情報処理学会論文誌 Vol.0 No.0 1–8 (??? 1959)

リポジトリマイニングに対するHadoopの導入に向けた性能評価

大坂 陽[1,a)] 山下 一寛[1,b)] 亀井 靖高[1,c)] 鵜林 尚靖[1,d)]

受付日 xxxx年0月xx日, 採録日 xxxx年0月xx日

概要：本論文では，Hadoop を用いたスケールアウトによる並列分散処理と，従来手法による逐次処理との性能差を評価するために、実行環境の条件を変更しながら実験を行った．ケーススタディでは，リポジトリに蓄積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．対象期間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．サーバの台数の変化による実行時間の比較を行った結果，サーバの台数が 2 台でも Hadoop の方が従来手法よりも高速に計算できた．一方，前回のコミットまでの実験結果を再利用した場合の実行時間の比較を行った結果，Hadoop を用いるよりも従来手法を用いるほうが高速に処理できることがわかった．

Evaluating to apply the Hadoop for Large-scale Mining Software Repositories Studies

Osaka Ataru[1,a)] Yamashita Kazuhiro[1,b)] Kamei Yasutaka[1,c)] Ubayashi Naoyasu[1,d)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: In this paper, we perform an experiment to compare processing speed on Hadoop (i.e., parallel distributed processing) with on CPU (i.e., serial processing) by calculating software metrics from the commit log data of the Eclipse project and the Android project. From the experiment, we find that the Hadoop approach is up to a factor of 1.71 faster than the CPU approach from the Eclipse project and up to a factor of 42.27 faster from the Android project. The result of comparing the number of servers shows that the Hadoop approach outperforms the CPU approach. When we make use of the cached data, the CPU approach is faster than the Hadoop approach.

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]．リポジトリには年々データが蓄積され，データが増え続けている [2]．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．

本論文では，リポジトリマイニング分野に対して Hadoop [3][4] を用いたスケールアウト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 [4]．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，性能評価が十分に行われていない．たとえば，どの程度のリポジトリの環境に Hadoop を適用すると良いか，という観点

[1] 九州大学 Kyushu Uniersity
[a)] osaka@posl.ait.kyushu-u.ac.jp
[b)] yamashita@posl.ait.kyushu-u.ac.jp
[c)] kamei@ait.kyushu-u.ac.jp
[d)] ubayashi@ait.kyushu-u.ac.jp

での評価は具体的にはまだ行われていない．

そこで本論文では，Hadoop を用いた並列分散処理と，従来手法である逐次処理（サーバ 1 台で実行したシングルスレッドによる逐次処理）との性能差を評価するために，実行環境の条件を変更しながら実験を行った．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．本論文では，以下の 3 つのリサーチクエスチョンに答えることを目的とする．

RQ1:データセットの規模にかかわらず Hadoop を利用する効果はあるのか データセットの規模が大きいときは Hadoop を利用する効果はあるが，データセットの規模が小さいときには Hadoop を利用する効果は小さいということが一般的に知られている．メトリクスの計算においても同様の結果は容易に推測できるが，その推測を確認するために，データセットの対象期間を変化させながら，Hadoop と従来手法の比較実験を行った．データセットの規模が 150MB より大きい場合 Hadoop の方が高速に計算できた．例えば，全期間のログデータを用いた場合（150MB 以上），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．

RQ2:従来手法よりも高速に処理するためには，どの程度のサーバの台数が必要であるのか メトリクスの計算にサーバの台数がどの程度影響を及ぼすのか，Android の全期間のログデータにおいて，サーバの台数を変化させながら実験を行った結果，より台数が多い方が好ましいが，2 台（マスターサーバ，スレーブサーバ 1 台ずつ）でも Hadoop の方が従来手法より高速に計算できた．

