MapReduce処理系SSS上のSawzall処理系の実装

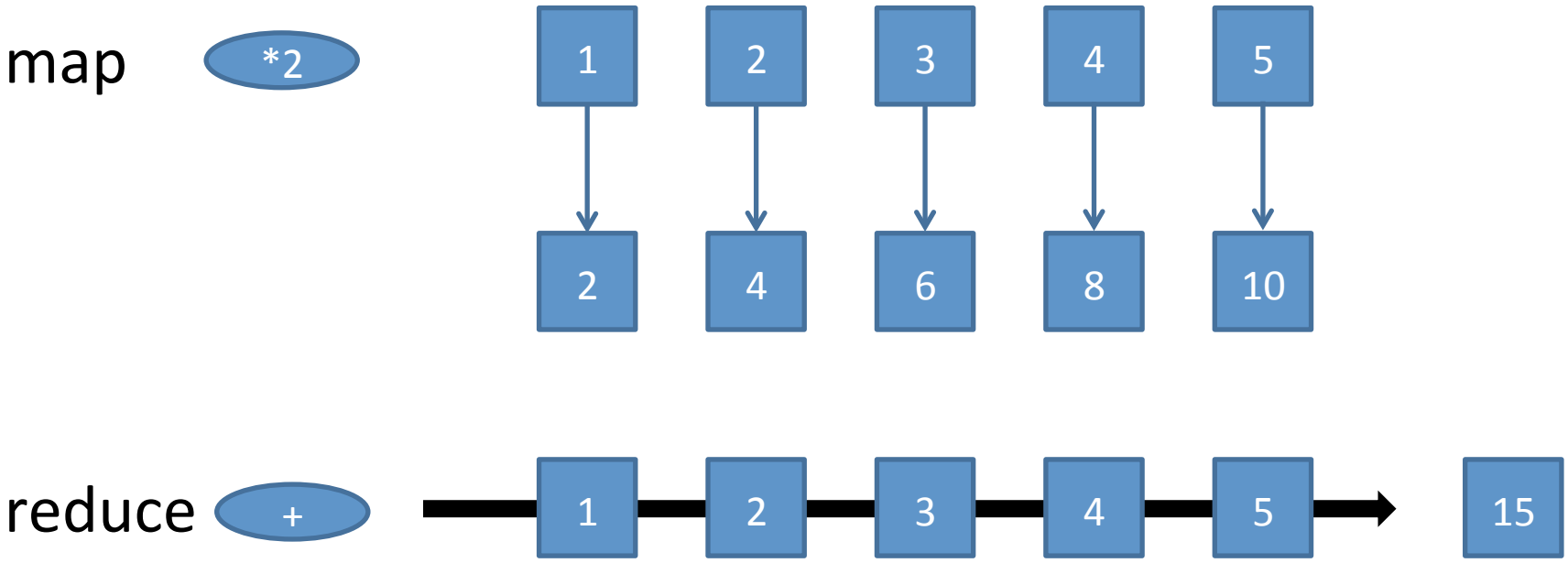
中田 秀基[1], 井上 辰彦[2,1], 小川 宏高[1], 工藤 知宏[1]

