Vol.2013-SE-179 No.21
2013/3/12

リポジトリマイニングに対するHadoopの性能評価

大坂 陽[1,a)]　山下 一寛[1]　亀井 靖高[1]　鵜林 尚靖[1]

概要：リポジトリマイニング分野において，リポジトリのデータサイズが増加し続けていることから，リポジトリマイニングの処理性能の向上は本研究分野の主な課題の 1 つである．本稿では，Hadoop を用いたスケールアウトによる処理性能の向上を図り，従来手法による逐次処理との性能差を実験的に評価する．リポジトリに保管されているコミットログデータから，ソフトウェアメトリクスの計算を行った．Eclipse，及び，Android を題材としたケーススタディを行った結果，ログデータのサイズが十分（150MB 以上）に大きい場合，Hadoop を用いることは有用であることがわかった．また，対象期間の大きさを変えながら実験を行った結果，全期間のログデータを用いた場合，Hadoop を用いる効果は最も大きく，Eclipse プロジェクトで 1.71 倍高速に，Android プロジェクトで 42.27 倍高速に計算できることがわかった．

1. はじめに

現在，ソフトウェア開発ではそのプロジェクトで計測可能な多くの情報（ソースコード，バージョン管理情報，開発者間でやり取りされたメール等）が版管理システムやバグ管理システムといったソフトウェアリポジトリに保管されている．それらのリポジトリには，実際のソフトウェア開発履歴が蓄積されていることから，プロジェクト特有の有用な情報が豊富に含まれていると考えられる [1]．この考えから，ソフトウェアリポジトリに対するデータマイニング（以降，リポジトリマイニング）はソフトウェア開発において必要不可欠なものとなっている．

リポジトリマイニング分野における課題の 1 つは，どのようにして大規模なデータを取り扱うかである [1]．このリポジトリマイニングで扱われるログデータは，大規模開発では大きいものとなる．そのため，分析データをログデータから抽出・加工する処理には大量の時間を要する．したがって，リポジトリマイニングの処理性能の向上が求められる．

そこで本稿では，リポジトリマイニング分野に対して Hadoop [2][3] による処理の高速化を図る．Hadoop は，大規模なデータを複数のサーバで処理するためのオープンソースの並列処理フレームワークである [3]．Hadoop は既に Web サービスに対して導入され，多数の成功事例が報告されているものの，リポジトリマイニング分野への Hadoop の適用事例は少なく，性能評価もわずかである．

そこで，本稿では Hadoop を用いたリポジトリマイニングの性能評価を行う．リポジトリマイニングに対する Hadoop の性能評価として，リポジトリマイニング研究として広く取り組まれているテーマの 1 つである，不具合モジュール予測に用いるためのメトリクスの算出を適用事例として取り上げる．評価尺度として実行時間を用い，Hadoop で実行した場合と従来手法（サーバ 1 台による逐次処理）で実行した場合で，実行速度にどの程度違いがあるかを比較する．

2. 背景

2.1 リポジトリマイニング

リポジトリマイニングは，リポジトリにある開発履歴に対してデータマイニング技術を適用し，有用な知見を得ることである．リポジトリマイニングの工程は，大きく分けて以下の 2 つに分かれる．

工程 1：　データ準備
ソフトウェアリポジトリから分析に必要なデータを抽出し，扱いやすいデータ形式に変換する．

工程 2：　データ分析
工程 1 で準備したデータをもとに統計解析手法や機械学習手法などを用いて分析を行い，有用な知見を得る．分析には統計解析ツールの R [4] や Weka [5] などが用いられる．

ソフトウェアリポジトリのデータサイズは増加し続け，テラバイトクラスにも及ぶ [6]．このために，工程 1 において多くの時間を要している．これはリポジトリマイニングにおける主な課題の 1 つである．そこで，本稿では，

