AspectJ を用いた Hadoop の監視とプロファイリング手法の提案

清水裕亮[†,☆] 櫻井孝平 山根智

近年，大規模分散フレームワークを利用してシステムを構築するクラウドコンピューティングが注目されているが，その内部動作は複雑であり，開発を行う際には，実際のシステム動作時の振る舞いの監視を行う手段が重要となる．また，多くの分散システムは，複数のコンポーネントを組み合わせて構築されるが，開発者がそれらすべての構成要素について，詳細を把握することは多くの時間と労力を必要とする．我々は，実行時のシステム動作を監視する軽量なモニタと，モニタが生成する実行命令列を用いたプロファイル手法を提案する．モニタは実行命令をロギングすることで実行命令の列であるトレースを生成し，このトレース内の実行命令の発生頻度によるプロファイル手法を提案する．既存の監視ツールである，Ganglia や Nagios はシステム運用の段階の監視を目的としているのに対し，本手法は分散システムの開発段階において有効な情報を提供することを目的とし，またそのシステムについての詳しい知識のないユーザなどが，システムの内部動作を理解することも支援する．実際に大規模分散処理フレームワーク Apache Hadoop をテスト対象として実験を行い，AspectJ を用いたモニタの実装によって実行トレースを取得し，プロファイルを行った．

Method for Monitoring and Profiling of Hadoop using AspectJ

Yusuke Shimizu,[†,☆] Kohei Sakurai and Satoshi Yamane

In recent years, systems based on large-scale distributed framework for cloud-computing are attracting much attention. However, the behavior of those systems is complex, and the method of monitoring the distributed system behavior is necessary. And, most of distributed systems are constructed from some components, it need a lot of time and effort for developers to understand about those components specification in correctly. This paper proposes a profiling method which use sequence of execution instructions generated by monitor. Monitor observes execution instruction of systems, and do logging to generate a trace which is a sequence of execution instructions. Using generated trace, we apply a method of analyzing large-scale distributed system's behavior, which relies on frequency of occurrence of execution instruction. Existing monitoring tools –Ganglia, and Nagios– are aimed to monitor status of a system which is in operation phase of the software lifecycle. While our proposing method is intended to provide valid information for development, additionally help general users to understand a system specification. We experimented in large-scale distributed framework Apache Hadoop, and obtain execution-trace by the monitor implemented in AspectJ. We show effectivity of proposing method by experiment.

1. はじめに

今日，クラウドコンピューティングやビッグデータが注目されており，それらは大規模な分散システムによって実現される．大規模な分散システムでは，多数のコモディティハードウェア間でネットワークを介したメッセージ通信を行うことで，同時並列的に複数の計算機上でプログラムを実行し処理の高速化を得る．このような分散システムの内部動作は，ネットワーク上の通信の非決定性や，複数ノードでの処理の一貫性，耐故障処理等のため複雑であり，開発を行う際には，システム実行時の振る舞いを把握する手段が重要となる．また，多くの分散システムは，複数のコンポーネントを組み合わせて構築されるが，開発者がそれらすべてのコンポーネントについて，使用の詳細を理解することは多くの時間と労力を必要とする．

分散システムは，Apache Hadoop[1)]☆☆のような分散処理フレームワークを利用することで，比較的容易に実装できる．Apache Hadoop 分散処理フレームワークは，分散システムの開発者からネットワーク上の通信における遅延や非決定性，ノードの障害を隠蔽し，フォールトトレラントなシステムを構成する．Apache Hadoop には運用の際に有効な統計情報を取得する手

† 金沢大学
Kanazawa University
☆ 現在，自然科学研究科
Presently with Natural Science & Technology

☆☆ ビッグデータを扱う為の大規模分散処理フレームワーク．
http://hadoop.apache.org/

段が提供されているが、開発者に対しては実行時のより詳しいシステム動作を手軽に取得する手段と，その情報を有効に活用する方法が必要である．

本研究では，大規模な分散システムを対象として，アスペクト指向を用いたモニタと，実行命令の発生頻度によるプロファイル手法を提案する．分散システムは複数計算機間でのメッセージ通信によって実現されるため，メッセージ通信を取得することで動作状況を把握する手法[3)4)5)] が有効であることが分かっている．アスペクト指向プログラミング (AOP[6)]) の技術によってオリジナルのプログラムへ変更を加えることなく分散システム動作時の実行命令を取得し，発生頻度に注目したプロファイルを行う．

我々は実際に，Apache Hadoop を用いた分散システムを対象として，AspectJ によるモニタを実装し，モニタの監視下で Hadoop MapReduce を動作させトレースを取得する．得られたトレースを用いて発生頻度によるプロファイル手法を適用する実験を行った．”**トレース**”[7)] とは，システム動作時にロギングされた実行命令の列である．モニタが既存のシステムに与える負荷は小さく，発生頻度によるプロファイル結果がシステムのデバッグ・監視に対して有効な情報となることを確認した．