1. 独立行政法人産業技術総合研究所

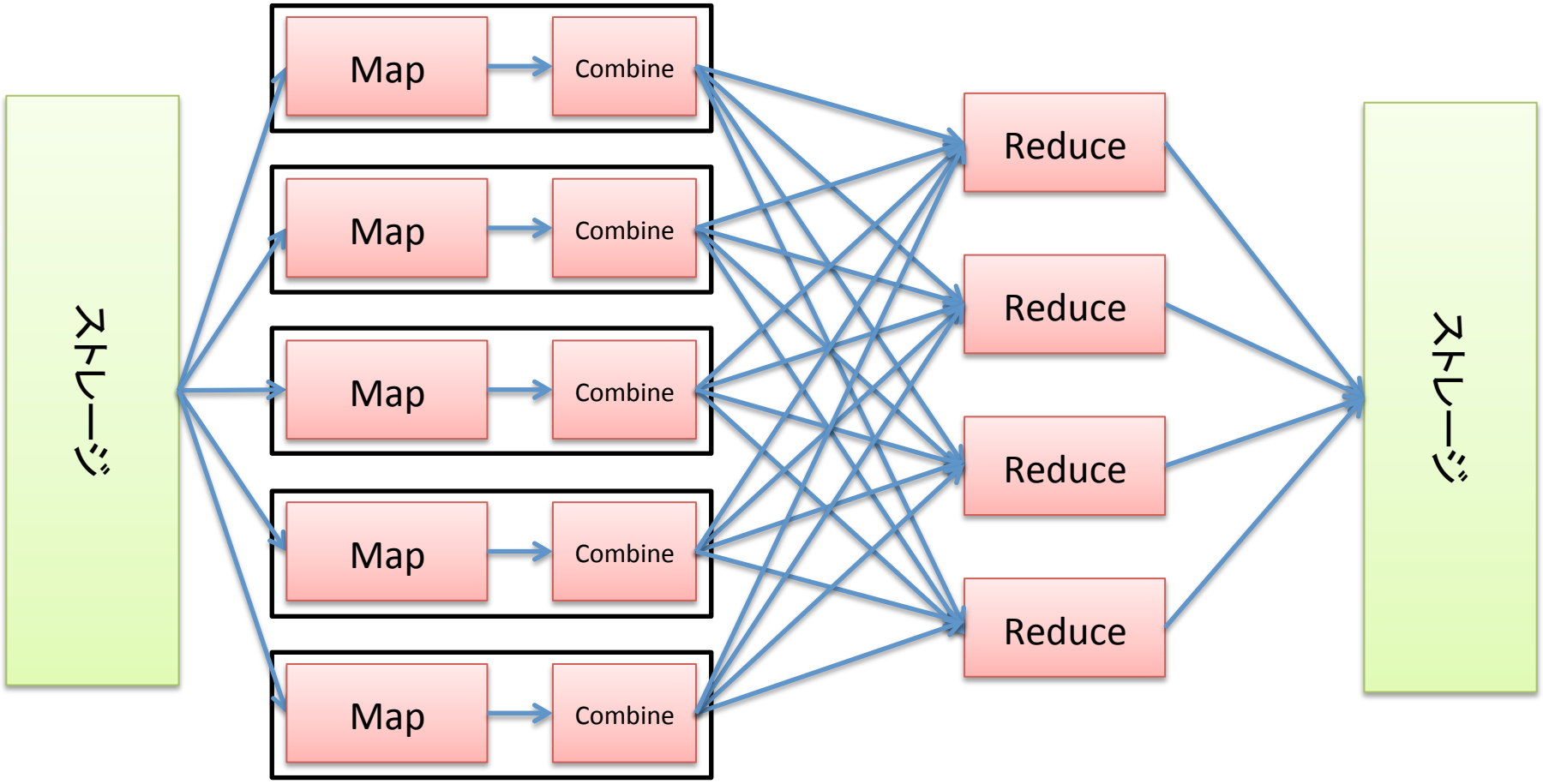
2. 株式会社創夢

MapReduceとは

高階関数を持つ言語に一般的なmapとreduce関数にヒント



MapReduceの概要



背景

- MapReduceの普及
  - カジュアルユーザによる並列プログラミング
- MapReduce プログラムの直接の記述は煩雑
  - たとえば Hadoop では Mapper, Reducer, メインプログラムの3つのプログラムが必須
  - プロトタイピングによるインクリメンタルなデータマイニングには不適
- MapReduce 向け言語
  - HiveQL, PigLatin, Jaql, Sawzall
  - MapReduce プログラムの容易な記述

- KVSをベースとしたMapReduce システム SSS
  - HPC研究会で昨日発表

Hadoopのプログラム

```
public class WordCount {
  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer tokenizer = new M;
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
      }
    }
  }
```

Mapper

```
  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, Context context)
     throws IOException, InterruptedException {
      int sum = 0;
      while (values.hasNext())
        sum += values.next().get();
      context.write(key, new IntWritable(sum));
    }
  }
```

