社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

MapReduce処理系SSS上のSawzall処理系の実装

中田　秀基† 　井上　辰彦††,† 　小川　宏高† 　工藤　知宏†

† 独立行政法人 産業技術総合研究所
〒 305–8568 茨城県つくば市梅園 1 − 1 − 1 中央第二
†† 株式会社創夢
〒 151-0072 東京都渋谷区幡ヶ谷 1-34-14 宝ビル 9F
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp, ††tatuhiko@soum.co.jp

あらまし　Sawzall は、Google が 2006 年に発表した大容量データの並列バッチ処理に適した言語である。Sawzall の計算モデルは MapReduce 型の分散演算であるが、リダクション操作を組み込みの Aggregator に限定することで、エンドユーザによる容易な記述を可能にしている。われわれは Hadoop およびわれわれが現在開発中の MapReduce 処理システム SSS を対象として、Scala 言語による Sawzall 言語処理系 SawzallClone を、Java をターゲットコードとするコンパイラとして実装した。SSS の Native API で直接プログラムを記述した場合と比較を行い、言語オーバヘッドを見積もった。また、Google による逐次処理系 Szl との比較を行った。その結果、一定の言語オーバヘッドはあるもののメリットを考えれば許容範囲であること、Szl と比較して遜色ない逐次性能を示すことを確認した。
キーワード　Map Reduce, Sawzall, 分散並列計算, 言語

An Implementation of Sawzall on SSS: a MapReduce Runtime

Hidemoto NAKADA†, Tatsuhiko INOUE††,†, Hirotaka OGAWA†, and Tomohiro KUDOH†

† National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba Central 2, 1-1-1 Umezono, Tsukuba, Ibaraki 305-8568, JAPAN
†† SOUM Corporation, Takara blg. 9F, 1-34-14 Hatagaya, Shibuya-ku, Tokyo, 151-0072, JAPAN
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp, ††tatuhiko@soum.co.jp

Abstract　Sawzall is a script language designed for batch processing of large amount of data, which is introduced by Google in 2006. The processing model of Sawzall is the MapReduce. Sawzall allows programmers only to program 'mappers' to ease the burden. Sawzall provides a set of 'built-in aggregators', from which programmer choose reducing function. We implemented a Sawzall processing system called SawzallClone for Hadoop and SSS; our own implementation of MapReduce. SawzallClone compiles Sawzall code into Java code, which will be compiled into bytecode and works with runtime library on the MapReduce systems. In the paper, we provide detailed implementation of the system. We compared the system with Google's open source implemntation called 'Szl'. We also evaluated the language overhead coparing programs written in native SSS API. We confirmed that, the performance of SawzallClone is comparable with Szl, and while the system introduces certain language overhead, it is acceptable regarding the easiness provided by the layer.
Key words　Map Reduce, Sawzall, Distributed Parallel Execution, Language

1. はじめに

MapReduce [1] は並列プログラミングパラダイムのひとつである。Apache Hadoop [2] の普及によって、本来のターゲットであったログデータ解析などだけでなく、科学データの解析などにも広く用いられるようになった。

MapReduce での並列プログラミングは、従来のメッセージパッシングモデルや共有メモリモデルと比較すると容易であると言われるが、実はそれほど簡単ではない。たとえば Hadoop では、プログラマは 3 種類のプログラムを記述する必要がある。Mapper と Reducer および、それらを束ねコンフィギュレーションなどを設定するメインプログラムである。さらに、これ

らの 3 つのプログラムはプログラマの責任で相互に整合していなければならない。これはプログラミングを専門としない技術者にとっては容易なことではない。

これに対して、MapReduce プログラムを生成する上位言語を定義することで、プログラマの負荷を低減する動きがある。HiveQL [3] [4]、Apache Pig [5] [6]、Jaql [7] [8] などである。Google の *Sawzall* [9] もこの種の試みの一つである。Sawzall は Reducer を言語組み込み機能として提供することで、プログラマには Mapper のみを記述させる。記述モデルが MapReduce よりもさらに単純化され、一般エンジニアでも容易に記述できる言語となっている。Google は、オープンソースで処理系 Szl [10] を提供しているが、これは逐次処理に限定されている。