[1] 九州大学
Kyushu Uniersity
[a)] osaka@posl.ait.kyushu-u..ac.jp

Hadoop を用いたスケールアウト[*1]による処理性能の向上を図り，工程 1 のデータ準備について高速化を目指す．

本稿では，リポジトリマイニング研究の 1 つとして広く取り組まれている，不具合モジュール予測 [7] に用いるためのメトリクス（ソースコード行数といったソフトウェアの特徴量）の算出を行う．不具合モジュール予測は，ソフトウェアテストやレビューの効率化に寄与する有用な分野の 1 つであり，モジュールの不具合の有無について予測を行う研究である．不具合モジュール予測に用いられるメトリクスには，プロダクトメトリクスとプロセスメトリクスの 2 種類がある．不具合モジュール予測においてはプロセスメトリクスを用いるほうが適している [7][8][9][10] とされている．また，プロセスメトリクスは版管理システムのコミットログから計測可能であり，プログラムのソースコード解析も必要としない．そのため，本稿ではプロセスメトリクスの算出を行う．

2.2 Hadoop

Hadoop は大規模なデータを複数のサーバで処理するためのオープンソースの並列分散処理フレームワークのことである [3]．Hadoop のサーバはマスターサーバとスレーブサーバという 2 種類のサーバから構成される．マスターサーバは Hadoop 全体の管理を行い，スレーブサーバは複数台用意され，データの保存，及び，分散処理の実行を行う．Hadoop の特徴はスレーブサーバを増やすことによって，データ保存容量の拡張，分散処理能力の向上が容易に行えることである．Hadoop には様々なプロジェクトがあり [2]，HDFS(Hadoop Distributed File System) と呼ばれる分散ファイルシステムと，MapReduce と呼ばれる並列分散処理フレームワークが Hadoop の主要プロジェクトである．

2.2.1 HDFS

HDFS は，大規模なデータを格納するためのファイルシステムである．HDFS では，格納するファイルを任意の大きさのブロックで分け[*2]，そのブロックをスレーブサーバに分散して格納する．マスターサーバはブロックの格納先，およびスレーブサーバの動作を管理する．

2.2.2 MapReduce

MapReduce は，大規模なデータを並列分散処理するためのフレームワークである．マスターサーバが実行処理を各スレーブサーバに割り当て，スレーブサーバは割り当てられた実行処理を行う．MapReduce の入出力のパスは HDFS で行われ，[key/value] のペアのデータ集合が扱われる．MapReduce の処理は次の 2 つの処理に分けられる．

- Map 処理
  入力データの [key/value] の抽出・変換を行い，中間生成物の [key/value]’ を出力する．Map 処理の実行クラスを mapper と呼ぶ．
- Reduce 処理
  Map 処理の中間生成物の [key/value]’ を集約し，最終結果を [key/value]” として出力する．Reduc 処理の実行クラスを reducer と呼ぶ．

Map 処理の入力は HDFS に格納されているファイルである．実行クラスである mapper は，入力ファイルの 1 行ごとに [key/value] のデータ集合を取り出し，プログラムされた処理を実行し，[key/value]’ のデータ集合を出力する．[key/value] のペアのデータ集合の取り出し方はプログラム時に設定できる．mapper の数は入力ファイルの規模に応じて Hadoop によって調整される．

Reduce 処理では，実行クラスである reducer に Map 処理の出力の [key/value]’ のペアのデータ集合が渡される．このとき，同じ key は同じ reducer に自動的に振り分けられ，同じ key ごとに処理が実行され，[key/value]” のデータ集合が出力される．reducer の数だけファイルが生成され，HDFS に格納される．reducer の数はプログラム時に設定でき，出力結果を 1 つにまとめたいときは reducer の数を 1 に，集約処理が必要ないときは reducer の数を 0 にも設定できる．

3. Hadoop によるリポジトリマイニングの並列処理

3.1 実装する処理の概要

表 1 に不具合モジュール予測に用いるプロセスメトリクスを示す．本稿では，num_changes_co-changed_files の計算を実装し，Hadoop で実行した場合と従来手法（サーバ 1 台による逐次処理）で実行した場合で，実行速度にどの程度の違いがあるのかを比較する．num_changes_co-changed_files とは，あるファイルが変更された際，同時に変更された他のファイルの変更回数を全て足し合わせた数値のことである．num_changes_co-changed_files を選んだ理由は，表 1 の中でも最も計算量が大きく，性能の比較が行いやすいと考えたためである．