Reducer

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
  }
}
```

ドライバ

Hadoop Java API

本研究の目的

- Sawzall 言語処理系を実装
  - ScalaによるJavaへのコンパイラとして
  - SSS/Hadoop をターゲット
- 言語処理系を評価
  - Szl との比較
  - SSS上での実行速度をネイティブなAPIと比較

アウトライン

- Sawzallの概要
- MapReduce 処理系 SSSの概要
- SawzallClone の設計と実装
- 評価
  - Szlとの比較
  - SSSネイティブアプリケーションとの比較
- まとめと今後の課題

Sawzall

- Google のMapReduce用の言語
- Map部分の記述に特化、Reduceを隠蔽
  - Map - 1入力多出力 - c.f. Awk
  - 入出力は**Protocol Buffers**を前提
  - Inputという特殊な変数にバイト列として値がバインドされて呼び出される
- Reduceはテーブルという概念で抽象化
  - Mapはテーブルへ’emit’
  - テーブルは言語組み込み

Sawzallサンプル

ログから時刻を抜き出して、分単位で頻度をカウント

Proto文

テーブルの宣言

```
proto "p4stat.proto"
submitsthroughweek: table sum[minute: int] of count: int;

log: P4ChangelistStats = input;

t:      time = log.time;
minute: int  = minuteof(t)+
               60*(hourof(t) + 24*(dayofweek(t)-1))

