Reduce task slot を何もしないまま消費し、クラスタ全体の Reduce task slot 不足を引き起こしてしまい、TaskTracker の CPU や I/O リソースを活用できなくなってしまう状態となる。これが、文献[8]にて指摘されている“Slot hoarding”問題であり、図 13 の Reduce task 群の割り当てが遅延し、Reduce task が一斉に開始できていない原因である。

なお、Job0 の ReduceTask1,2 等で見られる Reduce フェイズの遅延も Sort 処理時に発生する HDD I/O がボトルネックとなっていた。また、Job 実行中、各 TaskTracker の CPU コアが全て 100%に固定されることはなく、常に一部のコアに余裕が見られていたことから、CPU 処理速度に起因する Task の速度低下はないと言える。

5.2. EC2 cluster における性能低下

同様の分析を EC2 クラスタにおいても行った。EC2 クラスタで実行した各 Job の実行時間は図 15 の通りである。また、全 Job の Map と Reduce task 処理時間の累積分布はそれぞれ図 16 と図 17 に示す。そして図 18 が中間出力データ転送時間の累積分布である。これらの図から、Lab クラスタと比較して、特に Reduce task 処理時間の分布に偏りが少な

く、極端に速度が低下している Task の数が少ない事が分かる。

図 15 を見ると Job0 から 3 迄と、Job4 以降の Job 実行時間には 50%以上の大きな差が見られるため、その原因を調査する。図 19, 20 は、それぞれ Job0 と Job4 の Task 進行状況ガントチャートである。図 20 を見ると分かるように、Job4 は MapTask8 の遅延によって、Reduce task 群の進行が遅延している。また、それにより引き起こされた Reduce task slot 不足によって半数以上の Reduce task の開始時間も著しく遅延している。この MapTask8 の処理速度が低下している理由は、Lab cluster で観測された現象と同じく HDD I/O に起因するものであった。MapTask8 が遅延した結果、投機的実行タスクである MapTask8_1 が生成され、先に Task を完了させた。

EC2 環境においては、比較的遅い Reduce task（全体の下位 10%）の約 8 割が、HDD I/O ボトルネックに起因するものであった。そして、残りの 2 割は中間データ転送に因るネットワーク輻輳が原因であった。

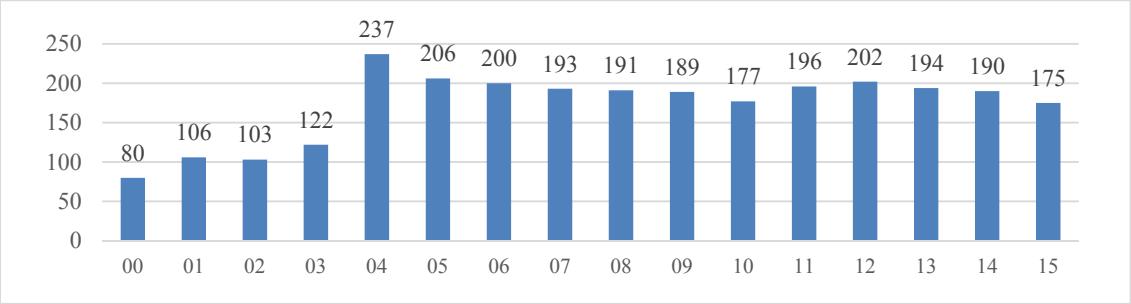


図 15, Job 実行時間（EC2 cluster）

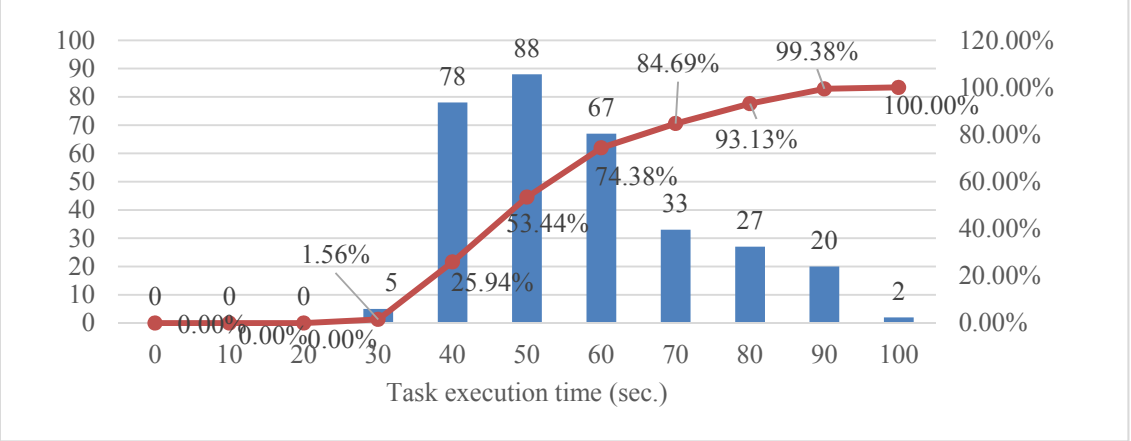


図 16, Map task 実行時間の累積分布 (EC2 cluster)