以下，第 2 章に，基盤技術としての分散フレームワーク Hadoop と，一般的な Hadoop のデバッグ・監視手法について述べ，次いで問題を提起する．第 3 章で，提案するモニタとプロファイル手法について述べ，第 4 章で，実装と実験を行う．第 5 章で提案手法について考察し，本論文をまとめる．

2. 基盤技術・関連研究

本章では，まず 2.1 にて，分散処理フレームワーク Apache Hadoop の概要について説明し，次に Hadoop のデバッグや監視の手法について述べる．

2.1 Apache Hadoop

Hadoop とは，大規模データを扱うための分散並列処理フレームワークである☆．Hadoop MapReduce[10)](以降単に MapReduce と呼ぶ) と Hadoop Distributed File System(HDFS)[11)] から構成される．MapReduce とは，分散処理モデルであり大規模クラスタでの分散処理の実行環境である．HDFS とは大規模クラスタ上の分散ファイルシステムである．テラバイトやペタバイト級のデータを大量のコモディティマシンを用いて同時並列に処理することで処理性能がスケールアウトする．

Hadoop はマスタスレーブ型のアーキテクチャであり，マスタでは NameNode，JobTracker と呼ばれるデーモンが動作し，スレーブでは，DataNode と TaskTracker デーモンが動作する．マスタは，クライアントから渡された 1 つの大きな処理を小さな処理単位へと分割し，その処理を各スレーブに割り振ることを担う．例えば，HDFS ならば，マスタである NameNode は，HDFS 上の大規模データを “**ブロック**” へと分割し，DataNode に対してブロックの格納命令を行い，自身はその名前空間を保持する．MapReduce 処理の場合，JobTracker がジョブを複数の小さな処理単位である Map タスク，Reduce タスク等に分割し，TaskTracker へタスク割当てを行う．これらのノード間における協調動作は，RPC 通信によって実現する．

2.2 Hadoop のデバッグ・動作監視

Hadoop のデバッグを行う際には，システム動作のログを取得する必要があるが，これは Hadoop の設定ファイル内の log4j☆☆パラメータを変更することで，ログレベルに応じたログの取得が可能である．

Hadoop の動作の監視・解析を行うための情報として，HDFS，MapReduce，JVM，そして RPC に関するメトリクスを取得する手段が提供されている．**メトリクス**とは，ソフトウェアの品質や性能の評価尺度である．具体的には，まず HDFS 関連の統計情報ならば，`dfs.namenode.AddBlockOps` メトリクスによって，大規模データを分割した単位である”**ブロック**”を NameNode が HDFS に追加した回数を取得できる．`dfs.FSNameSystem.BlockTotal` メトリクスでは HDFS 上にて管理しているブロック数の合計，`dfs.datanode.blocks_written` メトリクスでは DataNode がブロック書き込みを行った回数がそれぞれ取得できる．JVM 関連の統計情報ならば，`jvm.metrics.gcCount` メトリクスを用いてガーベジコレクションを行った回数や，`jvm.metrics.logError` メトリクスにて ERROR レベルのメッセージを出力した回数が取得できる．

これらの統計情報は，単純にファイルへ出力するだけでなく，Hadoop の Web インターフェース経由でも確認することができる．モニタリングソフトウェアである *Ganglia*☆☆☆を用いることで，複数ホストにまたがるメトリクスを収集し，グラフ表示することも可能である．

☆ Google が開発したフレームワークソフトウェア MapReduce と BigTable のオープンソースクローンであり，Yahoo!の Doug Cutting 氏が開発，現在 Apache プロジェクトの一つである．

☆☆ `http://logging.apache.org/log4j`

☆☆☆ `http://ganglia.sourceforge.net`

2.3 問題提起

従来手法のようにテキストデータとしてのログを確認するデバッグ手法に対して，Hadoop ではフォールトトレラント性の為に，データブロックは複製され，いくつかのノードへ分散配置しており，Map 処理や Reduce 処理のコード移動の技術などから，障害の依存関係は複雑である[8)9)]．よって，あるひとつのノードで確認された障害の原因の特定の為には，そのノード内のログを確認するだけでは特定できない可能性がある．また，ログは各ノードに分散して生成されるため，それらのログを１つずつ確認することはノード数が大量の場合には現実的ではないという問題がある．よって，システム実行の概要を容易に監視する手段が求められる．

Hadoop の各種メトリクス情報は，Hadoop の実行状況や処理結果を統計的に確認するものであり，使用される場面とはソフトウェアのライフサイクルでの運用段階にあたる．分散システムを開発する際には，これらの情報では不十分と考えられる．実際，Hadoop を利用しているサービスを行っている Facebook では，システムの監視に Ganglia と合わせて，*ODS* と呼ばれる Facebook 独自の監視ツールを使用している．Ganglia は高速に複数ノードにまたがる統計情報を収集しグラフ化するが，高速な分，情報粒度[19)] は粗いため，それを補うようにより粒度の細かいアプリケーション動作に注目した監視を行うソフトウェアが ODS である[12)]．このことからも，開発者を支援するより詳細なシステム監視の手段が求められていると言える．