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

入力データをキャスト

テーブルにemit

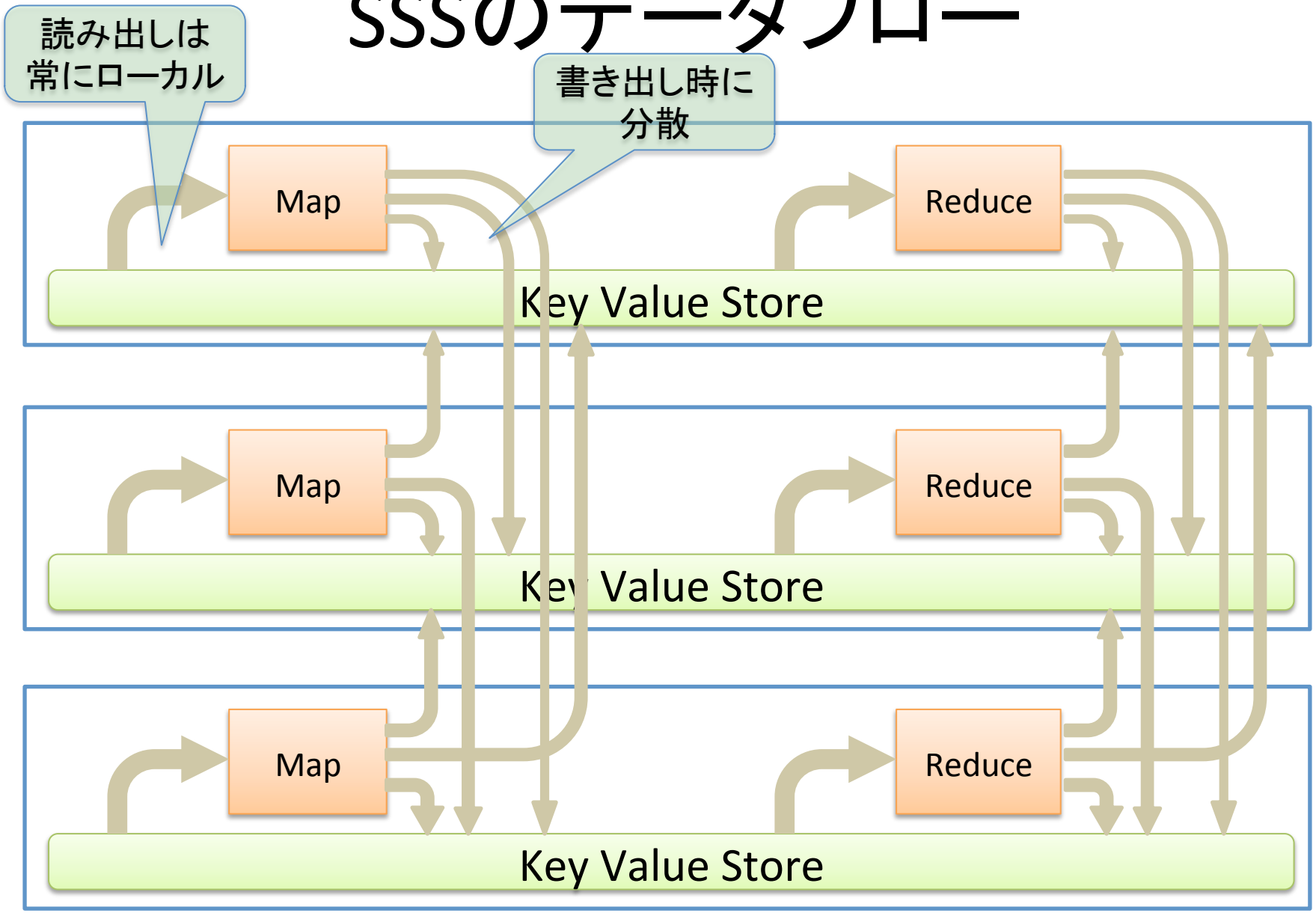
Sawzallのテーブル例

- 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

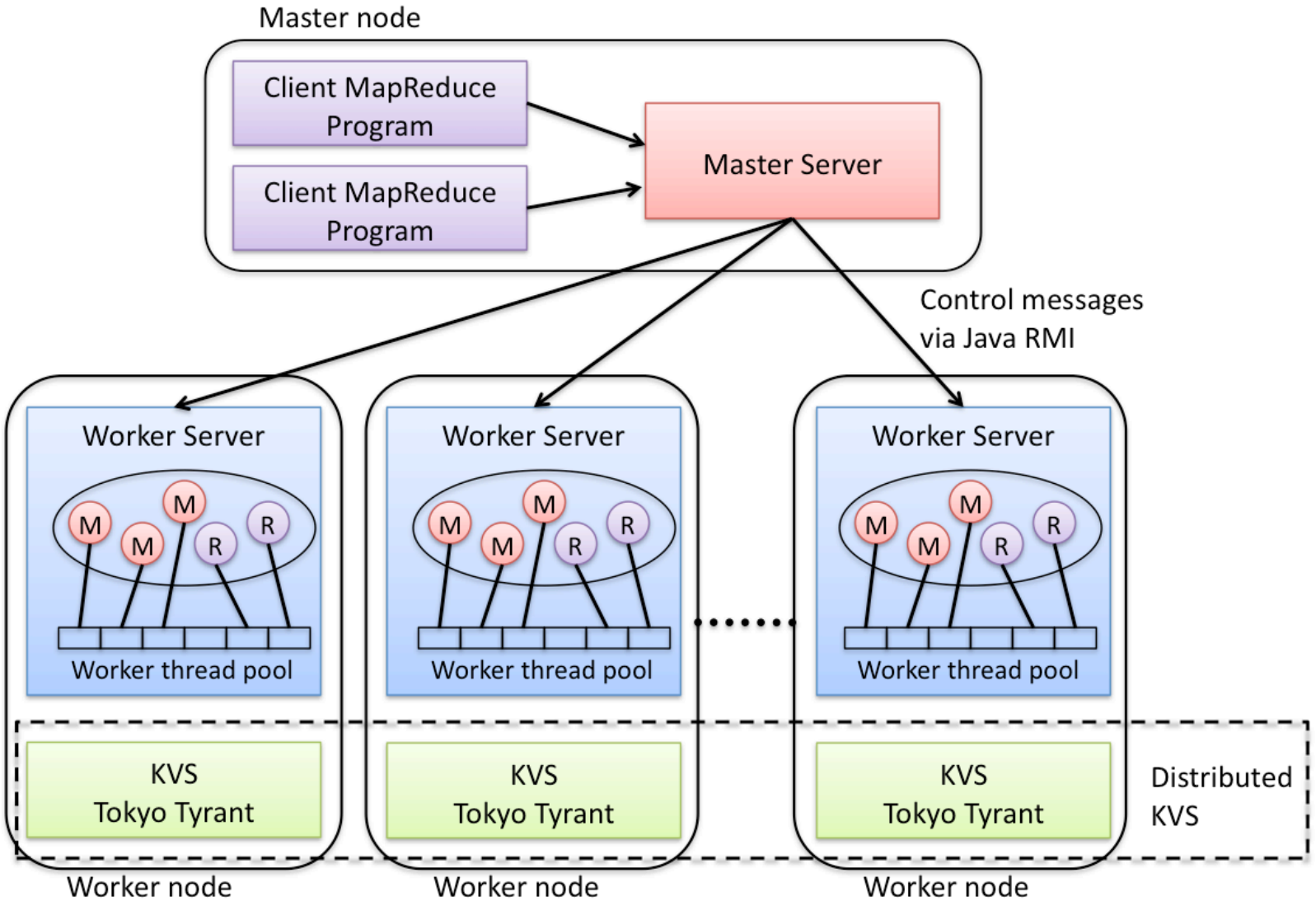
SSS

- ファイルシステムではなく分散KVSをベースに
  - 繰り返し実行が高速
- Owner Computes Rule
  - データをハッシュで分散
  - データがある場所でMap / Reduce
  - スケジューリングコストが殆どかからない
- KVSは、Tokyo Tyrant をハッシュで分散したものを利用
  - ソート済みデータのバルク読み出し・書き込みに特化してTokyo Cabinetを修正

SSSのデータフロー

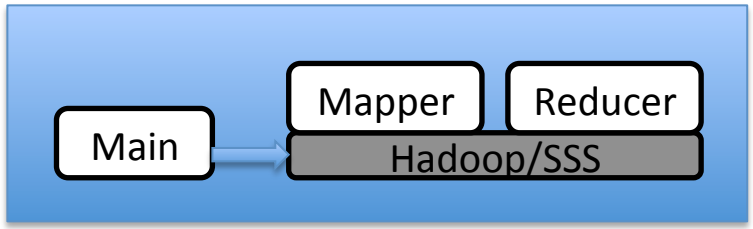

読み出しは
常にローカル
書き出し時に
分散
Map
Reduce
Key Value Store
Map
Reduce
Key Value Store
Map
Reduce
Key Value Store