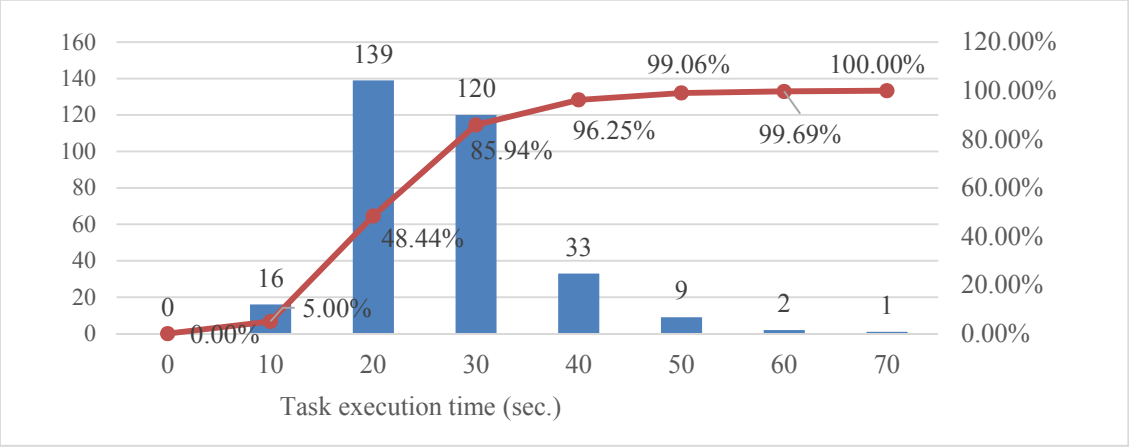


図 17, Reduce task 実行時間の累積分布 (EC2 cluster)

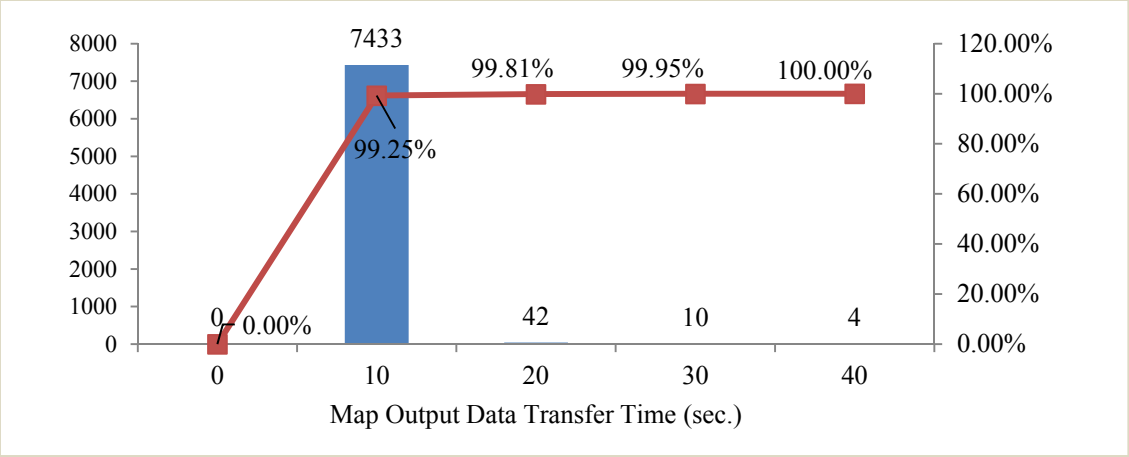


図 18 , Map 中間出力データ転送時間の累積分布 (EC2 cluster)