われわれは、この Sawzall の処理系 SawzallClone [11] を、Java をターゲット言語としたコンパイラとして Scala 言語を用いて実装した。SawzallClone は MapReduce 処理系に対して独立に設計されており、複数の MapReduce 処理系に対応することが可能である。現在、Hadoop および、われわれが開発中の MapReduce 処理系 SSS がサポートされている。本稿では、SSS 上での SawzallClone の実装と評価を示す。

本稿の構成は以下の通りである。2. 節で対象となる MapReduce 処理系 SSS を概説し。3. 節で Sawzall の言語仕様を概観する。4. 節で SawzallClone の設計と実装を述べる。5. 節で評価結果を示す。6. 節に、まとめと今後の課題を述べる。

2. MapReduce と SSS

2.1 MapReduce

MapReduce とは、入力キーバリューペアのリストを受け取り、出力キーバリューペアのリストを生成する分散計算モデルである。MapReduce の計算は、Map と Reduce という二つのユーザ定義関数からなる。これら 2 つの関数名は、Lisp の 2 つの高階関数からそれぞれ取られている。Map では大量の独立したデータに対する並列演算を、Reduce では Map の出力に対する集約演算を行う。

一般に、Map 関数は 1 個の入力キーバリューペアを取り、0 個以上の中間キーバリューペアを生成する。MapReduce のランタイムはこの中間キーバリューペアを中間キーごとにグルーピングし、Reduce 関数に引き渡す。このフェイズを**シャッフル**と呼ぶ。Reduce 関数は中間キーと、そのキーに関連付けられたバリューのリストを受け取り、0 個以上の結果キーバリューペアを出力する。各 Map 関数、Reduce 関数はそれぞれ独立しており、相互に依存関係がないため、同期なしに並列に実行することができ、分散環境での実行に適している。また、関数間の相互作用をプログラマが考慮する必要がないため、プログラミングも容易である。

MapReduce 処理系では、Mapper の後段に Combiner と呼ばれる処理を挿入することがある (図 1)。これは、Mapper と Reducer の間のデータ転送量を低減することを目的に、Mapper を実行したノード上でローカルに Reducer と同様のリダクション操作を行うものである。Mapper から出力されたキーバリューペアは、ローカルに蓄積され Combiner でリダクション処理され、量的に少量のキーバリューペアが生成される。これらのキーバリューペアが Reducer に送られ、グローバルなリダクション操作を受ける。

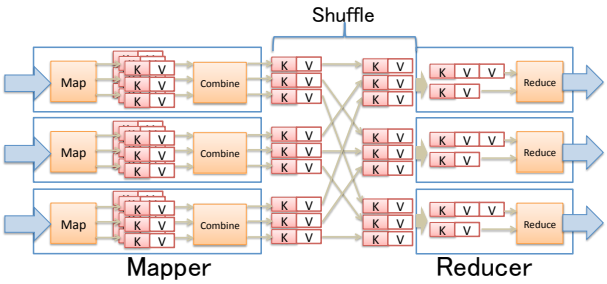

図 1　MapReduce の概要

Combiner が実行可能であるためには、Reducer の操作が結合法則を満たす必要があるが、実際に Reducer で行われる処理の多くがこの法則を満たすので、ほとんどの場合 Combiner を利用できると考えて良い。

2.2 SSS

SSS [12], [13] は、われわれが開発中の MapReduce 処理系である。SSS は Hadoop のようにファイルシステムを基盤とせず、分散 KVS を基盤とする点に特徴がある。入力データは予めキーとバリューの形で分散 KVS にアップロードしておき、出力結果も分散 KVS からダウンロードする形となる。