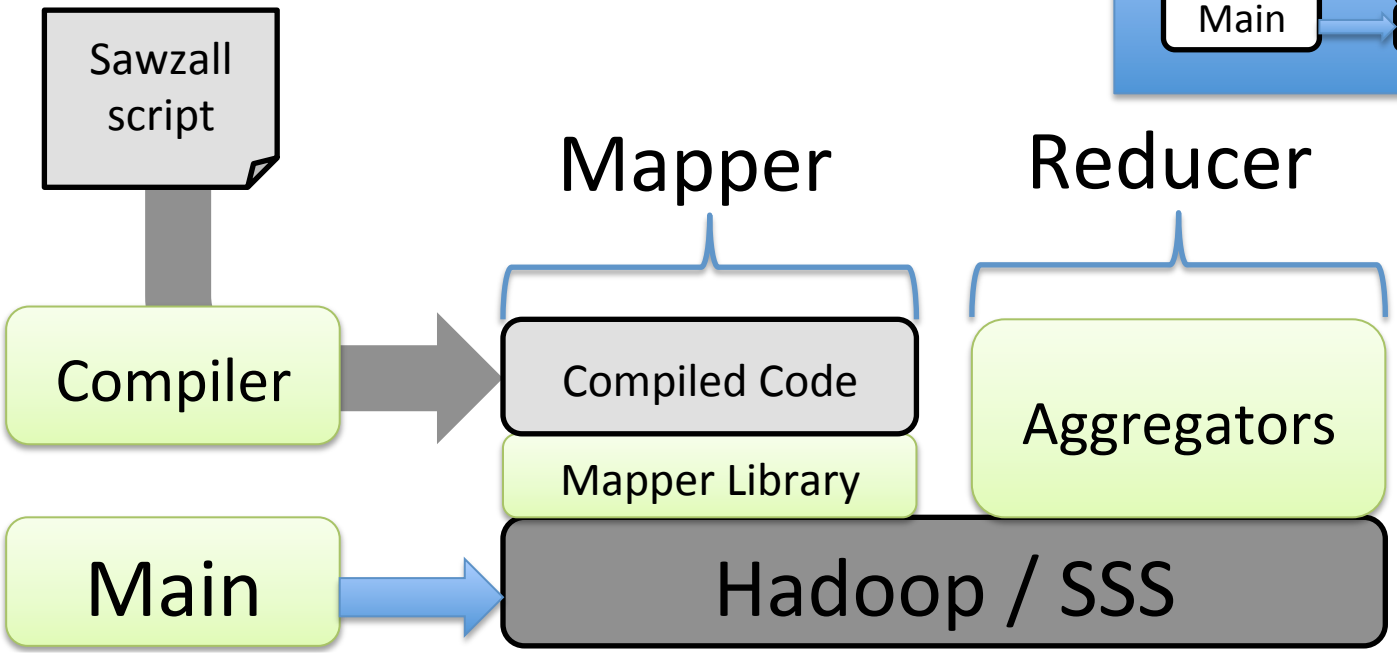
SSSの構成


Master node
Client MapReduce Program
Client MapReduce Program
Master Server
Control messages via Java RMI
Worker Server
M
M
M
R
R
Worker thread pool
Worker Server
M
M
M
R
R
Worker thread pool
Worker Server
M
M
M
R
R
Worker thread pool
KVS Tokyo Tyrant
KVS Tokyo Tyrant
KVS Tokyo Tyrant
Distributed KVS
Worker node
Worker node
Worker node