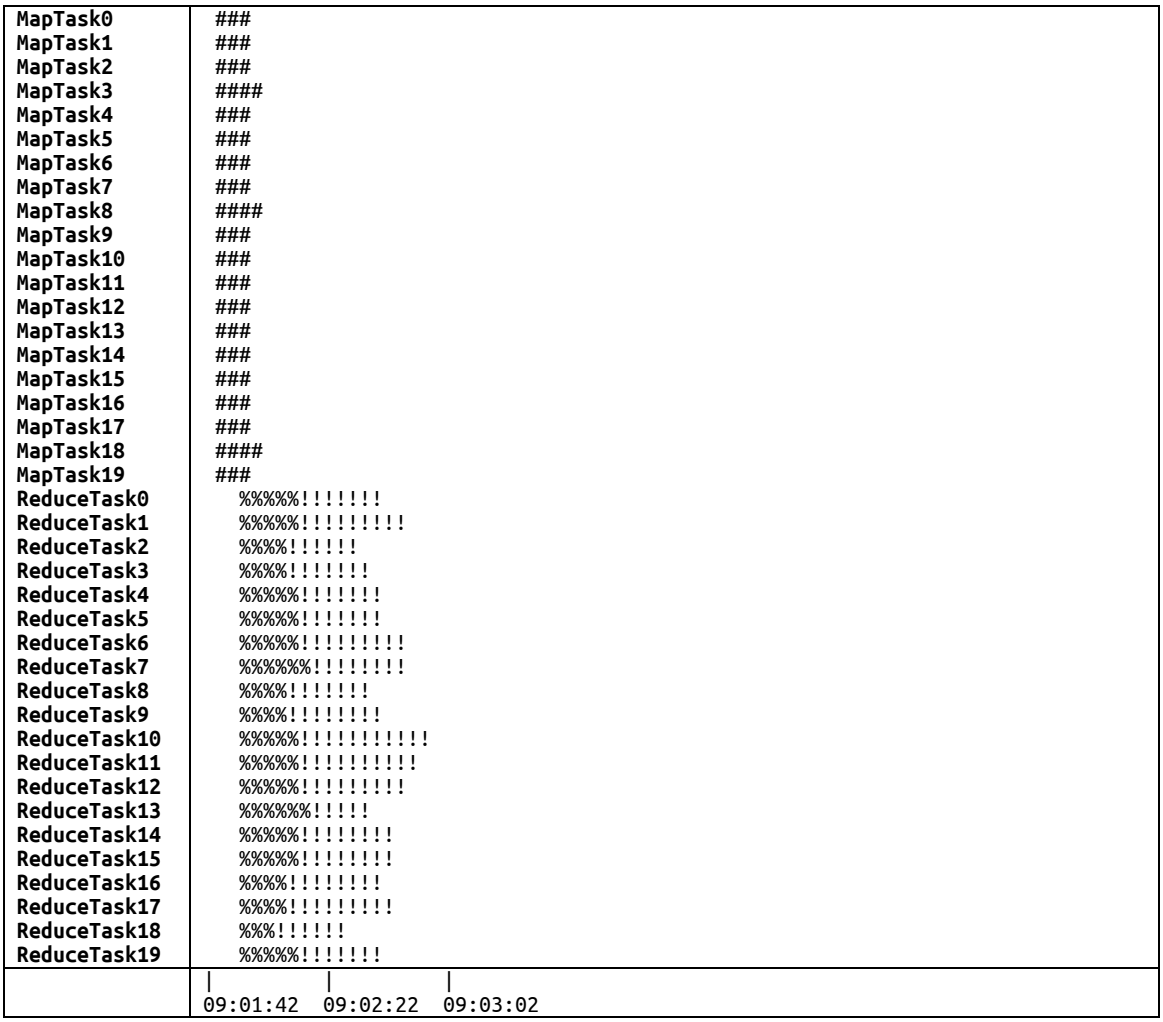


図 19, Job0 の Task 進行ガントチャート (EC2 cluster)