SSS ではデータをキーに対するハッシュで分散した上で、Owner Compute ルールにしたがって計算を行う。つまり、各ノード上の Mapper/Reducer は自ノード内のキーバリューペアのみを対象として処理を行う。これは、データ転送の時間を削減するとともに、ネットワークの衝突を防ぐためである。

Mapper は、生成したキーバリューペアを、そのキーでハッシングして、保持担当ノードを決定し直接書き込む。書きこまれたデータは各ノード上の KVS によって自動的にキー毎にグループ分けされる。これをそのまま利用して、Reduce を行う。つまり SSS においては、シャッフルは、キーのハッシングと KVS によるキー毎のグループ分けで実現されることになる。

SSS のもうひとつの特徴は、Map と Reduce を自由に組み合わせた繰り返し計算が容易にできることである。前述のように、SSS では Map と Reduce の間でやりとりされるデータも KVS に蓄積されるため、Map と Reduce が 1 対 1 に対応している必要がない。したがって、任意個数、段数の Map と Reduce から構成される、より柔軟なデータフロー構造を対象とすることができる。

3. Sawzall

Sawzall [9] は、Google が内部的に利用しているデータ処理用の言語で、Google の MapReduce 実装上で稼働している。Sawzall は、MapReduce モデルの map 部と reduce 部のうち、reduce 部を Sawzall が提供する固定的な Aggregator で提供し、ユーザには map 部分のみを記述させる。ユーザが記述する範囲を限定することによって、非技術者のユーザでもさらに容易

に並列プログラミングを行うことを可能にしている。

Sawzall は型付きの言語である。Sawzall コードはマップの入力データ 1 レコードに対して処理を行う。複数のレコードにまたがった処理を記述することはできない。レコード単位で処理するという点は、テキストファイルを 1 行ずつ処理する Awk 言語と類似している。ただし、Awk では行単位の処理を行いながら内部状態を更新することで、文書全体に対する処理を行うことができる。これに対して、Sawzall では言語処理系内部の状態はレコードごとにリセットされる。文書全体に対する総計処理は Aggregator 部分で行われる。

3.1 Protocol Buffers との連携

Sawzall は Protocol Buffers [14] と呼ばれるデータ表現形式でマーシャリングされたデータ構造の処理を前提としている。Protocol Buffers は Google が内部で広範に用いている、言語非依存の構造データのバイナリ表現形式である。XDR [15] と目的は類似しているが、データ量の削減に主眼を置いている点に特徴がある。Protocol Buffers では、データ構造の定義を言語非依存の IDL で記述する。このファイルを proto ファイルと呼ぶ。

Sawzall には、Protocol Buffers 用に定義された proto ファイルを取り込む構文として `proto` 文が用意されている。

```
proto "p4stat.proto"
```

proto ファイルを取り込むと、proto ファイル内の定義された構造体に相応する構造体が Sawzall 内で定義されると同時に、構造体とバイト列の間の自動変換（マーシャリングとアンマーシャリング）が定義される。

3.2 変数 input

Sawzall プログラムは入力を特殊な変数 `input` から受け取る。これは awk プログラムが変数`$0` に処理の対象となる行を受け取るのと類似している。

`input` の型は byte 列であるが、これを適当なデータ型に型変換して利用する。一般には Protocol Buffers で定義された型を用いる場合が多い。ただし、Sawzall では型変換は暗黙裡に行われるので、単なる代入の形をとる。

```
log: P4ChangelistStats = input;
```

3.3 テーブルと emit

Aggregator は、Sawzall の言語要素としてはテーブルという特殊な型として表現され、キーワード `table` を用いて定義される。

```
submitsthroughweek: table sum[minute: int] of count: int;
```

ここで、`sum` は予め定義されたテーブルの型であり、出力された結果を積算していくタイプのテーブルを示している。ここでは、`int` 型をおさめる `sum` 型テーブルの配列を定義している。