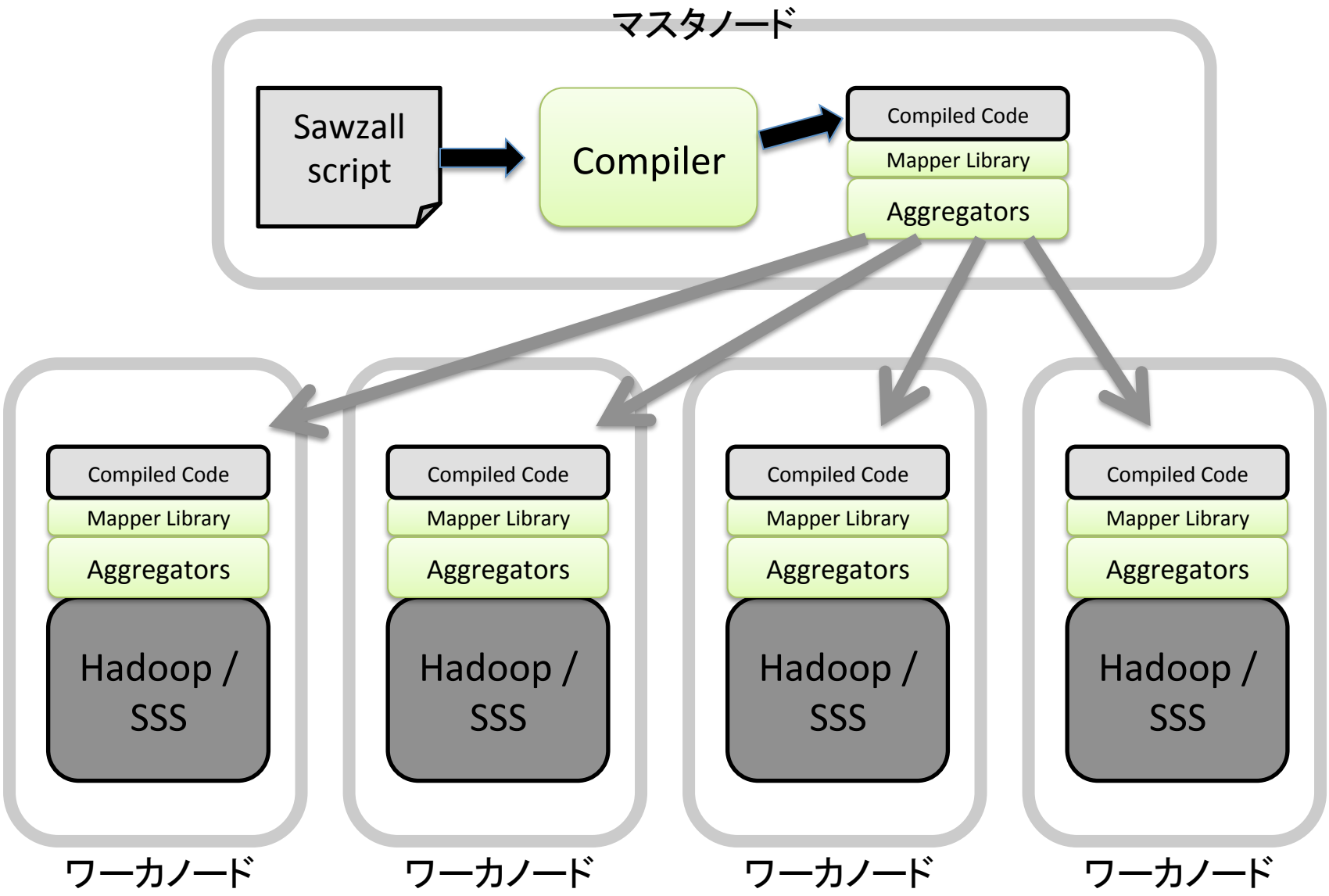
SawzallCloneの概要

- スクリプトで記述するのはMapper部分のみ
  Reducer 部分はライブラリで提供
- Scala によるJava言語をターゲットとしたコンパイラとして実装
- 4つのモジュールから構成



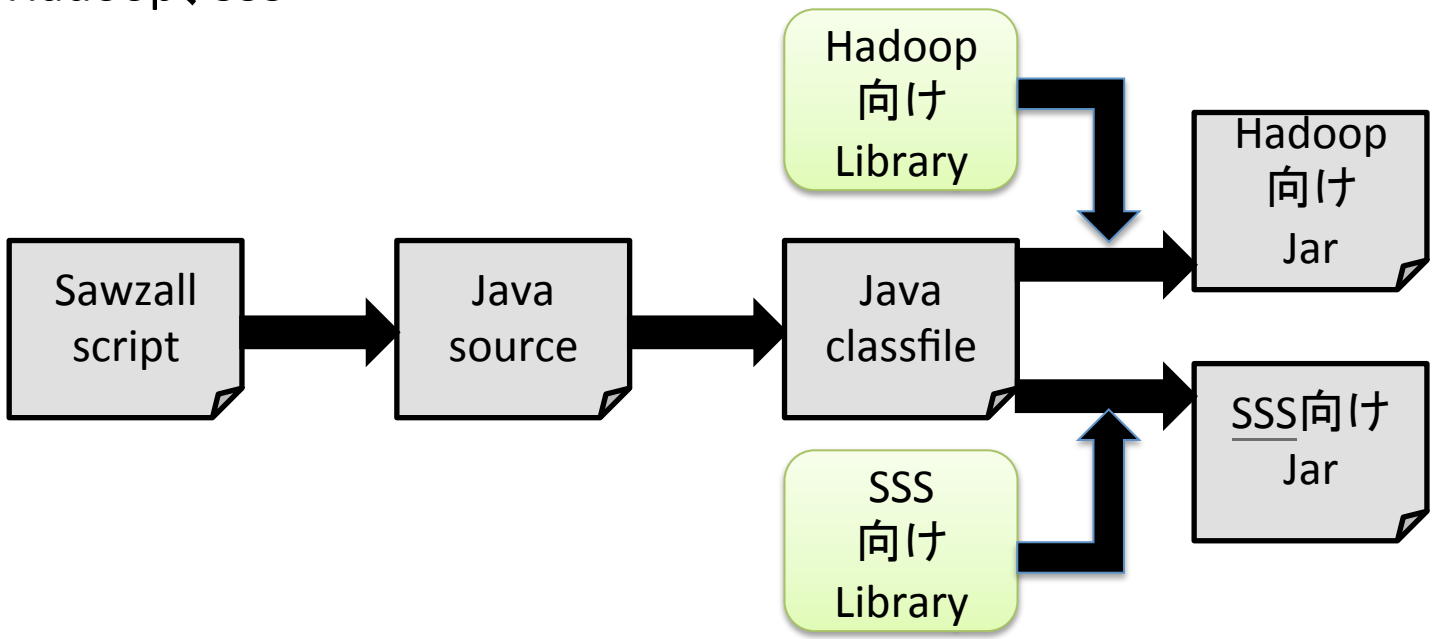


SawzallCloneの実行イメージ



コンパイル

- Javaコードを出力
  - 実装を簡素化するため
- ライブラリで抽象化することで下位のMapReduceシステムから独立のコードを出力
  - Hadoop、SSS



出力コード

```
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));
      }
    }
  }
}
```