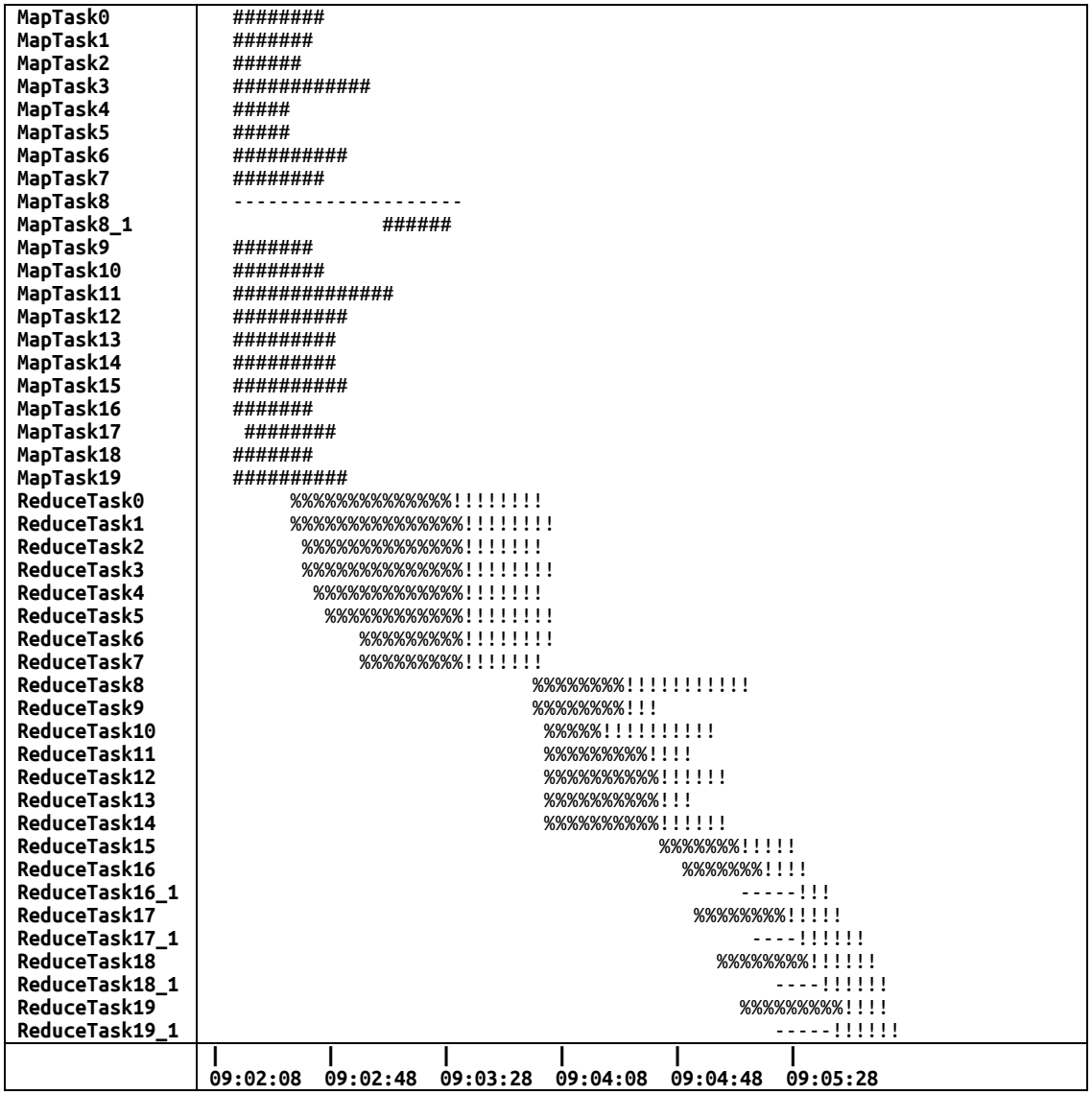


図 20, Job4 の Task 進行ガントチャート (EC2 cluster)

5.3. 考察

Lab cluster では、ネットワーク輻輳による Task 遅延は観測されず、全ての低速 Task のボトルネックが HDD I/O であった。一方、Lab cluster と比較して、EC2 cluster のノードは HDD の数が多く（5 台）、ノードの性能も一様である為、HDD I/O がボトルネックになる確率が少なかった。

Lab cluster においてネットワーク輻輳が観測されなかった理由は 3 つ考えられる。1 つは、Lab cluster の規模が小さいため、Map 中間データ受信時に発生するトラヒックの量が少なかったこと。2 つ目は、Lab cluster がその CPU 性能に比べ搭載している HDD 数が少なかったことが HDD I/O 帯域不足を引き起こしているということだ。文献[1]によると、TaskTracker の典型的な構成は、クアッドコア CPU に対して 4 台の HDD が用いられているという。それに対し我々の Lab cluster の TaskTracker の HDD 数は表 1 に書かれているように非常に少ない。そして 3 つめの理由として、TaskTracker 性能の大きなばらつきがあげられる。Task scheduling を分析すると、一部の高速なノードが複数同時にタスクを完了させるために Map 中間データを多数生成し、Reduce task へ送信する際に HDD I/O 輻輳を引き起こしてしまっていた。

一方、EC2 cluster は、均等な性能の TaskTracker や、十分な数の HDD を持っていたため、Lab cluster と比較して HDD I/O 輻輳の発生数が著しく少なかった。それにも拘わらず、ネットワーク輻輳があまり発生しなかった理由は、次の通りであると考えられる。一般的なの Hadoop クラスタが複数のラックから構成され、ラック間をまたぐ通信がボトルネックになり易い。それに対し、我々の EC2 cluster のネットワークトポロジがスター型であり、さらに上述したように状況によっては 2Gbps 弱の実行帯域をもつ為だ。

これらの結果を踏まえると、Lab cluster に対しては HDD I/O のボトルネック軽減、そして EC2 cluster に対しては HDD I/O とネットワーク I/O 双方のボトルネックを軽減することで Job 実行時間の短縮が期待できる。

第6章 提案方式

予備実験では、Lab cluster, EC2 cluster の双方の環境において、データ I/O が Hadoop の Job 実行のボトルネックとなっていることが判明した。Hadoop の従来の Task Scheduler は、TaskTracker の Task slot の利用状態に基づいて Task を割り当てる。しかし、Task の種類やその進行状況によって Task Tracker に生じる負荷が異なるため、クラスタ内で TaskTracker のリソース使用率に偏りが発生しうる。故に既存の Task scheduling 条件が最適であるとは言いがたい。

そこで、私は Hadoop の Job 実行時間の短縮を目的として、クラスタ内の TaskTracker のデータ I/O 使用率の偏りを減らす Task scheduling 方式を 3 種類提案する。“Receive Rate Scheduling”、“Disk Rate Scheduling”、そして“Potential Reception Scheduling”である。これらの Scheduling 方式の思想は共通している。従来の Task slot 数による Task scheduling 方式を維持したまま、TaskTracker の特定のリソース使用率に着目し、その使用率があらかじめ決めておいた閾値を超えた場合、その TaskTracker に新規 Task を割り当てないという条件を追加したものである。TaskTracker のリソース使用率は、HeartBeat を用いて JobTracker 上の Task Scheduler に伝えられる。