このテーブルに対して値を出力するのが `emit` 文である。出力された値は Aggregator に送出され積算される。

表 1　Sawzall の代表的なテーブル

テーブル名	意味
collection	emit されたすべての要素を含む集合を作る
maximum	値の大きい順に指定された要素数を保持
sample	指定された要素数を統計的にサンプリング
sum	emit された要素をすべて積算
top	頻度の高いものを指定された要素数保持。統計的処理
quantile	出力された値を、指定した数で分位する数を統計的に求める
unique	重複を排除した要素数を推定

```
1 proto "p4stat.proto"
2 submitsthroughweek: table sum[minute: int] of count: int;
3
4 log: P4ChangelistStats = input;
5
6 t:      time = log.time; # microseconds
7 minute: int  = minuteof(t)+60*(hourof(t)+24*(dayofweek(t)-1))
8
9 emit submitsthroughweek[minute] <- 1;
```

図 2　Sawzall プログラムの例

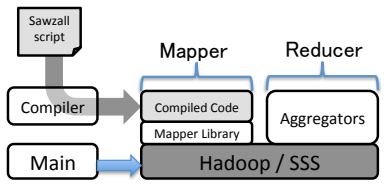

図 3　SawzallClone の概要

```
emit submitsthroughweek[minute] <- 1;
```

表 3.3 に、Sawzall の代表的なテーブルを示す。

3.4 Sawzall のプログラム例

図 2 に、文献 [9] より抜粋した Sawzall によるログ解析プログラムの例を示す。このプログラムは、Protocol Buffers 形式で格納されたログデータの頻度を分単位で集計するものである。

4. SawzallClone の実装

4.1 SawzallClone の概要

SawzallClone は、1) Sawzall コンパイラ、2) Mapper ライブラリ、3) Aggregator、4) メイン管理モジュールの 4 つのモジュールから構成される (図 3)。Sawzall コンパイラは、Sawzall のソースコードを読み込み、Java のソースコードを生成。さらにそれを `javac` でコンパイルしてバイトコードを生成する。Mapper ライブラリは、コンパイルされたバイトコードが利用するライブラリで、この両者が SSS の Mapper として稼働する。Aggregator は、SSS の Reducer として実装されている。

メイン管理モジュールは、SSS のクライアントプログラムとして機能する。Sawzall コンパイラを起動し、コンパイルされたコード、Mapper ライブラリ、Aggregator を一つの jar にまとめて、SSS に投入する。

4.2 Protocol Buffers スキーマの取り込み

3.1 で述べたように、Sawzall では Protocol Buffers のスキーマファイルを処理できなければならない。スキーマ情報

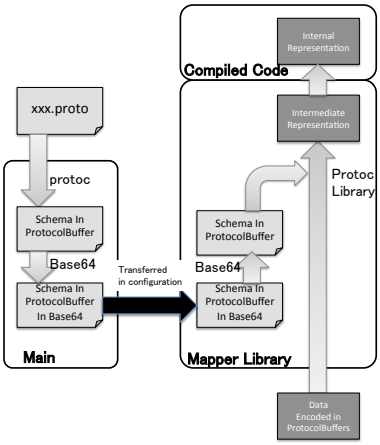

図 4　Including Protocol Buffers schema

```
public class Mapper implements SCHelpers.Mapper {
...
  @Override
  public void map(SCHelpers.Emitter emitter,
                  Helpers.ByteStringWrapper global_0_input)
  throws java.lang.Throwable {
    String local_0_document = BuildIn.func_string(global_0_input);
    List<String> local_1_words =
                            BuildIn.func_split(local_0_document);
    {
      Long local_2_i = 0l;
      for (; (((((local_2_i) <
                 (BuildIn.func_len(local_1_words)))?1l:0l)) != 0l);
                 (local_2_i) = (((local_2_i) + (1l)))) {
        emitter.emit(statics.static_0_t,
                     BuildIn.func_bytes(
                      (local_1_words).get((local_2_i).intValue())),
                     BuildIn.func_bytes(1l));
      }
    }
  }
}
```