以上より，本研究では，大規模な分散システムの低コストなシステム振る舞いの情報取得の手段と，その情報を用いた開発者の立場で有効な，システム監視の手法を提案する．

3. 提案手法

3.1 にて，大規模分散システムの動作を取得する機能について述べる．3.2 にて，大規模な分散システムのプロファイリングの手法を提案する．

3.1 アスペクト指向を用いたトレース生成

分散システムの動作状況を把握するための手段として，実行状況モニタが知られている[13)]．モニタは実行時のシステム動作を監視し，あるイベントが発生すると，あらかじめ定義した処理を行う．あらかじめ定義されたイベント発生時の処理とは，例えば，ロギング処理や発生順序 (*happens-before*[14)]) の判定が考えられる．

本手法では，モニタの実装には**アスペクト指向プログラミング** *(AOP[6)])* の処理系である *AspectJ* を用いる．アスペクト指向プログラミングとは，オブジェクト指向プログラミングではモジュール化を行いにくい横断的関心事をアスペクトとしてモジュール化する技術であり，AspectJ を用いることで実際のプログラムに手を加えずにモニタの処理を追加することが可能となる．

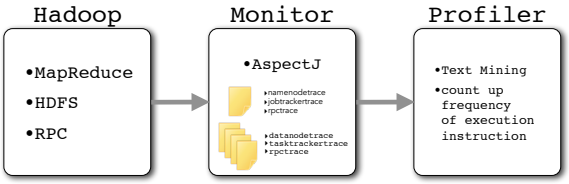

図 1 提案システム概要図

AspectJ の基本的な機能として，**アスペクト** *(aspect)*，**ポイントカット** *(point cut)*，**アドバイス** *(advice)*，**ジョインポイント** *(join point)* を持つ．AspectJ の動作の概要は，まずジョインポイントがコード上の位置を示しており，ポイントカットはジョインポイントの集合からジョインポイントの部分集合を選択する．ポイントカットで指定されたジョインポイントの位置で実行されるイベントをアドバイスという．これらのポイントカットとアドバイスの組み合わせを指定したモジュールをアスペクトと呼ぶ．

Hadoop は Java で記述されているため，モニタの実装にはアスペクト指向の Java 実装である AspectJ が利用できる．また，性能の面において，AspectJ を用いた場合，アスペクトはバイトコードへ組込まれるため，その他の動的解析ツールである *jdb* **デバッガ**☆1や，*JVMTI (Java Virtual Machine Tool Interface)*[15)] よりも高速に動作することが期待できる．AspectJ と同様にバイトコードの書き換えを行うツールとして，ASM☆2や，Javassist☆3があげられるが，Hadoop のソースコードは 10 万行にも及ぶため，同期制御などを把握して Hadoop の Java バイトコードを書き換えることは困難である．また，モニタの機能の実装は少ないコストで実現でき，簡単に取り除くことができるべきである．以上より，トレース生成には AspectJ を用いる．

AspectJ で記述したアスペクトは *ajc (AspectJ-Compiler)*☆4によってコンパイルされ，ポイントカットで指定した Hadoop ソースコードの位置へ Java バイトコードに変換されたアドバイスが組み込まれる．

☆1 JDK に標準で付属しているコマンドラインデバッガ
☆2 http://asm.ow2.org/
☆3 http://www.csg.is.titech.ac.jp/~chiba/javassist/
☆4 http://www.eclipse.org/aspectj/index.php

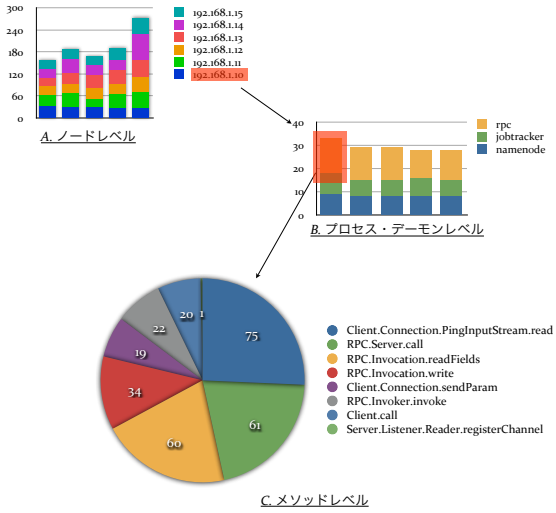

図 2 各粒度ごとのプロファイリング概要図

Hadoop 動作時の実行命令をポイントカットで指定し，その際のアドバイス処理として，ポイントカットで指定したジョインポイントでの実行命令をロギングする，このようなアスペクトモジュールである”**モニタ**”を作成する (図 1)．ここで注意しなければならないのが，テスト・デバッグのためのシステム動作を解析するモニタがシステムの性能を損なわせることは好ましくない．

3.2 Hadoop に有効なプロファイリング

我々は，Hadoop のような大規模な分散システムでは分割統治のアルゴリズムを用いているため，稼働中に繰り返し実行される処理が存在することに注目し，実行命令の発生頻度に注目したプロファイリングを提案する．アスペクト指向を用いた実行命令のロギングを行うモニタで取得したトレースデータについて，単位時間あたりに確認できた実行命令の数を数え上げることでプロファイリングを行う．