6.1. 提案方式の実装

私は TaskTracker に小さな改変を加え、リソース使用率を取得させる機能を追加した。また、HeartBeat にも改変を加え、取得したリソース使用率の通知機能をもたせた。各 Task scheduling 方式は、Hadoop に実装されている Fair Scheduler を改変することで実装し、各方式の有効化、無効化、そして閾値設定は Hadoop の設定ファイルで記述できるようにした。

6.2. Receive Rate Scheduling

TaskTracker のネットワークインターフェースの受信帯域が閾値を超えていた場合に Data Local でない Map task と Reduce task を割り当てない制御方式である。

TaskTracker は、HeartBeat を送信するタイミングで、前回の HeartBeat からの通信量を計算して、理論値である最大ネットワーク帯域：1Gbps との比を“RxRate”として、JobTracker に送信する。

そして、JobTracker は予め設定しておいた RxRate の閾値を超えている TaskTracker には Data Local でない Map task と Reduce task を割り当てない。この 2 種類のタスクは、割り当て直後に大量の受信ネットワークトラヒックを引き起こすからである。一方、Data Local Map task はネットワークを使用しないため、制御を行わない。

6.3. Potential Reception Scheduling

Potential Reception Scheduling は、TaskTracker に所属している Reduce task 群が、Map 中間データ受信に伴って発生させるネットワーク負荷を見越して Task scheduling を行う。TaskTracker に割り当てられた Reduce task 群が現在受信している Map 中間データの合計データサイズが閾値を超えた場合、その TaskTracker に Data Local でない Map

task と Reduce task の新規割り当てを行わない制御方式である。TaskTracker は、所属している Reduce task が現在受信している Map 中間データの合計データ量を“PotentialReceptionAmount”として HeartBeat を用いて JobTracker に通知する。JobTracker は、TaskTracker の PotentialReceptionAmount が閾値を超えていた場合、Data Local でない Map task と Reduce task を割り当てない。

Receive Rate Scheduling が現在の瞬間的なデータ受信帯域のみ考量しているのに対し、この手法は今後数秒間にわたって発生しうるネットワーク負荷を見越した予測制御方式である。

6.4. Disk Rate Scheduling

上述した 2 つの Task scheduling 方式はネットワーク I/O への負荷集中を避ける事を目的としている事に対し、Disk Rate Scheduling は、HDD I/O の負荷の分散を目指している。TaskTracker は、HeartBeat 送信時点の CPU の HDD I/O 待ち時間を元に HDD I/O 使用率を計算し、JobTracker に送信する。なお、TaskTracker に HDD が複数搭載されていた場合はそれら全ての使用率の平均値を用いる。JobTracker は、予め設定した閾値と HDD 使用率を比較し、超えていた場合には Map task を割り当てない。Map task は Map Input Data Split の受信を HDD で行う為、割り当て直後に HDD I/O 負荷を発生させる。一方、Reduce task の Map 中間データ受信は、一度メモリ上に受信した後にマージ処理され、HDD に記録される。そのため、Reduce task は割り当てられてから HDD I/O 負荷を発生させるまで数秒以上の遅延がある。本方式では HeartBeat 送信時の TaskTracker の HDD I/O 使用率を用いるため、Reduce task が実際に HDD I/O 負荷を発生させるタイミングの情報ではない。そこで、過剰制御を回避するため、Reduce task は TaskTracker の HDD I/O 使用率に因らず割り当てる。

第7章 提案方式の評価

Lab cluster と EC2 cluster でそれぞれ 5GB と 20GB のランダムファイルを用いた。また、Lab cluster では 5 個、EC2 cluster では 14 個の Job を 5 秒間隔で複数投入して並列実行した。各 Job の詳細は表 4 に示す。

表 4，提案方式評価 Job の設定

Target Cluster	Job 種別	入力データ	Map task 数	Reduce task 数	同時実行 Job 数
Lab. cluster	Sort	Random Files 5GB	20	10	5
EC2 cluster	Sort	Random Files 20GB	40	38	14

この Job 群の実行を 1 回として、これを 7 回繰り返し行い、Job 実行時間の平均値を取る事で統計的ばらつきの影響を抑えた。提案方式の閾値は以下の通りである。

表 5，各クラスタにおける提案方式の閾値

	Receive Rate Scheduling	Potential Reception Scheduling	Disk Rate Scheduling
Lab cluster	160Mbps	100Mbytes	30%
EC2 cluster	300Mbps	150Mbytes	40%