RQ3:前回の実験結果を再利用できる場合でも Hadoop の効果はあるのか 全期間のデータセットを対象としてプロセスメトリクスを一から計算するのではなく，開発者が前回のコミットまでのプロセスメトリクスの計算の結果を保持していて，その結果を再利用してプロセスメトリクスの計算を行う場合を想定する．このような場合でも Hadoop の有用性があるのか，Android の全期間のログデータにおいて従来手法との比較実験を行った結果，Hadoop を用いるよりも従来手法を用いるほうが高速に処理できることがわかった．

2. 背景と関連研究

2.1 リポジトリマイニング

リポジトリマイニングは，リポジトリに蓄積されてい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 [5] や Weka [6] 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 [7]．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．そこで本論文では，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，工程 1 のデータ準備について高速化を目指す．

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 [4]．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 [3]，分散ファイルシステムである HDFS(Hadoop Distributed File System) と，並列分散処理フレームワークである MapReduce が Hadoop の主要プロジェクトである．

MapReduce の入出力は HDFS で行われ，[key/value] のペアのデータ集合が扱われる．MapReduce は Map 処理と Reduce 処理の 2 つの処理に分けられる．Map 処理の入力は HDFS に格納されているファイルである．Map 処理の実行クラスである 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．mapper の数は入力ファイルのファイルサイズやファイル数に応じて Hadoop によって決められる．

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．この Reduce 処理の出力が MapReduce の出力となる．reducer の数だけファイルが生成され，HDFS に格納される．

2.3 関連研究

データ処理の性能向上のためのアプローチとしては 2 つ

ある．サーバの台数を増やして分散処理による処理の高速化を図るスケールアウトと，サーバの性能を向上して処理の高速化を図るスケールアップである．

スケールアウトによるアプローチ 本論文と同様にリポジトリマイニングに対して Hadoop を適用する研究を Shang らが行っている [2][8]．文献 [8] では，リポジトリに保管されている全変更履歴からファイルごとの変更履歴の抽出に MapReduce を適用し，リポジトリの種類，サーバの台数による実行時間の比較を行っている．文献 [2] では，Hadoop ライブラリである Pig の性能評価として MapReduce，逐次処理との比較を行っている．Pig とは，データ処理言語で書いたプログラムを MapReduce プログラムに変換して実行するプラットフォームである．Pig を使うことで MapReduce の知識がなくとも，Hadoop の分散処理を実行することができる．Pig と MapReduce では，MapReduce の方が若干高速であるが，Pig と逐次処理では，Pig の方がはるかに高速であることを示している．本論文では，リポジトリマイニングへの適用実験としてメトリクスの計算を扱い，比較対象としてサーバの台数，リポジトリの種類に加え，データサイズの期間（1 週間，1 ヶ月，半年，1 年，全期間）による実行時間の比較を行い，その結果から Hadoop を用いるほうが望ましいケースとそうでないケースについて考察した．

スケールアップによるアプローチ 我々はこれまでもリポジトリマイニングの処理の高速化を目指して研究を行ってきており，GPGPU を用いたスケールアップによる高速化処理の性能評価を行った [9]．GPGPU とはコンピュータシステムにおいて画像処理を担う GPU を使って，大量の計算を並列処理する技術のことである．ケーススタディとして，Eclipse プロジェクトのバージョン履歴上における代表的なリポジトリマイニングを行った結果，CPU のみを用いた手法に比べ 43.9 倍高速に計算できた．これまでは複雑な計算量に対して GPGPU を適用することで，計算空間の拡張を行うスケールアップのアプローチによって処理の高速化を行った．しかしながら，リポジトリマイニング分野ではリポジトリには年々データが蓄積されデータサイズが増え続けているため，データ空間の拡張を行うスケールアウトのアプローチも必要である．本論文と文献 [9] との違いは，データ空間の拡張のアプローチとして Hadoop を用いたスケールアウトを使用した点である．

スケールアウトとスケールアップの関係 スケールアウト，スケールアップともに並列処理によって処理の高速化を図っているが，並列処理の適用範囲が異なる．スケールアウトはデータを分割し，複数台のサーバに各データを振り分け，並列処理を行っている．一方，スケールアップは計算処理を分割し，分割された各計算処理を並列に行っている．この様にスケールアウトとスケールアップの適用範囲が違うので，スケールアウトとスケールアップのアプローチを組み合わせることで，より高速な処理ができるのではないかと考える．