ロギングされた実行命令はタイムスタンプ以外に，ノードの IP アドレスやデーモン・プロセスの名称などの付加情報を持たせることで，プロファイリングを**粒度**を変えて行うことが可能となる．ここでの粒度とは，ロギングされた実行命令の付加情報による数え上げの際の分割の境界を意味する．実行命令の付加情報より，”**ノードレベル**”，”**デーモン・プロセスレベル**”，そして”**メソッドレベル**”の 3 種類のプロファイリングの粒度が考えられる．

ノードレベルのプロファイリングでは，分散システムを構成するクラスタ全体に対するプロファイリングを，ロギングされた実行命令の付加情報であるノード IP で分割して行う．これは最も粗い粒度でのプロファイルであり，分散システム全体の稼働状況や，分散システムを構築するノードの稼働状況を，ノード内で実行した命令の総数で簡単に表現する．プロセスレベルのプロファイリングでは，ロギングされた実行命令をプロセス・デーモン付加情報で区別しプロファイルを行う．Hadoop のマスタならば NameNode，JobTracker，そして RPC についてのプロファイルを行い，スレーブならば，DataNode，TaskTracker，RPC についてのプロファイルを行う．メソッドレベルのプロファイリングは，最も細かい粒度のプロファイリングとなる．

以上の，プロファイリングの粒度ごとにプロファイリングを行うことで，多数のノードを用いた場合でも目的のノードの，注目するプロセス・デーモンの振る舞いの監視がしやすくなる (図. 2)．開発者はまずノードレベルでのプロファイリングによって，大域状態の判定を[16] 行ったり，実行命令の発生回数のノード間の違いから乖離度[17][18] を計算し，障害の可能性のあるノードを発見し，次にプロセス・デーモンレベルのプロファイリング結果を用いてさらに障害原因の範囲であるプロセス・デーモンを限定し，最終的にメソッドレベルのプロファイリングによってシステム動作の評価を行う．

表 1 計算機性能

CPU	Intel(R) Core(TM) i5-3470 CPU
動作周波数	3.20GHz
コア数	4
RAM	8GB
ディスク	1TB SATA HDD (7200 回転)

4. 実装と実験

検証対象の分散システムとして Apache Hadoop を用いたシステムを構築する．AspectJ を用いたモニタを実装し，検証対象の Hadoop クラスタに配置し，テスト動作させ実行命令のロギングを行う．モニタが収集する実行命令の列であるトレースを用いて，提案するプロファイリングを適用する．

4.1 実験環境

表 1 に示す性能の計算機を 6 台用いて，Hadoop 検証環境を構築する．その際の各種ソフトウェアのバージョンを表 2 に示す．Hadoop クラスタのデーモンの配置を表 3 に示す．Hadoop の各種設定項目を表 4, 表 5, 表 6 に示す．

4.2 モニタの実装

Hadoop クラスタの動作時の振る舞いを取得し，実行命令のロギングを行うモニタを実装する．

2.1 より，Hadoop は分散ファイルシステム HDFS と

表 2 ソフトウェアのバージョン

OS	Linux 2.6.32-279.el6.x86_64 SMP
Hadoop	1.0.3
AspectJ	1.7.1
Java	1.7.0

表 3 Hadoop クラスタ構成

IP アドレス	デーモン
192.168.1.10	NameNode, JobTracker
192.168.1.11 ~ 15	DataNode, TaskTracker

表 4 mapred-site.xml

プロパティ名	設定値
mapred.tasktracker.map.tasks.maximum	4
mapred.tasktracker.reduce.tasks.maximum	4
mapreduce.reduce.maxattempts	2
mapreduce.map.memory.mb	512
mapreduce.reduce.memory.mb	512
mapreduce.map.java.opts	-Xmx512m -Xms512m
mapreduce.reduce.java.opts	-Xmx512m -Xms512m
mapred.child.java.opts	-Xmx512m -Xms512m -Djava.net.preferIPv4Stack=true
mapreduce.task.io.sort.mb	128

表 5 hdfs-site.xml

プロパティ名	設定値
dfs.replication	1
dfs.permissions.enabled	false
dfs.block.size	134217728

大規模データの処理環境 MapReduce から構成される，マスタスレーブ型のアーキテクチャであり，マスタでは NameNode と JobTracker デーモンが動作し，スレーブでは DataNode と TaskTracker デーモンが動作する．Hadoop クラスタでの協調動作は RPC 通信によって行われる．以上より，モニタアスペクトのポイントカットで選択するべきジョインポイントのコード上の位置を，NameNode，DataNode，JobTracker，TaskTracker，そして RPC 通信を行うクラス内のジョインポイントとする．モニタは各ノード内のプロセス・デーモンごとにトレースを生成する (図. 3).