これは提案方式の効果が最大となるような経験的に得た値であり、クラスタの性能や規模、トポロジ、そして投入する Job によって変化すると考えられる。

これらの条件で、Lab クラスタと EC2 クラスタの双方で各 Scheduling 方式の評価を行った。評価対象は、Hadoop デフォルトの Fair Scheduler を用いた場合、提案方式をそれぞれ単体で適応した場合、そして、3 つの提案方式を全て実装した場合の 5 種である。但し、EC2 cluster では、Receive Rate Scheduler と Disk Rate Scheduler の 2 つを合わせて実装した Scheduling 手法も評価した。その結果については第 8 章 1 節にて触れる。

表 6, Lab cluster における各 Scheduling 方式の Job 実行時間の平均 (秒)
(Job 数 35)

	Default	Receive Rate	Potential Reception	Disk Rate	全提案方式実装
Avg. Job time	465sec	461sec	464sec	443sec	456sec
Δ	-	-4sec	-1sec	-22sec	-9sec

表 7, EC2 cluster における各 Scheduling 方式の Job 実行時間の平均（秒）
(Job 数 98)

	Default	Receive Rate	Potential Reception	Disk Rate	Receive Rate + Disk Rate	全提案方式実装
Avg. Job time	538sec	534sec	524sec	507sec	506sec	511sec
Δ	-	-14sec	-4sec	-31sec	-32sec	-27sec

表 8, EC2 cluster における各 Scheduling 方式の Task 処理時間の平均 (秒)
(Map task 数: 7840, Reduce task 数: 1940)

	Default		Receive Rate		Potential Reception		Disk Rate		Receive Rate + Disk Rate		全提案方式実装	
	Map	Reduce	Map	Reduce	Map	Reduce	Map	Reduce	Map	Reduce	Map	Reduce
Avg. Task time	33.27	318.49	32.06	301.99	32.55	316.26	29.54	297.84	30.7	299.61	30.13	294.26
Δ	-	-	-1.21	-16.50	-0.72	-2.23	-3.73	-20.65	-2.57	-18.88	-3.14	-24.22

表 6, 7 に、2 クラスタ上でこれら 5 種の Task scheduling 方式を用いた際の Job 実行時間の平均と、デフォルトの Fair Scheduler との差を記す。なお、各 Scheduler に対して Lab cluster に投入された Job 数は 35 Job、EC2 cluster では 98 Job である。実験結果から、Lab cluster と EC2 cluster の双方で、Disk Rate Scheduling 方式が平均してそれぞれ約 5%, 約 6% Job 実行時間を短縮した事が分かる。

第 5 章で考察した通り、ネットワーク I/O に着目した 2 種類の Scheduling 方式は、Lab cluster よりも EC2 cluster において効果を発揮した。Receive Rate Scheduling は EC2 cluster において 2.6% Job 実行時間を短縮した一方、Lab cluster では短縮率は 1%未満にとどまった。また、Potential Reception Scheduling の効果は限定的で、統計的ばらつきの範囲内であると言える。また、全提案方式を実装した際は、両方のクラスタにおいて Disk Rate Scheduling 単体を実装した場合と比較して性能が若干低下した。

表 8 に、EC2 cluster における Scheduling 方式毎の Map task と Reduce task の平均処理時間を記した。Task 単位で見ると、Receive Rate Scheduling 方式では Map task 処理時間を約 3.5%、Reduce task 処理時間を約 5%短縮している。Map task の制御効果が低かった理由は、Data Local でない Map task が全体の 2 割程度であったためである。

Potential Reception Scheduling 方式は Reduce task の割り当て制御である為、Map task に対する性能向上はほぼない。また、期待されていた Reduce task に対する制御効果も低かった。

Disk Rate Scheduling は、予想通り Map task の平均処理時間を約 11%と大幅に向上させている。また、Reduce task の平均 6.5%短縮させている。これは、Map task による Disk I/O 輻輳を防ぐことにより、Disk I/O 輻輳に起因する Reduce task の遅延を防いだことにある。最後に、提案手法を組み合わせる事によって、それぞれの制御効果が重複した結果となった。全提案方式を実装した場合、Map task 平均実行時間を約 9.4%、Reduce task 平均実行時間を約 7.6% それぞれ短縮させることが出来た。

総合すると、Disk Rate Scheduling は Map task と Reduce task 両方の処理時間の大幅な短縮を達成し、Job 実行時間にその結果が反映されている。また、Receive Rate Scheduling は、Reduce task 処理時間短縮効果を十分発揮し、Job 実行時間を短縮させた。これらに対し、Potential Reception Scheduling 方式の効果は非常に小さなものだった。

第8章 考察と今後の課題

8.1. 全提案方式実装時の性能低下原因

表 6, 7 を参照すると分かるように、3 つの提案方式を全て実装した場合に、Disk Rate Scheduling 単体を実装した場合と比較して性能が低下した。これは Receive Rate Scheduling と Potential Reception Scheduling の 2 種類の Reduce task 割り当て規制アルゴリズムを動作させたことで、Reduce task 割当が過剰に抑えられてしまったからだと考えられる。表 8 を見ると、平均 Reduce 処理時間自体は短縮できていることからも類推できる。