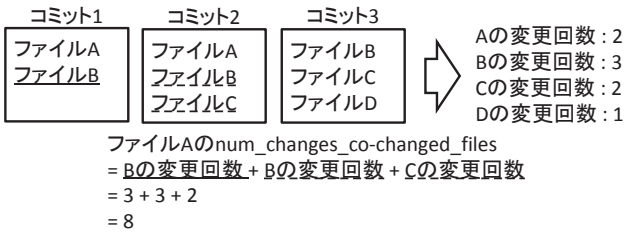

図 1 num_changes_co-changed_files の計算方法
Fig. 1 Calculation of num_changes_co-changed_files

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

本論文では，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 [10][11][12][13] に用いるためのメトリクス（ソースコード行数などの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 [10][11][12][13] 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，本論文ではプロセスメトリクスの算出を対象とする．

表 1 に不具合モジュール予測に用いられる主なプロセスメトリクスを示す．本論文では，num_changes_co-changed_files（以降，NCCF）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）で実行した場合とで，実行速度にどの程度の違いがあるかを比較する．NCCF 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．NCCF 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．

NCCF の計算方法を，図 1 を用いて説明する．例えば，ファイル A の NCCF を求める．コミット 1 ではファイル B が，コミット 2 ではファイル B とファイル C が，ファイル A と同時に変更されている．したがって，ファイル A のコミット 3 終了時点の NCCF = 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．

この様に，NCCF 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

表 1 不具合モジュール予測に用いられる主なプロセスメトリクス
Table 1 List of major process metrics using bug prediction

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

† f はファイル数，$\bar{c}$ は 1 ファイルにおける平均コミット数，$\overline{f}$ は 1 コミットにおける平均ファイル数を表す．

変換．本論文では，入力となるログデータとして，git の log コマンド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

表 1 の NCCF の計算量 $O(f*\bar{c}+f*\bar{c}*\overline{f})$ の $f*\bar{c}$ の部分に該当

Step 2. NCCF を算出

表 1 の NCCF の計算量 $O(f*\bar{c}+f*\bar{c}*\overline{f})$ の $f*\bar{c}*\overline{f}$ の部分に該当

$O(f*\bar{c}+f*\bar{c}*\overline{f})$ の f はファイル数，$\bar{c}$ は 1 ファイルにおける平均コミット数，$\overline{f}$ は 1 コミットにおける平均ファイル数を表す．

3.2 従来手法による実装

3.2.1 パース処理

ログデータは，1 行ごとに各コミット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，変更されたファイルのリスト $FILES$，コミット ID のリスト $COMMITS$，key にコミット ID（変更履歴を識別する ID），value にコミットごとの変更されたファイルのリストをもつハッシュ $COMMIT$ 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コミット ID のリストをもつハッシュ $COCHANGED$ を抽出する．

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 $COMMIT(file)$ 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 Step 1 の計算量は $O(f*\bar{c})$ となる．

Algorithm 1 Algorithm for Step1

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

次に，Step 2 のアルゴリズムを，Algorithm2 として示す．Algorithm2 は各ファイルの NCCF 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 $commit$ 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．このようにして NCCF が求まり，Step 2 の計算量は $O(f*\bar{c}*\overline{f})$ となる．

3.3 MapReduce による実装

3.3.1 パース処理

図 2 に MapReduce によるパース処理の実装を示す．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，[バイトオフセット[*1]/各行の内容]