また実行命令の発生頻度によるプロファイリングを行うために，ロギングされる実行命令はその命令の**タイムスタンプ**と，命令を実行したノードの *IP* **アドレス**，その実行命令のロギングを現在行っている**アスペクト自身の名称**などの付加情報をもたせる．以下では，RPC 通信を取得する際の AspectJ によるモニタの実装について記述する．

Hadoop のノード間の RPC 通信には，`org.apache.hadoop.ipc` パッケージが使われている．ここへ AspectJ を用いたアスペクトによってモニタ機能を埋め込む．モニタのソースコードの一部を以下に示す．

表 6 core-site.xml

プロパティ名	設定値
io.file.buffer.size	65536
io.sort.factor	20
io.sort.mb	600
fs.inmemory.size.mb	200

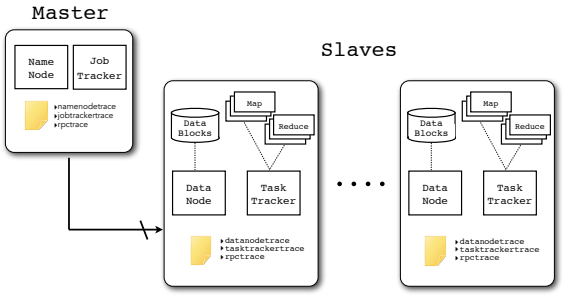

図 3 Hadoop アーキテクチャと各種トレース

ソースコード 1 がモニタ機能のアスペクトのポイントカットを示し，ソースコード 2 がソースコード 1 のポイントカットに対するアドバイスを示す．

```
1  privileged aspect RPCMonitor {
2    public pointcut MethodExecution()
3      : execution(public * *.*(..))
4      && within(org.apache.hadoop.ipc.*)
5      && !execution(* org.apache.hadoop.metrics.**.*(..))
6      && !execution(* org.apache.hadoop.metrics2.**.*(..))
7      && !execution(* org.apache.hadoop.security.**.*(..))
8      && ...
9      && !within(Pointcut_RPCMonitor)
10     && !cflow(adviceexecution());
11 }
```

ソースコード 1 : ポイントカット

ソースコード 1 は，モニタ機能のアスペクトのポイントカットを示している．このポイントカットは，`org.apache.hadoop.ipc.RPC` クラスに含まれる全てのメソッドの実行をジョインポイントとして選択する．ただし，Hadoop 自身が提供するメトリクス取得のためのパッケージである `org.apache.hadoop.metrics` と `org.apache.hadoop.metrics2` はポイントカットに含めない．これらの metrics パッケージは，HDFS，MapReduce 処理，そして JVM や RPC 通信動作についての統計情報を取得するために大量に呼び出されるため，大量のジョインポイントが発生し，モニタがシステム動作に大きな負荷を与えると考えられる．システム動作の解析のためのモニタが，もともとのシステム動作に大きな負荷を与えることは望ましくない．

```
1  privileged public aspect RPCMonitorAdvice {
```

```
2
3   private static String hostname = getHostName(); //ホスト名取得
4
5   public static long now() {
6     return System.currentTimeMillis();
7   }
8   before() : RPCMonitor.MethodExecution() {
9     Signature signature = thisJoinPoint.getSignature();
10    StringBuffer trace = new StringBuffer();
11    trace.append(now()+hostname+current_pointcut+"={");
12    trace.append(signature.toString());
13    trace = appendArgText(trace, thisJoinPoint); // 引数を取得
14    trace.append("}");
15  }
16 }
```

ソースコード 2 : アドバイス

ソースコード 2 は．ソースコード 1 で選択したポイントカットに対するアドバイス処理を記述している．モニタの処理とは．ポイントカットで指定したジョインポイントの位置において，実行されたメソッドとその引数をタイムスタンプを付加してロギングを行うことである．これによってトレースを生成する．またロギングする情報には，自身のホスト名と，この場合は RPC に関するトレースであることもトレース情報に付加する．

モニタのアドバイス処理によって作成されるトレースの出力結果の一行を以下に示す．

```
1351576371078-192.168.1.10-rpcmonitor = {
  Writable org.apache.hadoop.ipc.RPC.Server.call(Class, Writable, long),
  [interface org.apache.hadoop.hdfs.protocol.ClientProtocol,getFileInfo
  (/hadoop/mapred),1351576371078]
}
```

モニタの出力結果

モニタの出力結果の 1 行目は，命令が実行されたノードの IP アドレス (="192.168.1.10") と，その実行命令をポイントカットで取得したアスペクトの名前 (="rpcmonitor")，そしてタイムスタンプ (="1351576371078") を意味する．2～4 行目が，ロギングされたメソッド (="org.apache.hadoop.ipc.RPC.Server.call") と，その引数となる．

同様に HDFS，MapReduce のデーモンに対しても AspectJ を用いたモニタを配置した．

4.3 モニタの評価

システムの動作状況を取得するためのモニタアスペクトの処理が，システム性能に与える影響を評価する．本研究では，Hadoop のサンプルプログラムである Terasort を用いてモニタの性能のテストを行った．Terasort プログラムは，MapReduce の性能評価として利用されており[20]，テキストデータを key によってソートし，そのまま出力するプログラムである．テキストデータの生成には teragen プログラムを使用し，作成した 1GB，10GB，100GB のデータに対して，モニタアスペクトを使用した場合と使用しない場合のスループットを計測した．性能評価の際には，HDFS は 1GB，10GB，100GB のデータのみ保持しており，以前の terasort の結果の出力データは毎回削除し HDFS 動作の負荷を均等にし，また MapReduce 処理時に以前のヒープメモリを参照しないように，メモリも計測する前にクリアするものとする．実験結果を表 7 に示す．