図 5　コンパイル中間 Java コード

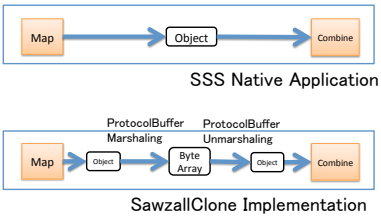

図 6　Mapper Combiner 間の通信

は Mapper が利用するため、何らかの方法でスキーマ情報を Mapper に転送しなければならない。proto ファイルそのものを転送してもよいが、proto ファイルの解釈には時間がかかるため、すべての Mapper が解釈を行うような方法は、好ましくない。

われわれは Google の Protocol Buffers 実装を利用し、スキーマ情報自体を Protocol Buffers 形式にする方法を用いた (図 4)。Sawzall コードのコンパイル時に、コンパイラは `proto` 文を見つけると、`protoc` を指定されたファイルに対して実行し、スキーマ定義を Protocol Buffers 形式で得る。このスキーマ定義は base64 形式にフォーマットされ、各 Mapper の設定情報の一部として Mapper ライブラリに転送される。Mapper ライブラリは、base64 形式をデコードしてスキーマ情報を取得する。このスキーマ情報と、protoc 実装に含まれるライブラリを用いて、バイト列を Protocol Buffers 実装固有のデータ構造に変換し、さらに SawzallClone 内部表現に変換する。

4.3　コンパイル

SawzallClone はバイトコードを直接生成せず、Java のソースコードを経由する。これは実装を容易にするためである。もう一つの理由としては、一般に Sawzall プログラムは一般にそれほど大きくないので余分に追加される Java コンパイルフェイズは十分無視できることが挙げられる。

図 5 に、図 9 に示したワードカウントのコードをコンパイルした結果生成されたコードの一部を示す。このコードは、SSS の Mapper を直接実装せずに、`SCHelpers.Mapper` インターフェイスを実装している。Mapper ライブラリに含まれるこのインターフェイスの実装によって、MapReduce システム間の相違を抽象化し吸収している。コンパイラは、ターゲットが SSS でも Hadoop でも全く同じコードを出力する。

4.4　Mapper - Combiner 間のバイト列化

SawzallClone では、実装の容易化のため、Mapper Reducer 間で受け渡されるオブジェクトを SawzallClone のレイヤでバイト列に変換し、SSS 上では単なるバイト列として扱う。したがって、Mapper は常にバイト列を出力することになる。

通常はこれは大きなオーバヘッドの原因にはならない。しかし Combiner を利用する場合には問題となる。Mapper と Combiner は同一の Java VM 内で稼働しているため、本来これらの間ではオブジェクトのバイト列化は不要である。にもかかわらず SawzallClone の実装では Mapper が常にバイト列化を行うためマーシャリングとアンマーシャリングが発生してしまう (図 6) 。

5.　評　　価

本節では、SawzallClone をコンパイル時間、Szl との比較、SSS のネイティブ API で記述した場合との比較で評価する。

評価には、17 ノード（うち 1 ノードがマスター）から構成される小規模なクラスタを用いた。各ノードは、Intel Xeon W5590 を 2 ソケット（8core)、48 ギガのメモリ、SAS HDD 147GB を持ち、10Gb ネットワークで相互接続されている。

5.1　コンパイル時間

SawzallClone では、Sawzall で記述されたプログラムは Java にコンパイルされ、さらに Java バイトコードへとコンパイルされる。図 9 に示す簡単なスクリプトをコンパイルするのにかかる時間を計測した。

コンパイル時間は 1.2 秒程度であった。これには Javac によるバイトコードへのコンパイル時間 0.7 秒もふくまれる。MapReduce プログラムの実行には通常数分から数十分かかることを考えると、Sawzall のコンパイル時間は無視できると言える。