[*1] ファイルの先頭からの各行の距離

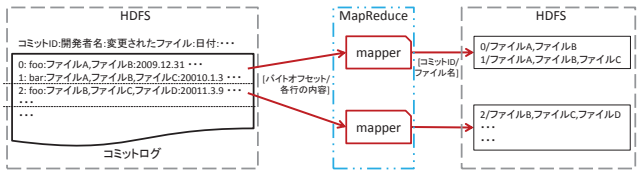

図 2　MapReduce によるパース処理の実装
Fig. 2　Parse process using the MapReduce

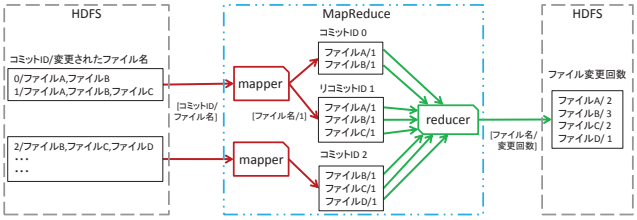

図 3　MapReduce による Step 1 の実装
Fig. 3　Step 1 process using the MapReduce

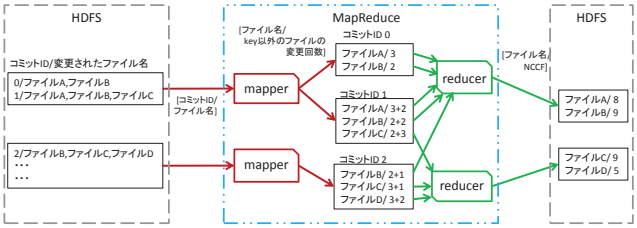

図 4　MapReduce による Step 2 の実装
Fig. 4　Step 2 process using the MapReduce

である．mapper では，value からコミット ID と変更されたファイル名を抽出し，[コミット ID/ファイル名のリスト] を出力する．mapper の出力の key について集約処理を行う必要はないため，reducer の数は 0 に設定する．

3.3.2　メトリクス算出

Step 1.　図 3 に MapReduce による各ファイルの変更回数の計算方法を示す．mapper への入力にはパース処理の出力結果 [コミット ID/ファイル名のリスト] を使用する．各コミットでのファイルの変更回数 1 を value として，mapper は [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（ファイル名）の value を加算することで，各ファイルの変更回数が求まり，reducer は [ファイル名/変更回数] を出力する．

Step 1 の出力結果は Step 2 で用いるため，出力ファイルを 1 つにして，Step 2 でファイルの I/O を少なくするようにしたい．そのため，reducer の数を 1 つに設定し，出力ファイルを 1 つに設定した．

Step 2.　図 4 に MapReduce による NCCF の計算方法を示す．NCCF は，同時に変更されたファイルの変更回数の総和である．処理の概要は，Map 処理でコミットごとの同時に変更されたファイルの変更回数の和を求め，Reduce 処理で Map 処理の出力を同じファイル名ごとに集約して，同じファイル名のコミットごとの同時に変更されたファイルの変更回数の和を加算して NCCF を求める．処理の内容を具体的に述べると，Step 2 の mapper の入力は Step 1 の mapper の入力と同じパース処理の出力結果 [コミット ID/ファイル名のリスト] を使用する．mapper ではコミットごとの同時に変更されたファイルの変更回数の和を求めるので，mapper の出力は [ファイル名/key 以外のファイルの変更回数の和] 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．

コミット 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．コミット 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 の変更回数を加算した結果を出力する．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+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（ファイル名）の value を加算し，同時に変更されたファイルの変更回数の総和を求める．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 は [ファイル名/NCCF] を出力する．

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 では分散処理を可能な限り行うことで高速化を図るために，本論文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．

表 2　実験に用いたサーバの環境
Table 2　Server environment used in the experiment

	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

4.　評価

4.1　評価方法

NCCF 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．従来手法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．

4.3　データセット

本研究で用いたデータセットの統計を表 3 に示す．本研

表 4 メトリクス計算の実行時間 [sec] の比較-Eclipse-
Table 4 Comparison of time to calculate the metrics -Eclipse-