これより，モニタを使用した場合のスループットに対する，モニタを使用しない場合のスループットの性能比は，1GB，10GB，100GB の順に，84.1%，88.3%，96.2%となり，モニタアスペクトがもともとのシステム動作に与える負荷は非常に小さい．また，MapReduce 処理を行う対象のデータサイズが大きくなり，システム動作時間が長くなるほど，モニタの有無による性能の違いは小さくなる．Hadoop が TB や，PB のビッグデータを対象とするシステムであることから，実際の使用場面からすると作成したモニタが与える負荷は非常に小さいと言える．100GB のテストデータに対する結果より，6.43KB/s でトレースファイルが生成されることとなる．

次に，100GB のテスト用データに対して terasort プログラムを行った際に生成されたトレースについて，モニタの種類ごとのトレースのサイズを表 8 に示す．192.168.1.11~15 のスレーブにおける TaskTracker，RPC 通信によって生成されるトレースのサイズに偏りが見られるが，これは実行命令数の違いを意味する．しかし，トレースサイズが小さいことから 192.168.1.13 の性能が他のノードに比べて，低いとは言えない．マスタからスレーブへの距離はここではすべて同一であるため，タスクの割当のランダム性，ネットワーク上の通信パケットの遅延や紛失，例外の発生が少なかったスの確保が滞りなく行われた，などさまざまな要因が考えられる．

4.4 プロファイリング

モニタを各ノードに配置した Hadoop クラスタを動作させ，その際の実行命令をモニタがロギングすることで生成されたトレースを用いて，提案する発生頻度によるプロファイリングを行う．プロファイリングのための Hadoop のテスト動作として，100GB データの terasort 処理を用いる．100GB テストデータに対して terasort 処理を実行し，その際に配置したモニタが生成したトレースについて，"**ノードレベル**"，"**プロセス・デーモンレベル**"，**実行命令レベル**でのプロファイリングを行う．

まず，ノードレベルでのプロファイリングを行う．**ノードごとの**単位時間あたりに実行された実行命令を

表 7 モニタが与える負荷の評価 –terasort–

データサイズ [GB]	モニタの有無	実行時間	スループット [MB/s]	トレースサイズ [MB]
1	○	2m25s (145s)	6.9	2.4
1	×	2m02s (122s)	8.2	0
10	○	8m45s (525s)	19.0	3.6
10	×	7m45s (465s)	21.5	0
100	○	1h21m54s (4,914s)	20.4	31.6
100	×	1h18m37s (4,717s)	21.2	0

※ Hadoop が提供するサンプルプログラム teragen によって 1GB，10GB，100GB テスト用データを作成し，terasort プログラムを用いて MapReduce 処理の性能をモニタがある場合とない場合で比較を行った．

表 8 各種モニタによって生成されるトレースサイズ比較

host	NameNode	JobTracker	DataNode	TaskTracker	RPC	合計
192.168.1.10	1.8	1.8	0	0	4.2	7.8
192.168.1.11	0	0	0.07	2.0	3.2	5.3
192.168.1.12	0	0	0.07	1.9	3.0	5.0
192.168.1.13	0	0	0.08	1.3	2.2	3.6
192.168.1.14	0	0	0.08	1.8	2.8	4.7
192.168.1.15	0	0	0.08	1.9	3.3	5.3

※ 各ファイルのサイズは MB 単位である．

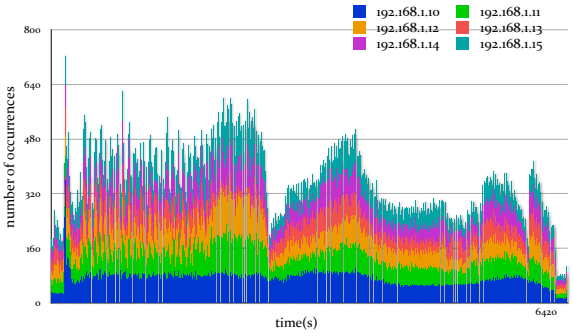

図 4 ノードごとの実行命令回数プロファイル

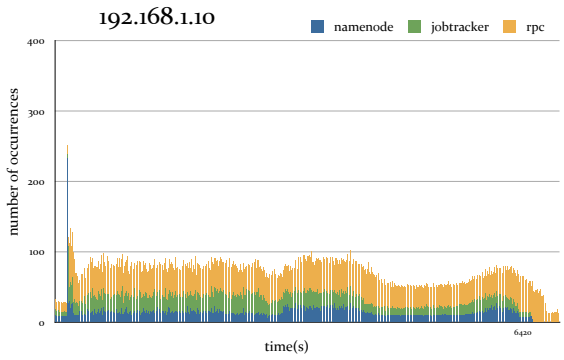

図 5 マスタのプロファイリング結果

すべて数え上げることでプロファイリングを行う．プロファイリング結果を図 4 に示す．この図より，全体の実行時間の中盤ではマスタを除く全てのスレーブノードでの実行命令数が急激に減少しているが，この時点は Map 処理から Reduce 処理へ移行する時間と一致する．このことから Map から Reduce への移行時の