5.2　Szl との比較

Google による Sawzall のオープンソース実装である Szl と比較した。Szl は現在並列実装をサポートしていないため、比較は SawzallClone の逐次実行モードで行った。逐次実行モー

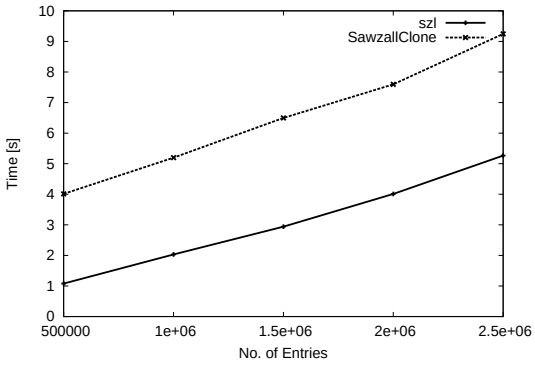

図 7　Szl との比較

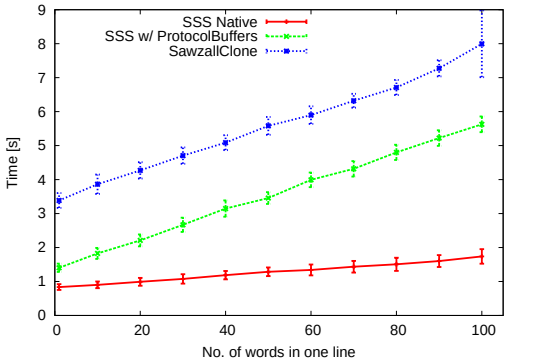

図 8　逐次実行による SSS ネイティブプログラムとの比較

ドでは、SSS も Hadoop も用いず、直接実行を単一 VM 内で、Mapper と Reducer を実行する。比較のためのアプリケーションとしては、文献 [9] に紹介されているログ解析プログラム (図 2) を用いた。解析対象のデータとして、50 万, 100 万, 150 万, 200 万, 250 万レコードのログを Protocol Buffers 形式で用意した。

結果を図 7 に示す。Y 軸は実行時間である。SawzallClone は Szl と比較して低速であるが、ログレコードを増やした際の実行時間の増大の傾きはほぼ同じである。つまり、SawzallClone は起動コストが定数時間だけ Szl よりも大きいだけであり、ログレコードあたりの処理速度はほぼ同じであることが分かる。

これは、Szl が内部バイトコードへの直接変換を行うのに対して、SawzallClone が Java コードを介した 2 段階のコンパイルを行うためであると考えられる。

5.3　SSS ネイティブアプリケーションとの比較

SawzallClone の言語レイヤによるオーバヘッドを知るために、同一のアプリケーションを Sawzall と SSS のネイティブ API で記述し、比較した。アプリケーションとしては、ワードカウントを用いた。図 9 に、Sawzall 版のワードカウントプログラムを示す。SSS ネイティブ版のプログラムは省略するが 60 行程度であった。

入力は 10 万行からなるランダムに生成した文書で、各行の単語数を 1 から 100 まで変化させた。Mapper は各行に対して起動され、Mapper 内部で単語の回数だけループを実行する。単語数を変化させることで Mapper 内のループ処理の速度を見積もることができる。実行は 1 ノードのみを用いておこなった。

```
document: string = input;
words: array of string  = split(document);
t: table sum[string] of int;
for (i: int = 0;i < len(words);i = i + 1) {
    emit t[words[i]] <- 1;
}
```

図 9　Sawzall によるワードカウントの実装

図 8 に結果を示す。X 軸は各行の単語数を、Y 軸は実行時間を示す。実験はそれぞれ 10 回行い平均値と標準偏差をエラーバーとしてプロットしている。SawzallClone は SSS ネイティブと比較して、単語数 0 の時の所要時間が大きく、さらに、単語数増大の際の所要時間増加の傾きも大きい。