num_changes_co-changed_files の計算方法を，図 1 を用

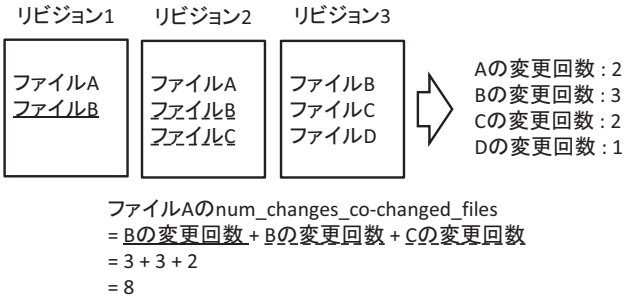

図 1 num_changes_co-changed_files の計算方法

*1 サーバの台数を増やすことによって処理能力を向上すること
*2 初期設定で 64MB のブロックに分けられる

表 1　プロセスメトリクス一覧

メトリクス	説明	計算量†
pre_defects	リリース前のバグの数	$O(f*\overline{r})$
pre_changes	リリース前の変更回数	$O(f*\overline{r})$
file_size	ファイルの行数	$O(f)$
num_co-changed_files	コミットごとの同時変更ファイル数	$O(f*\overline{r}*\overline{f})$
size_co-changed_files	コミットごとの同時変更ファイルのサイズ	$O(f*\overline{r}*\overline{f})$
modification_size_co-changed	コミットごとの同時変更ファイルの変更行数	$O(f*\overline{r}*\overline{f})$
num_chnages_co-changed_files	ファイルごとの同時変更されたファイルの変更回数の和	$O(f*\overline{r}+f*\overline{r}*\overline{f})$
pre_defects_co-changed_files	コミットごとの同時変更ファイルのリリース前のバグ数	$O(f*\overline{r}+f*\overline{r}*\overline{f})$
latest_change_before_release	最後の変更からリリースまでの時間	$O(f*\overline{r})$
age	ファイルの年齢	$O(f*\overline{r})$
modification_size_files	ファイルの合計追加・削除行数	$O(f*\overline{r})$
modification_size_package	パッケージの合計追加・削除行数	$O(f*\overline{r})$
num_authors_files	ファイルごとの，そのファイルを変更した編集者数	$O(f*\overline{r})$

† f はファイル数，$\overline{r}$ は 1 ファイルにおける平均リビジョン数，$\overline{f}$ は 1 リビジョンにおける平均ファイル数を表す．

いて説明する．例えば，ファイル A の num_changes_co-changed_files を求める．ファイル A と同時にリビジョン 1 ではファイル B が変更され，リビジョン 2 ではファイル B とファイル C が変更されている．したがって，ファイル A のリビジョン 3 終了時点の num_changes_co-changed_files = ファイル B の変更回数＋ファイル B の変更回数＋ファイル C の変更回数 = 3+3+2 = 8 となる．

num_changes_co-changed_files を求めるために，次の 3 つの処理を要する．

パース処理　ログデータを解析し，扱いやすいデータ型に変換．本稿では，入力となるログデータとして，git の log コマンド[*3]から得られる形式を扱う．

Step 1.　各ファイルの変更回数を計算
$O(f*\overline{r}+f*\overline{r}*\overline{f})$ の $f*\overline{r}$ の部分

Step 2.　num_changes_co-changed_files を算出
$O(f*\overline{r}+f*\overline{r}*\overline{f})$ の $f*\overline{r}*\overline{f}$ の部分

$O(f*\overline{r}+f*\overline{r}*\overline{f})$ の f はファイル数，$\overline{r}$ は 1 ファイルにおける平均リビジョン数，$\overline{f}$ は 1 リビジョンにおける平均ファイル数を表す．

3.2　従来手法による算出法

3.2.1　パース処理

ログデータは，1 行ごとにリビジョンごとの変更されたファイル，変更された日付，編集者名などが記録してある．このログデータから，次の 2 つのハッシュを作成する．key にリビジョン ID（変更履歴を識別する ID），value に変更されたファイルのリストをもつ $REVS$ のハッシュと，key に変更されたファイル，value にそのファイルが変更されたリビジョン ID のリストをもつ $FILES$ のハッシュの 2 つである．

[*3] `git log --pretty=format:'<head>%H,%T,%P,%an,%ae,%ad,%cn,%ce,%cd,%s' --name-only | tr '\n' ',' | sed -e 's/,*<head>/\n/g'`