図 6 スレーブ群のプロファイリング結果

投機的実行を行うことで Hadoop の処理性能が向上する可能性があることが推察できる．

次にプロセス・デーモンレベルでのプロファイリングを行う．プロセス・デーモンレベルのプロファイリングでは，各ノードでは NameNode，DataNode，JobTracker，TaskTracker，RPC 通信のデーモンやプロセスが動作するが，これらのプロセスやデーモンごとにプロファイリングを行う．マスタである 192.168.1.10 では NameNode と JobTracker に対するモニタと，RPC 通信に対するモニタを配置しているため，この 3 種類について単位時間あたりの実行命令数を数え上げる．同様にスレーブ群に対しては，DataNode と TaskTracker，RPC 通信によるトレースについてプロファイリングを行う．プロファイリング結果を図 5 と，図 6 に示す．

マスタでのプロファイリング結果からは終止動作が安定していることが伺える．一方スレーブでは，処

理内容が頻繁に変化していることが分かる．図 6 の 192.168.1.11 ノードについて考察すると，tasktracker での単位時間あたりの実行命令数が実行時間全体の中間でほぼゼロとなっていることから Map 処理と Reduce 処理へ移行する推移が見て取れる．Map 処理から Reduce 処理への移行期間では，RPC 通信による実行命令数が増加しているが，しかしこの RPC 通信の単位時間あたりの実行命令数はノードごとに偏りが見られるためこの RPC 通信のロードバランスを行うことでパフォーマンスの上のボトルネックが解消できる可能性が指摘できる．

最後に，実行命令レベルでのプロファイリングを行う．取得した実行命令のメソッド名ごとに実行命令を数え上げることで，プロファイリングを行うため，もっとも細かい粒度でのプロファイルが，可能である．結果の一部を以下に示す．

```
1 : org.apache.hadoop.mapred.JobTracker.heartbeat(TaskTracker..
2 : org.apache.hadoop.mapred.JobTracker.getTaskCompletionEven..
5 : org.apache.hadoop.mapred.JobTracker.getTaskTracker(String)
1 : org.apache.hadoop.mapred.JobTracker.getReduceTaskReports..
5 : org.apache.hadoop.mapred.JobTracker.ExpireLaunchingTasks..
7 : org.apache.hadoop.mapred.JobTracker.getJob(JobID)
1 : org.apache.hadoop.mapred.JobTracker.getTaskTrackerStatus..
1 : org.apache.hadoop.mapred.JobTracker.getJobStatus(JobID)
```

結果 1 : 192.168.1.10 の JobTracker の実行命令数分布

```
4 : org.apache.hadoop.mapred.TaskTracker.LRUCache.get(Object)
8 : org.apache.hadoop.mapred.TaskTracker.MapOutputServlet.doGet..
1 : org.apache.hadoop.mapred.TaskTracker.getLocalJobDir(String,..
2 : org.apache.hadoop.mapred.TaskTracker.getMapCompletionEvents..
4 : org.apache.hadoop.mapred.TaskTracker.getLocalTaskDir(String ..
2 : org.apache.hadoop.mapred.TaskTracker.getUserDir(String)
2 : org.apache.hadoop.mapred.TaskTracker.getLocalTaskDir(String ..
1 : org.apache.hadoop.mapred.TaskTracker.FetchStatus.getMapEven ..
```

結果 2 : 192.168.1.11 スレーブの TaskTracker の実行命令数分布

5. 評価とまとめ

本研究では，特に大規模な分散システムに対する軽量なトレースの取得手法と，トレースを用いた分散システム向けのプロファイリング手法を提案した．アスペクト指向を用いることで，もとのコードに変更を加えることなくモニタ機能を持つモジュールを作成し，システム動作時の実行命令をロギングしトレースを作成する．大規模な分散システムでは分割統治法による処理が行われており，繰り返し実行される命令が存在することから，発生頻度によるプロファイル手法が有効である．

Hadoop のデバッグの際には，一般的なシングルプロセスのプログラムのデバッグ手法と同様に，テキストとしてのログを取得し解析することはノード数が増加した場合に，確認の作業が困難であることがあげられる．Hadoop では多数の計算機を用いる為に，大量のログが作成されると開発者は重要なデバッグの為の情報を見逃す可能性が考えられる．

Hadoop の動作状況の監視には Ganglia による各種メトリクスのグラフ化が有用であり広く活用されているが，Ganglia によるメトリクス情報はシステム運用者に対しては有効であると考えられるが，システムの開発者に対してはより詳細な解析が必要であると考えられる．本提案手法は，AspectJ を用いた低コストで生成可能な軽量なモニタによって，ロギングされた実行命令の列であるトレースを生成し，生成されたトレースについて，粒度ごとのプロファイルを行うことでシステムの動作の監視をシステム全体からメソッドレベルまで可能とする．