Hadoop	合計	パース処理			Step 1				Step 2			
		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	**37.733**	2.057	6.666	3	3.496	3	9.993	1	4.178	3	9.989	14
1 年	**33.400**	0.824	5.326	1	3.217	1	9.357	1	3.748	1	9.674	14
半年	**32.512**	0.797	5.234	1	2.974	1	9.368	1	3.415	1	9.597	14
1 ヶ月	**30.538**	0.646	5.032	1	2.279	1	9.121	1	2.508	1	9.597	14
1 週間	**29.946**	0.645	4.940	1	2.304	1	9.074	1	2.493	1	9.421	14

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

† mapper，reducer の生成された数

表 5 メトリクス計算の実行時間 [sec] の比較-Android-
Table 5 Comparison of time to calculate the metrics -Android-

Hadoop	合計	パース処理			Step 1				Step 2			
		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	**65.533**	18.248	12.054	26	3.337	26	13.912	1	4.598	26	10.265	14
1 年	**42.081**	4.507	8.945	7	3.201	7	10.297	1	4.061	7	9.917	14
半年	**36.295**	2.532	6.542	4	3.015	4	9.668	1	3.472	4	9.867	14
1 ヶ月	**33.139**	0.968	5.524	1	2.842	1	9.359	1	3.436	1	9.655	14
1 週間	**31.273**	0.791	5.125	1	2.715	1	9.058	1	2.890	1	9.583	14

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

表 3 データセットの統計
Table 3 Statistics of the data set

	期間	サイズ	コミット数	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.

究で用いたリポジトリのデータは，MSR Mining Challenge 2011[*2]，2012[*3] 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．

[*2] http://2011.msrconf.org/msr-challenge.html
[*3] http://2012.msrconf.org/challenge.php

4.4 実験結果

RQ1:データセットの規模にかかわらず Hadoop を利用する効果はあるのか

データセットの規模が大きいときは Hadoop を利用する効果はあるが，データセットの規模が小さいときには Hadoop を利用する効果はないということが一般的に知られている．プロセスメトリクスの計算においても同様の結果が容易に推測できるが，その推測を確認するために，NCCF 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との比較を行った．その結果を表 4，表 5 に示す．結果は推測通りで，全期間のデータの場合，Hadoop を用いることで表 4 の Eclipse のデータセットで 1.71 (=64.533/37.733) 倍，表 5 の Android のデータセットで 42.27（=2685.667/65.533）倍速く計算することができた．一方で，データセットが小さいとき（Eclipse では全期間未満場合，Android では 1 年未満の場合）は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

表 6 スレーブサーバの台数による実行時間 [sec] の比較
Table 6 Comparison of time to calculate the metrics changed number of slave server

台数	パース処理	Step 1	Step 2	合計
0	68.357	45.192	115.297	**228.846**
1	69.490	43.718	102.278	**215.448**
3	41.260	23.416	42.374	**107.051**
5	33.085	18.399	28.782	**80.266**
7	30.302	17.994	15.237	**65.533**

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．

表 4，表 5 のパース処理の内訳において，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．

Step 1 の計算時間は，全ての場合において 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 $O(f * \overline{r})$ が小さかったためである．

表 5 の全期間において，従来手法による Step 2 の実行時間が Step1 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，スワップ[*4]が頻発したためだと考えられる．

まとめ 表 4，表 5 より，プロセスメトリクスの計算において，データセットの規模が十分に大きいとき（Eclipse では全期間の場合，Android では 1 年以上の場合）は，Hadoop を用いるほうが高速に計算できたが，データセットの規模が小さくなると，従来手法の方が高速に計算でき，Hadoop を利用する効果はないたことが確認できた．

RQ2:従来手法よりも高速に処理するためには，どの程度のサーバの台数が必要であるのか

スレーブサーバの台数を変えて，Android の全期間のログデータにおける実行時間の比較を行った．この実験では，スレーブサーバをそれぞれ 1，3，5，7 台と台数を変えて用いた場合と，1 台でマスターサーバ，スレーブサーバの役割を行う擬似分散と呼ばれる機能を用いた場合とで比較を行った．