Algorithm 1 Algorithm for Step1

```
1: changeNum ← 0
2: for each file in FILES do
3:     for each revision in REVS(file) do
4:         changeNum ← changeNum + 1
5:     end for
6:     changeNum[file] ← changeNum
7:     changeNum ← 0
8: end for
```

Algorithm 2 Algorithm for Step2

```
1: changeNum ← 0
2: for each file in FILES do
3:     for each revision in REVS(file) do
4:         for each file' in COCHANGED(revision) do
5:             if file! = file' then
6:                 changeNum += changeNum[file']
7:             end if
8:         end for
9:     end for
10:    NumChangesCoChangedFiles[file] ← changeNum
11:    changeNum ← 0
12: end for
```

3.2.2　メトリクス算出

まず，Step 1 のアルゴリズムを，Algorithm1 として次に示す．今回のデータ参照にはすべてハッシュを用いており，データ 1 件あたりの計算量を $O(1)$ としている．Algorithm1 は各ファイルの変更回数を求める．2 行目で各ファイル $file$ に対して for 文を実行し，3 行目の for 文でファイル $file$ が変更された変更履歴 $REVS(file)$ を取り出す．その都度，$changeNum$ に 1 を加算して，6 行目の $changeNum$ がそのファイルの変更回数である．このときの計算量は $O(f*\overline{r})$ となる．

次に，Step 2 のアルゴリズムを，Algorithm2 として次に示す．Algorithm2 は各ファイルの num_changes_co-changed_files を求める．2 行目と 3 行目は Algorithm1 と

同じである．4 行目で変更履歴 *revision* から 2 行目のファイル *file* とは別のファイル *file′* の変更回数を，Algorithm1 の結果から取り出し，*changeNum* に足し合わせる．こうすることで num_changes_co-changed_files が求まる．したがって，計算量は $f * \overline{r} * \overline{f}$ となる．

3.3 提案手法による算出法

3.3.1 パース処理

入力には従来手法と同様のログデータを用いる．mapper の入力の [key/value] は，[リビジョン ID/key 以外の行の内容] である（図 2）．mapper では，value から変更されたファイル名を抽出し，[リビジョン ID/変更されたファイル名のリスト] を出力する．mapper の出力の key には集約処理を行わないため，reducer の数は 0 に設定する．

3.3.2 メトリクス算出

Step 1. MapReduce による各ファイルの変更回数の計算方法を説明する（図 3）．Map 処理の入力にはパース処理の出力結果を使用する．まず，mapper の [key/value] として [リビジョン ID/変更されたファイル名のリスト] を入力する．そして，各リビジョンでどのファイルが変更されたかを示すために，[ファイル名/1] を出力する．reducer では mapper の出力をもとに，同じ key によって value を加算し，[ファイル名/変更回数] を出力することで，各ファイルの変更回数を求める．