そこで、同様の条件下において Receive Rate Scheduling と Disk Rate Scheduling のみを実装した場合、平均 Job 実行時間は 506 秒 に短縮された。Receive Rate Scheduling と Potential Reception Scheduling の両方を実装する場合は、それぞれの閾値を緩和する事で過剰制御を防ぐことが可能だと思われる。

8.2. Potential Reception Scheduling の低性能原因

7 章の結果から、3 つの提案方式のうち、Potential Reception Scheduling の効果は非常に小さなものであった。本方式は、現在の TaskTracker の瞬間的な受信ネットワーク I/O 負荷ではなく、これから Reduce task の Map 中間データ受信によって発生しうる受信ネットワーク I/O 負荷を予測して制御を実施する方式であるが、Hadoop において、TaskTrackerの受信ネットワークI/O負荷は、全てReduce task に起因するものでは無い。HDFS の複製トラヒックや、Map task のデータ入力時等が占める割合が少なくなかったのである。これにより、Reduce task の Map 中間データ受信量のみを監視する Potential Reception Scheduling の制御効果が出なかったと考えられる。

しかし、本方式のように、負荷発生を予測した制御は有効であると考えている。現在の負荷状況のみを考慮した制御方式では、本質的に負荷集中状態の発生を防ぐことが出来ないからである。故に、制御条件となる対象の拡張や、負荷発生予測手法の検討などを通じ、方式の改良を続ける予定である。

8.3. 提案方式の閾値に因る影響

提案方式はいずれも予め閾値を設定しておく必要がある。そこで、閾値の値が Job 実行時間に及ぼす影響を調査するため、Lab cluster 上で実験を行った。実験環境と Job 内容、実行回数は第 7 章の提案方式評価時と等しくした。表 9, 10, 11 は、それぞれの提案方式において、閾値を変動させた際の Job 実行時間の平均値である。

表 9, Receive Rate Scheduling の閾値による影響（Job 実行時間 秒）

80Mbps	320 Mbps	480 Mbps	640 Mbps
455sec	475sec	469sec	436sec

表 10, Potential Reception Scheduling の閾値による影響（Job 実行時間 秒）

100Mbyte	150Mbyte	200Mbyte	250Mbyte
454sec	465sec	486sec	476sec

表 11, Disk Rate Scheduling の閾値による影響（Job 実行時間 秒）

20%	40%	60%	80%
439sec	443sec	451sec	477sec

いずれの方式においても、最大 5~10%程度の差異が認められ、提案方式の性能は閾値によって大きく左右されることが判明した。したがって、クラスタの環境や、Job 内容に応じて適切な閾値を推定する方法が今後必要である。

8.4. ネットワーク I/O ボトルネックの影響

本論文では HDD I/O とネットワーク I/O の双方に着目した Task scheduling 方式を提案したが、後者に着目した提案方式の効果は限定的なものであった。

Hadoop の Job 実行においては、ネットワーク転送時間が Job に大きく影響し、ボトルネックになり得ることは非常に多くの文献で指摘されている。それにも拘わらず、本研究の予備実験において観測されたネットワーク輻輳件数・時間は非常に少なかった。また、図 11, 12, 13, 19, 20 のガントチャートからも、Job 実行過程におけるデータのネットワーク転送時間の占める割合も非常に少ない事が読み取れる。

この理由の一つに、今回実験環境として用意した Lab cluster と EC2 cluster ではネットワークボトルネックが発生しづらい環境であったことが考えられる。第 5 章 3 節で述べたように、Lab cluster は TaskTracker の HDD I/O 帯域が非常に少なく、ネットワーク輻輳が起きる前に HDD I/O 輻輳が発生してしまっていた。また、EC2 cluster では多数のノードが 1Gbps 以上の帯域を持つネットワークでスター型に接続されているため、複数ラックで構成された一般的な Hadoop クラスタと比較すると、ボトルネックになりやすいラック間リンクが存在していない。これらの環境ではネットワーク輻輳は発生しづらく、それ故ネットワーク I/O に着目した Task scheduling 方式の成果が芳しくなかったと考えられる。

8.5. TaskTracker の HDD I/O 使用率計算時の平均化問題

Disk Rate Scheduling は、TaskTracker に複数台の HDD が搭載されていた場合、全ての HDD I/O 使用率を平均化して扱う。この方式は一定の効果は上げたものの、最適解とは言えない。HDD I/O の輻輳は各 TaskTracker（DataNode）の HDD 単位で発生するため、平均 HDD I/O 使用率が低いからといって、必ずしも HDD I/O 輻輳が発生していないとは限らない。しかしながら、各 TaskTracker に搭載されている全ての HDD の状態を把握し、新たな Task がどの HDD の I/O に負荷をかけるか、等を考慮した Scheduling を行う事は、オーバーヘッドやスケーラビリティの観点から現実的では無い。故に、最適ではないが、平均 HDD I/O 使用率を使用する事で妥協する必要があると思われる。