まず、単語数 0 の時の所要時間の差の多くは、コンパイル時間に起因するものである。SawzallClone は 2 段階でコンパイルを行うため 1 秒あまりの定数オーダのオーバヘッドがある。一方 SSS ネイティブの場合は、ソースコードのコンパイル時間は計測に含まれていない。

次に、所要時間増加の傾きが大きい理由としては、4.4 節で述べた、Mapper-Combiner 間で受け渡される中間データを Protocol Buffers によってバイト列にマーシャリングしているオーバヘッドが考えられる。

この仮説を検証するために、SSS のネイティブ API を利用しつつ Mapper-Combiner 間のデータを SawzallClone 同様にマーシャリングする版を作成し、比較した (図 8)。単語数増大による実行時間増加の傾きは SawzallClone とほぼ同じであることから、Protocol Buffers による中間データのマーシャリングコストが単語数増大時の差の大半を占めることが確認できた。

5.4　並列実行時の SSS ネイティブアプリケーションとの比較

並列実行時の挙動に関して評価を行った。評価には図 2 に示したログ解析プログラムを用い、SawzallClone による実行と SSS ネイティブアプリケーションとを比較した。入力データはいずれの場合も Protocol Buffers でフォーマットされている。実験は 1 0 回行い平均を取った。

図 10 に実行時間を示す。エラーバーは標準偏差を示している。同じデータを、SSS ネイティブの 1 ノードの場合の実行時間に対する相対速度としてプロットしたものを図 11 に示す。

SSS ネイティブと比較して SawzallClone はやや低速となっている。これは逐次での場合と同様に、Protocol Buffers を用いた中間データの余分なマーシャリングによるオーバヘッドであると思われる。

図 11 を見ると、SSS ネイティブではリニア以上の速度向上を見せている。これは、2 ノード以上の場合が速いのではなく、1 ノードの場合が特に遅いのだと思われる。今回の評価では問題のサイズを固定しノード数を増減したため、1 ノードの場合にはメモリやキャッシュへのプレッシャーが特に大きく、性能低下を起こしていると考えられる。