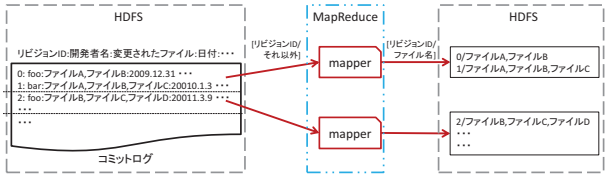

図 2 MapReduce によるパース処理の実装

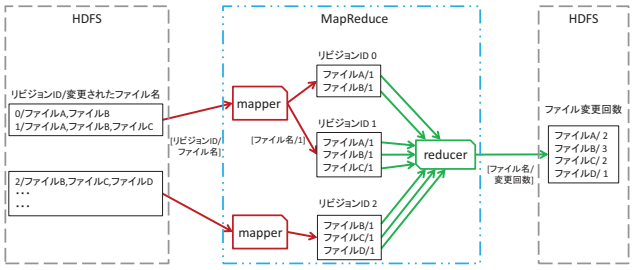

図 3 MapReduce による Step 1 の実装

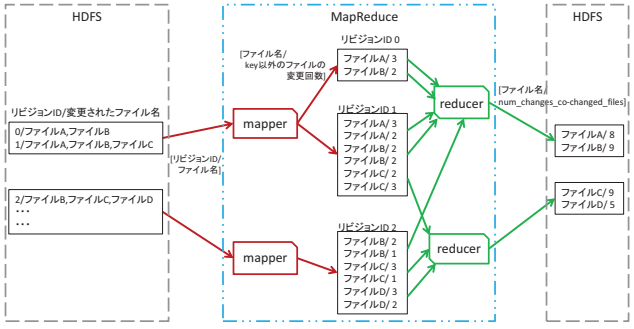

図 4 MapReduce による Step 2 の実装

Step 2. MapReduce による num_changes_co-changed_files の計算方法を説明する（図 4）．Step 2 の Map 処理の入力は Step 1 の Map 処理の入力と同じパース処理の出力結果である．num_changes_co-changed_files は，同時に変更されたファイルの変更回数の総和であるため，mapper の出力は [変更されたファイル名/key 以外のファイルの変更回数] である．この際，ファイルの変更回数は，Step 1 の出力結果を用いる．

リビジョン ID が 2 のとき（図 4 の中央下の部分）を例に，mapper での具体的な処理を説明する．リビジョン ID2 では，ファイル B，ファイル C，ファイル D が変更されている．同時に変更されたファイルの変更回数の総和を求めるため，ファイル B では，ファイル C，及び，ファイル D の変更回数を出力する．ファイル C，及び，ファイル D の変更回数は，Step 1 の出力結果（図 3 の右部分）より，2 と 1 であることがわかるため，[ファイル B/2]，[ファイル B/1] が出力される．同様の処理が，ファイル C と D にも行われる．

reducer には mapper の出力のうち同じ key の value が集められ，その value が足し合わせて出力される．reducer の処理は Step 1 の reducer の処理と同じである．このようにして reducer の出力 [ファイル名/num_changes_co-changed_files] が求まる．

Hadoop では 1 つのスレーブサーバで mapper，reducer ともに 2 つまでしか並列処理ができない．Step 2 の出力結果はファイルをまとめる必要がなく，分散処理を可能な限り行うことで高速化を図りたかったため，本稿で使用したスレーブサーバの台数（表 2）の 2 倍の 14 を reducer の数に設定した．

4. 評価

4.1 評価方法

num_changes_co-changed_files を，従来手法の逐次処理で求めた場合と，提案手法の並列分散処理で求めた場合について評価する．評価尺度は，実行時間である．

4.2 環境

本研究で使用したサーバの環境を表 2 に示す．マスターサーバが 1 台，スレーブサーバが 7 台で構成される．従来方式の逐次処理の実行環境はマスターサーバで行い，Java で実装した．

表 2　実験に用いたサーバの環境

	マスターサーバ	スレーブサーバ ×7
CPU	Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz , 6 cores	Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz , 6 cores
Memory	16 GB	16 GB
OS	CentOS 6.3	CentOS 6.3
Disk type	SSD 1.8 TB	SSD 600 GB
Hadoop vr	0.20	0.20

表 3　データセットの統計

	期間	サイズ	リビジョン数	関連ファイル数
Eclipse	全期間†	165[MB]	264,898	210,503
	1 年	22.4[MB]	35,500	47,952
	半年	15.2[MB]	22,077	41,135
	1 ヶ月	1.83[MB]	3,366	7,808
	1 週間	284[KB]	646	1,191
Android	全期間†	1.58[GB]	1,766,981	567,014
	1 年	411[MB]	434,799	323,789
	半年	204[MB]	218,807	99,930
	1 ヶ月	36.5[MB]	39,539	21,522
	1 週間	15.0[MB]	16,479	3,916

† Eclipse は 2001〜2009，Android は 2005〜2011.

4.3　データセット

本研究で用いたリポジトリのデータは，MSR Mining Challenge 2011[*4]，2012[*5] により提供された Eclipse の CVS リポジトリと，Android の Git リポジトリのコミットログである．このデータセットの統計を表 3 に示す．