表 6 に結果を示す．台数が 0 台の結果は，擬似分散の場合の結果である．スレーブサーバ 0 台と 1 台で結果に大きい違いがないのは，本質的には計算をマスターサーバ単体で行うか，スレーブサーバ単体で行うかの違いで，計算を行うサーバの台数は同じであるためであるだと考えられる．表 5 の従来手法の全期間の結果とスレーブサーバが 1 台の結果を比較すると，スレーブサーバが 1 台の方が高速に計算できた．これは，スレーブサーバ内で mapper，reducer が複数生成され並列処理が行われていたためだと考える．

[*4] 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

RQ3:前回の実験結果を再利用できる場合でも Hadoop の効果はあるのか

全期間の NCCF を一から計算するのではなく，開発者がリポジトリに変更をコミットする際，それまでのファイルの変更回数と NCCF の値が保持されているケースを想定した実験を行った．つまり，開発者がコミットにおける変更分だけ NCCF を再計算した場合を想定している．なお，この 1 コミットのときには 10 個のファイルが同時に変更されているものとする．

Android の全期間のログデータにおいて実験を行った結果，従来手法の場合は 6.591 秒で計算できるのに対して，Hadoop の場合は 14.476 秒かかり，Hadoop を利用する効果はないことがわかった．前回の NCCF の計算過程をうまくファイルに出力しておくことで，新しいコミットに対して多くの計算量が必要ではなくなったため，従来手法でも十分に高速に実行できたと考える．その一方で，Hadoop は実行の準備のためにある程度の時間を要するため，従来手法よりも遅くなってしまったと考える．

5. 考察

5.1 適用シナリオ

Hadoop が処理性能の向上を図ることができる有用な手段の 1 つであることは，4 章の結果から確認できた．しかしながら，全ての処理で Hadoop の方が高速に処理できたわけではなく，従来手法の方が高速に処理できる場合もあった．そこで本章では，Hadoop の効果的な適用方法についてシナリオを交えて考察する．

ケース 1：**プロジェクト管理者が開発中のプロジェクトにおいてメトリクス計算を行う場合**． 考察対象のプロジェクトでは，Git などの分散型の版管理システムを利用していると仮定する．例えば，Android のように約 6 年間のログデータを解析し，NCCF を計算する場合，メトリクス導入開始の時点では，再利用可能なデータも存在せず，一からメトリクスの計算を行う必要がある．この場合，表 4，表 5 のように，Hadoop に比べると従来手法は多くの時間を要するため，Hadoop を用いるほうが好ましい．

ケース 2：**プロジェクト管理者が特定期間のメトリクス計算を行う場合**． ケース 1 の後，プロジェクト管理者が特定期間のみの NCCF を計算したい場合が考えられる．例えば，約 1 年間のメトリクスを計算する場合，表 5 から考慮すると Hadoop を用いるほうが好ましい．しかし，Hadoop と従来手法のどちらを用いるほうが好ましいかはプロジェクトの規模に左右される．

ケース 3：**開発者が開発者自身のローカルリポジトリにおいてメトリクス計算を行う場合**． ケース 1 の後に，開発者が自身の行う変更を行う際，そのコミットの品質がどの程度であるかを知りたい場合があると考えられる．その際には，既にケース 1 の実行結果の再利用を行うことができ

るので，前回と今回の差分のみを求めるだけでよい．この場合，RQ3 の結果のように，従来手法のほうが，実行の前準備が必要でないため，高速に処理を終了できる．また，Hadoop のように，複数台のサーバが必要でないため，開発者が自身の開発環境でも実行することが可能である．

まとめ：Hadoop は処理性能の向上を図ることができる有用な手段の 1 つである．しかし，Hadoop はデータセットの規模が小さいと（Eclipse では全期間未満の場合，Android では 1 年未満の場合），十分な効果が得られない．そこで，前述のケース 1 からケース 3 のように利用者の開発環境を考慮して Hadoop の適用方法を選ぶ必要がある．ただし，Hadoop の利用には多少の専門知識を必要とする．そのため，導入の際に Hadoop の知識を持つ開発者が一人もいない場合は，その学習コストを考慮することが望ましい．