6.　おわりに

われわれは、Sawzall スクリプトを現在開発中の MapReduce

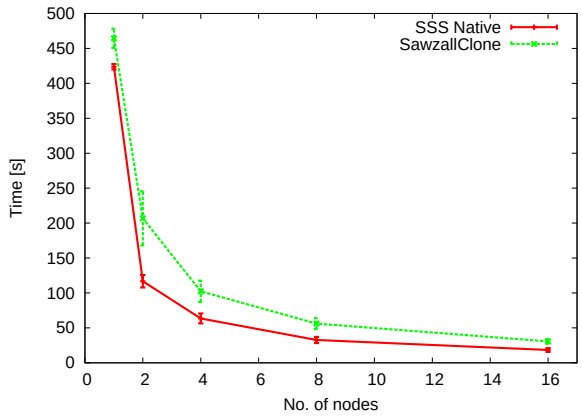

図 10　並列実行時の挙動

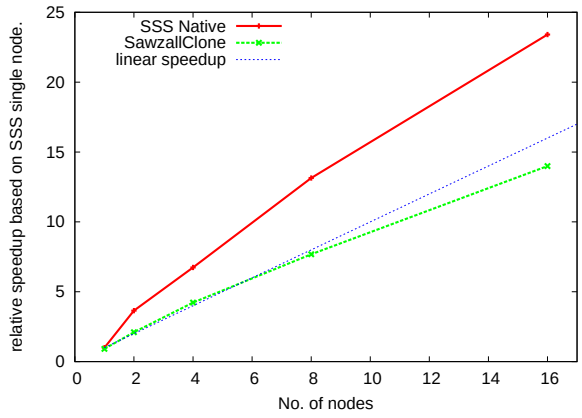

図 11　相対実行速度

処理系 SSS 上で動作させるための処理系 SawzallClone を実装した。SawzallClone は、Scala 言語で実装され Java 言語をターゲットとしたコンパイラおよび、SSS 上で動作するランタイムシステムとして実装されている。Google によるオープンソース Sawzall 処理系 Szl と比較したところ、2 段階のコンパイル時間をのぞけばほぼ同等の実行速度であることを確認した。さらに、SSS のネイティブ API で実装した場合と比較し、Sawzall 言語を用いたことによるオーバヘッドは明確に存在するものの、記述の容易性を勘案すれば十分許容可能であることを確認した。

SawzallClone の Hadoop 上で動作するバージョンはオープンソースソフトウェアとして公開されている [16]。SSS 上のバージョンも今後公開する予定である。

今後の課題としては以下が挙げられる:

- **Protocol Buffers によるマーシャリングオーバヘッドの除去**

オーバヘッドの多くの部分が、Mapper-Combiner 間で行われる不要なマーシャリングに起因すると思われる。実装を改善し、オーバヘッドの除去を行う。

- **Aggregator 実装の改良**

Sawzall の基本的なコンセプトは、大規模問題にスケールする実装が困難になりがちな Reducer 部分を、Aggregator という形で言語に取り込み専門家が記述することにある。しかし、現状の SawzallClone の Aggregator 実装は比較的ナイーブであり、大規模な問題にスケールしない。実用システムとして利用するには実装を改良する必要がある。

- **言語機能の拡張**

Sawzall は MapReduce の Mapper 部分のみの記述を許すことでプログラミングの敷居を下げているが、同時に記述できる問題のクラスを限定してしまっている。たとえば複数回の MapReduce を繰り返して解くようなプログラムを記述することはできない。言語機能を拡張し、応用範囲を広げることが課題である。

謝　　辞

本研究の一部は，独立行政法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務「グリーンネットワーク・システム技術研究開発プロジェクト（グリーン IT プロジェクト）」の成果を活用している．

文　　献

[1] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (2004).
[2] Hadoop, `http://hadoop.apache.org/`.
[3] Hive, `http://hive.apache.org/`.
[4] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. and Murthy, R.: Hive - A Petabyte Scale Data Warehouse Using Hadoop, *ICDE 2010: 26th IEEE International Conference on Data Engineering* (2010).
[5] Pig, `http://pig.apache.org/`.
[6] Olston, C., Chopra, S. and Srivastava, U.: Generating Example Data for Dataflow Programs, *SIGMOD 2009* (2009).
[7] jaql: Query Language for JavaScript(r) Object Notation, `http://code.google.com/p/jaql/`.
[8] Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J. and McPherson, J.: Ricardo: Integrating R and Hadoop (2010).
[9] Pike, R., Dorward, S., Griesemer, R. and Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall, *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, Vol. 13, No. 4, pp. 227–298 (2005).
[10] Szl - A compiler and Runtime for the Sawzall Language, `http://code.google.com/p/szl/`.
[11] 中田秀基, 井上辰彦, 工藤知宏: Hadoop 上で動作する Sawzall サブセットの実装, 情報処理学会プログラミング研究会 発表資料 2010-4-(1): (2011).
[12] Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-value Store based MapReduce Framework, *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (Accepted as a paper for First International Workshop on Theory and Practice of MapReduce (MAPRED'2010))*, pp. 754–761 (2010).
[13] 小川宏高, 中田秀基, 工藤知宏: 合成ベンチマークによる MapReduce 処理系 SSS の性能評価, 情報処理学会研究報告　2011-HPC-130 (to appear) (2011).
[14] Protocol Buffers - Google's data interchange format, `http://code.google.com/p/protobuf/`.
[15] Sun Microsystems, I.: XDR: External Data Representation Standard, RFC 1014 (1987).
[16] Sawzall Clone: a clone implementation for hadoop, `http://code.google.com/p/sawzall-clone/`.