4.4　結果

num_changes_co-changed_files を，従来手法の逐次処理で求めた場合と，提案手法による並列分散処理で求めた場合の比較結果を表 4，表 5 に示す．全期間のデータの場合，Hadoop を用いることで，Eclipse のデータセットで 1.71 倍，Android のデータセットで 42.27 倍速く計算することができた．一方で，パース処理を含む全ての処理でデータセットが小さいとき（Eclipse では全期間より小さい場合，Android では 1 年未満の場合）従来手法で処理したほうが速いという結果が得られた．これは，Hadoop が並列分散処理を行うための前後処理や，データ通信をサーバ間で行うために，より多くの時間を費やしたためであると考える．実行時間と表 3 のデータサイズを比較すると，プロセスメトリクスの計算に対する Hadoop 利用の恩恵は，150MB 程度以上のデータを扱う場合に得られるものと考えられる．

実行時間の内訳を表 6，表 7 に示す．Step 1 と Step 2 では Map 処理と Recude 処理で複数の mapper，reducer が実行されたため，最も実行時間を要したスレーブの実行時間を示す．そのため，Map 処理と Recude 処理の和が表 4，表 5 の値と一致しない．

パース処理の内訳であるが，put はローカルディスクにあるデータセットを HDFS へ格納する時間を表す．データサイズが大きい場合，put の時間を含めても従来手法より Hadoop のほうが速く計算できた．

全ての場合において Step 1 の計算時間は Hadoop よりも従来手法のほうが速く計算できた．これは，Step 2 の計算量 $O(f * \overline{r} * \overline{f})$ と比べて，Step 1 の計算量 $O(f * \overline{r})$ が少なかったためである．計算量が少ないと，データが大きくても Hadoop の効果が得られないことがわかった．

表 7 の全期間において，従来手法による Step 2 の実行時間が 1 年の場合と比べて 1 万倍以上かかっている．Step 1 と Step 2 の計算量の差だけで考えると，これほどの差が生じるのは妥当ではない．これは扱うデータサイズが大きいために，Step 2 の 3 重ループ目でメモリ不足となりスワップ[*6]が頻発したためだと考えられる．

表 4　Hadoop と従来手法による実行時間 [sec] の比較-Eclipse-

Hadoop	パース処理	Step 1	Step 2	合計
全期間	8.723	14.182	14.828	**37.733**
1 年	6.150	13.219	14.031	**33.400**
半年	6.031	13.149	13.332	**32.512**
1 ヶ月	5.678	12.124	12.736	**30.538**
1 週間	5.585	12.030	12.331	**29.946**
従来手法	パース処理	Step 1	Step 2	合計
全期間	13.275	0.084	51.269	**64.628**
1 年	1.389	0.028	9.217	**10.634**
半年	0.801	0.024	6.400	**7.225**
1 ヶ月	0.150	0.012	0.104	**0.266**
1 週間	0.075	0.004	0.024	**0.103**

表 5　Hadoop と従来手法による実行時間 [sec] の比較-Android-

Hadoop	パース処理	Step 1	Step 2	合計
全期間	30.302	17.994	15.237	**65.533**
1 年	13.452	14.128	14.501	**42.081**
半年	9.074	13.185	14.036	**36.295**
1 ヶ月	6.492	13.119	13.528	**33.139**
1 週間	5.916	12.219	13.138	**31.273**
従来手法	パース処理	Step 1	Step 2	合計
全期間	160.745	0.180	2524.742	**2685.667**
1 年	32.352	0.114	1710.438	**1742.904**
半年	15.183	0.047	38.332	**53.562**
1 ヶ月	0.885	0.031	0.840	**1.756**
1 週間	0.449	0.008	0.073	**0.530**

5.　おわりに

本稿では，リポジトリマイニングの高速化を目指して、リポジトリマイニングに対する Hadoop の性能評価を行った．ケーススタディの結果，Hadoop をリポジトリマイニングに適用することにより，プロセスメトリクスの 1 つである num_changes_co-changed_files を求める処理を，Eclipse のデータセットで 1.71 倍，Android のデータセットで 42.27 倍速く計算することができた．

*4　http://2011.msrconf.org/msr-challenge.html
*5　http://2012.msrconf.org/challenge.php
*6　メモリ不足を解消するために，現在使われていないメモリの内容をハードディスク上に書き出し，および読み込みをすること