実際に，アスペクト指向の Java 実装である AspectJ を用いてシステム動作時の実行命令を取得するモニタを実装し，検証対象の Hadoop クラスタについて実行時トレースの取得を行った．軽量なモニタはシステムのもともとの動作のパフォーマンスを大幅に下げることなくトレース情報を生成することが可能である．実行命令の発生頻度によるプロファイルは，プロファイルを行う粒度によって既存手法より詳細な解析が可能である．

既存研究[21] と比較し，対象とする分散システムの規模の違いが挙げられる．大規模な分散システムで行われる分割統治アルゴリズムのシステムに対しては，実行命令数の発生回数を数えることがシステム動作の把握に有効と考えており，処理の発生順序ではなく実行命令の発生頻度を用いたプロファイル手法であるという違いがある．そのため本手法では論理時計[14] をロギング情報に持たせていないためモニタがシステムに与える負荷は軽く，大規模な分散システムにも使用することが可能である．

今後の課題として，実行命令レベルのプロファイル結果を用いた障害検知のアルゴリズムの作成が考えられる．

参 考 文 献

1) Apache Software Foundation. (2007) ”Hadoop”. [Online] http://hadoop.apache.org/
2) Tom White : ”Hadoop :The Difinitive Guide”, O’Rilly Media, (2009) (Tom White, 玉川竜司・兼田聖士 (訳)(2010) : Hadoop, O’REILLY)
3) A. V. Mirgorodskiy, N. Maruyama, B. P. Miller: ”Problem Diagnosis in Large-Scale Computing Environments”, ACM/IEEE conference on Supercomputing, pp.88-88 (2006)
4) Newman, HB and Legrand, IC and Galvez, P: ”Monalisa: A distributed monitoring service architecture”, arXiv preprint cs/ . . . , (2003)
5) D. Olge, K. Schwan, and R. Snodgrass. ”Application-Dependent Dynamic Monitoring of Distributed Systems”. IEEE Transactions on Parallel and Distributed Systems, 21(4):593—622, December 1989.
6) Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold : ”An Overview of AspectJ”, ECOOP ’01 Proceedings of the 15th European Conference on Object-Oriented Programming, Springer-Verlag London, UK, pp.327-353 (2001)
7) Avgustinov, Pavel and Bodden, Eric and Hajiyev, Elnar : ”Aspects for trace monitoring”, Formal Approaches to . . . , pp.1-20 (2006)
8) Charles E McDowell, David P Helmbold : ”Debugging Concurrent Programs”, ACM Computing Surveys, pp.593-622 (1989)
9) Cristian, Flaviu : ”Understanding Fault-Tolerant Distributed Systems”, pp.1-45 (1993)
10) Jeffrey Dean and Sanjay Ghemawat, ”MapReduce: Simplified Data Processing on Large Clusters”, in OSDI, 2004
11) Konstantin Shvachko, Hairong Kuang, Sanjay Radia and Robert Chansler, ”The Hadoop Distributed File System”, 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp.1-10, (2010)
12) Tom Cook: ”A Day in the Life of Facebook Operations”, Velocity 2010, [Online]http://velocityconf.com/velocity2010 (2010)
13) C.E.McDowell, D.P.Helmbold: ”Debugging Concurrent Programs”, ACM Computing Surveys, Vol.21, No.4, pp.593-622 (1989)
14) L.Lamport: ”Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, Vol.21, No.7, pp.558-565 (1978)
15) C.K Prasad, Rajesh Ramchandani, Gopinath Rao, and Kim Levesque : ”Creating a Debugging and Profiling Agent with JVMTI”, (2004)
16) 六沢一昭：”非同期分散環境における大域状態決定方式”，情報処理学会論文誌，Vol.41, No.5, pp.1517-1525 (2000)
17) N. Maruyama, S. Matsuoka: ”Autonomous Fault Diagnosis in Clustered Distributed Environments”, JSSST (2007)
18) P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson : ”Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization”. ICAC (2005)
19) 松下 光範, ”情報粒度 Information granularity”, 日本ファジィ学会誌 10(2), 81, (1998) [Online]http://ci.nii.ac.jp/naid/110002939234
20) Owen O Malley and Arun C. Murthy. ”Winning a 60 second dash with a yellow elephant”, (2009)
21) 新垣紀子，山崎憲一，天海良治: ”分散プログラムの動作理解のための分散プロファイラ”，情報処理学会第 44 回全国大会，pp.265-266 (1992)
22) Vilas Jagannath, Zuoning Yin, and Mihai Budiu : ”Monitoring and Debugging DryadLINQ Applications with Daphne”, 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp.1266-1273 (2011)
23) Andrew.S.Tanenbaum, Maarten.van.Steen : ”Distributed Systems: Principles and Paradigms”, Prentice Hall(2002) (アンドリュー.S. タネンバウム，マールティン. ファン. スティーン，水野忠則・宮西洋太郎・鈴木健二・西山智・佐藤文明・東野輝夫 (訳) : 分散システム 原理とパラダイム，O’REILLY (2003))