また，Hadoop は Java で実装されているため，Java のプラットフォームを使用することができる．そのため，Hadoop と従来手法を組み合わせて，それぞれの特徴を生かした実装を行うことができる．例えば，本論文で使用した NCCF を実装するときには，Hadoop によりパース処理と Step 2 を，従来手法により Step 1 を実装することでより高速に処理することができる．

5.2 NCCF 以外のメトリクスに対する Hadoop の効果

本研究では，3 章の表 1 の中から計算量が最も大きい NCCF を選んで評価実験を行った．NCCF の Step 1 の計算量は表 1 の計算量 $O(f * \bar{r})$ と，Step 2 の計算量は表 1 の計算量 $O(f * \bar{r} * \bar{f})$ と同じであるため，これらのメトリクスについても 4 章の表 4，表 5 に示した Step 1，Step 2 と同程度の計算時間で実行されると推測できる．したがって，表 1 の計算量 $O(f * \bar{r})$ で実行されるメトリクス（PD，PC，LCBR，AGE，MSF，MSP，MAF）は全ての場合において，表 1 の計算量 $O(f * \bar{r} * \bar{f})$ で実行されるメトリクス (NCF，SCF，MSC) はデータセットが小さい場合（Eclipse では全期間未満の場合，Android では 1 年未満の場合）において，Hadoop よりも従来手法を用いた方が高速に計算できると考える．なお，表 1 のメトリクスについて実際に実験することは今後の課題である．

6. おわりに

本論文では，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．ケーススタディとして，プロセスメトリクスの 1 つである NCCF を求める計算を行った．対象期間を変えながら実験を行った結果，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．サーバの台数の変化による実行時間の比較を行った結果，サーバ 2 台でも Hadoop の方が従来手法より高速に計算できた．前回の実験結果を再利用した場合の実行時間の比較を行った結果，Hadoop を用いるよりも従来手法を用いるほうが高速に処理できることがわかった．

今後の課題としては，GPGPU の評価実験を加えた高速化手法提案を考えている．本論文では，Hadoop の評価実験を行ったが，他にも高速化手法として GPGPU [14] を用いたスケールアップによる処理性能向上がある．GPGPU の評価実験を行い，Hadoop，GPGPU それぞれの長所を生かしたリポジトリマイニングへの適用方法を提案したい．

謝辞 本研究の一部は，科学技術振興事業団「JST」の戦略的基礎研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」の研究課題「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，日本学術振興会科学研究費補助金（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] Shang, W., Adams, B. and Hassan, A. E.: Using Pig as a data preparation language for large-scale mining software repositories studies: An experience report, *J. Syst. Softw.*, Vol. 85, No. 10, pp. 2195–2204 (online), DOI: 10.1016/j.jss.2011.07.034 (2012).
[3] Apache: Apache Hadoop, http://hadoop.apache.org/.
[4] White, T.，玉川竜司，兼田聖士：Hadoop 第 2 版，オライリー・ジャパン.
[5] The R Foundation: R, http://www.r-project.org/.
[6] Machine Learning Group at University of Waikato: Weka, http://www.cs.waikato.ac.nz/ml/weka/.
[7] 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).
[8] Shang, W., Jiang, Z. M., Adams, B. and Hassan, A. E.: MapReduce as a general framework to support research in Mining Software Repositories (MSR), pp. 21–30 (online), DOI: 10.1109/MSR.2009.5069477 (2009).
[9] 永野梨南，中村央記，亀井靖高，ブラムアダムス，久住憲嗣，鵜林尚靖，　福田晃：GPGPU を用いたリポジトリマイニングの高速化手法— プロセスメトリクスの算出への適用 —，　Vol. 2012, pp. 1–8 (2012).
[10] 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).
[11] 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).
[12] 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).
[13] 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.
[14] Manavski, S.: CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography, pp. 65–68 (online), DOI: 10.1109/ICSPC.2007.4728256 (2007).