表 6 メトリクス計算の実行時間 [sec] の内訳-Eclipse-

Hadoop	パース処理			Step 1				Step 2				合計
	put	Map 処理		Map 処理		Reduce 処理		Map 処理		Reduce 処理		
		time†	task††	time†	task††	time†	task††	time†	task††	time†	task††	
全期間	2.057	6.666	3	3	3	9	1	4	3	8	14	**37.733**
1 年	0.824	5.326	1	2	1	8	1	2	1	8	14	**33.400**
半年	0.797	5.234	1	1	1	8	1	2	1	8	14	**32.512**
1 ヶ月	0.646	5.032	1	1	1	8	1	1	1	8	14	**30.538**
1 週間	0.645	4.940	1	1	1	8	1	1	1	8	14	**29.946**

従来手法	パース処理	Step 1	Step 2	合計
全期間	13.275	0.084	51.269	**64.533**
1 年	1.389	0.028	9.217	**10.634**
半年	0.801	0.024	6.400	**6.424**
1 ヶ月	0.150	0.012	0.104	**0.266**
1 週間	0.075	0.004	0.024	**0.103**

† mapper，reducer の実行時間の最大値

†† mapper，reducer の生成された数

表 7 メトリクス計算の実行時間 [sec] の内訳-Android-

Hadoop	パース処理			Step 1				Step 2				合計
	put	Map 処理		Map 処理		Reduce 処理		Map 処理		Reduce 処理		
		time	task	time	task	time	task	time	task	time	task	
全期間	18.248	12.054	26	3	26	12	1	5	26	8	14	**65.533**
1 年	4.507	8.945	7	3	7	9	1	3	7	8	14	**42.081**
半年	2.532	6.542	4	2	4	8	1	2	4	8	14	**36.295**
1 ヶ月	0.968	5.524	1	1	1	8	1	2	1	8	14	**33.139**
1 週間	0.791	5.125	1	1	1	8	1	1	1	8	14	**31.273**

従来手法	パース処理	Step 1	Step 2	合計
全期間	160.745	0.180	2524.742	**2685.667**
1 年	32.352	0.114	1710.438	**1742.904**
半年	15.183	0.047	38.332	**53.562**
1 ヶ月	0.885	0.031	0.840	**1.756**
1 週間	0.449	0.008	0.073	**0.530**

今後の課題としては以下のことが考えられる．

- より大きなデータセットによる性能評価
  本稿で用いたデータセットは数ギガバイトのサイズであった．今後，Hadoop の特性をより明確にするために，数百ギガバイトやテラバイトスケールの実データによる性能評価を行いたい．
- 他の Hadoop ライブラリの利用
  例えば，Mahout [11] はオープンソースの機械学習ライブラリで，Hadoop と組み合わせることで分散処理環境における機械学習が可能になる．他の Hadoop ライブラリを用いることで，機能面，性能面で充実したリポジトリマイニングの分散処理環境を構築することも今後の課題である．

謝辞: 本研究の一部は，JST CREST「ポストペタスケール時代のスーパーコンピューティング向けソフトウェア開発環境」による助成，及び，JSPS 科研費（若手 A：課題番号 24680003）による助成を受けた．

参考文献

[1] Hassan, A. E.: The road ahead for mining software repositories. (2008).
[2] : Apache Hadoop, http://hadoop.apache.org/.
[3] White, T.，玉川竜司，兼田聖士：Hadoop 第 2 版，オライリー・ジャパン.
[4] The R Foundation: R, http://www.r-project.org/.
[5] Machine Learning Group at University of Waikato: Weka, http://www.cs.waikato.ac.nz/ml/weka/.
[6] Mockus, A.: Amassing and indexing a large sample of version control systems: towards the census of public source code history, *Proc. Int'l Conf. on Mining Software Repositories (MSR'09)*, pp. 11–20 (2009).
[7] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B. and Hassan, A. E.: Revisiting common bug prediction findings using effort-aware models, *Proc. Int'l Conf. on Software Maintenance (ICSM'10)*, pp. 1–10 (2010).
[8] Nagappan, N. and Ball, T.: Use of relative code churn measures to predict system defect density, *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pp. 284–292 (2005).
[9] Moser, R., Pedrycz, W. and Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pp. 181–190 (2008).
[10] Graves, T. L., Karr, A. F., Marron, J. S. and Siy, H.: Predicting fault incidence using software change history, *IEEE Trans. Softw. Eng.*, Vol. 26, pp. 653–661.
[11] : Apache Mahout, http://mahout.apache.org/.