第9章 結論

本研究は、並列に複数の Job が実行される、一般的な Hadoop クラスタの性能向上を目指して行ったものである。本論文ではまず、Hadoop クラスタの各ノードのリソース使用率と Job と Task の進行過程を分析するツールの構造を紹介した。それを用いて、研究室内に構築した小規模な Hadoop クラスタと、Amazon EC2 上で構築した中規模な Hadoop クラスタ上において、データ I/O（HDD I/O とネットワーク I/O）が Job 実行時間に影響し、それによる Task の遅延が Job 全体だけでなく、他の Job にまで大きく影響することを示した。

その結果を受け、TaskTracker 間の HDD I/O 使用率とネットワーク I/O 使用率のばらつきを抑えることにより、Job 実行時間の短縮を目的とする Task scheduling 方式を 3 種類提案した。ネットワーク I/O に着目し、Reduce task の割り当て規制を行う Receive Rate Scheduling と Potential Reception Scheduling、そして HDD I/O に着目し、Map task の割り当て規制を行う Disk Rate Scheduling である。

提案方式の評価は上述した 2 つのクラスタ上で行った。HDD I/O に着目した Task scheduling 方式は、双方の環境において平均して 5%以上の Job 実行時間短縮、10%以上の Map task 処理時間の向上を達成した。また、ネットワーク I/O に着目した 2 つの Task scheduling 方式では、Job 実行時間を平均約 2.6%短縮した。そして、ネットワーク I/O に着目した制御方式が比較的効果を挙げることが出来なかった原因として、ネットワーク I/O がボトルネックとなり、Job 実行時間に影響するような環境を構築できていなかったことを挙げた。

結論として、提案した 3 種類の Task scheduling 方式により、Hadoop クラスタのスループットを向上させる事が出来た為、TaskTracker のリソース使用率をフィードバックして、Task 割り当てを制限する制御方式が有効であることが示された。従って、今後、リソースの着眼点を変更し、同様なポリシーを採用した Hadoop 性能向上方式の検討が可能となった。

謝　辞

本研究を行うに際し、筑波大学大学院図書館情報メディア研究科の川原崎雅敏教授には、研究目標の設定から研究手順、そして本稿執筆に至るまで、手厚い御指導を戴いた。ここに深謝の意を表する。また、技術面、食料面、そして精神面で研究をサポートして下さった筑波大学ユビキタスネットワーキング研究室の勝股恵璃奈君と田中大地君、工学システム学類の的場早紀君、そして卒業後であるにも拘わらず、Hadoop の挙動の解説や研究に対する助言を下さった同研究室 OB の片岡駿君と小西響児先輩に深い感謝を表する。

参考文献

[1] T. White. Hadoop: The Definitive Guide, 3rd Edition, O'Reilly Media / Yahoo Press, California, 2012.

[2] 株式会社 エヌ・ティ・ティ・データ, 平成 21 年度産学連携ソフトウェア工学実践事業 実践事業 (コウ信頼クラウド実現用ソフトウェア開発 (分散制御処理技術等に係るデータセンターの更新来夏に向けた実証事業)) 事業成果報告書, 2010 年 3 月.

[3] 株式会社 東芝, “プレスリリース: 東芝製「HEMS」のビッグデータ分析による「ライフスタイルに合わせたサービス提供」電力需給ピーク時の省エネ行動を促進する「居住者メリットの創出」分譲マンション初、暮らしのサービス提供に「ＨＥＭＳ」を活用”, 入手先 <http://www.toshiba.co.jp/about/press/2013_07/pr_j0202.htm>, 2014 年 1 月 16 日参照.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In OSDI, pages 137-150, 2004.

[5] The Apache Software Foundation, “Apache Hadoop”, <http://hadoop.apache.org>, 2014 年 1 月 16 日参照.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoika. Managing data transfers in computer clusters with orchestra. In SIGCOMM, pages 98-109, 2011.

[7] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In EuroSys, 2010.

[8] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical Report of EECS Department, University of California, Berkeley, 2009.

[9] A. Konwinski. Improving MapReduce Performance in Heterogeneous Environments. Technical Report of EECS Department, University of California, Berkeley, 2009.

[10] S.Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. “The Nature of Datacenter Traffic: Measurements & Analysis”. IMC '09 Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference. pp. 202-208. 2009.

[11] Amazon Web Services, Inc., “Amazon Web Services, Cloud Computing: Compute, Storage, Database”, <http://aws.amazon.com>, 2014 年 1 月 16 日参照.

[12] Amazon Web Services, Inc., “Amazon EC2 Instances”, <http://aws.amazon.com/ec2/instance-types/>, 2014 年 1 月 16 日参照.

[13] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In MASCOTS, pages 390-399, 2011.

[14] O'Malley, Owen. "Terabyte sort on apache hadoop." Yahoo,
入手先 <http://sortbenchmark. org/Yahoo-Hadoop.pdf>, May 2008,
2014 年 1 月 16 日参照