スキーマ情報のとりこみ

- ProtocolBuffers IDLで記述された
  スキーマ情報へのアクセス

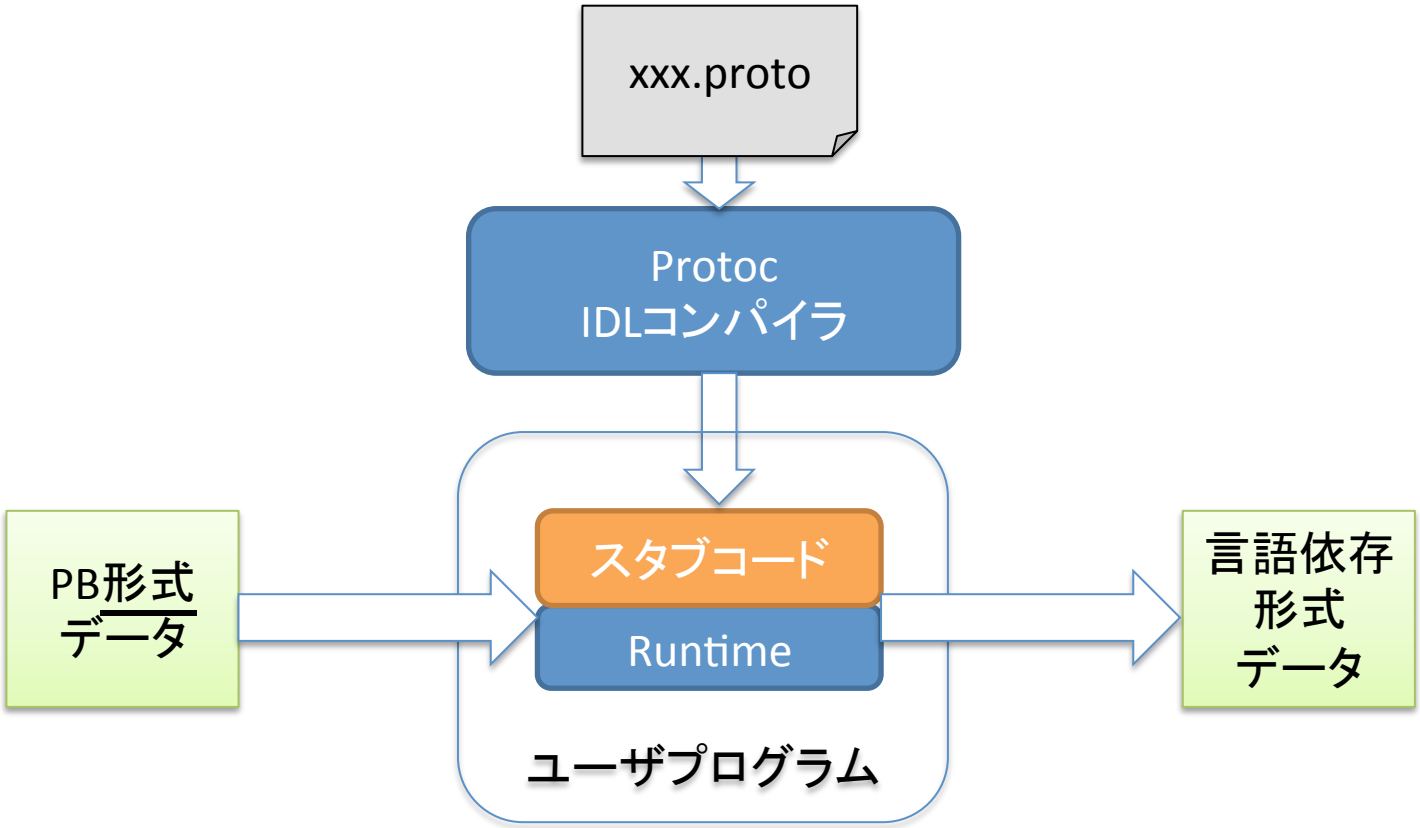
```
proto "p4stat.proto"
submitsthroughweek: table sum[minute: int] of count: int;

log: P4ChangelistStats = input;

t:      time = log.time;
minute: int  = minuteof(t)+
               60*(hourof(t) + 24*(dayofweek(t)-1))

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

Protocol Buffers

- Googleが使用している言語・アーキテクチャ非依存な構造データのマーシャリング形式
- Protoファイルと呼ばれるIDLで構造を定義
- ProtocというIDLコンパイラでスタブファイルの生成とメタ情報の書き出しを行う

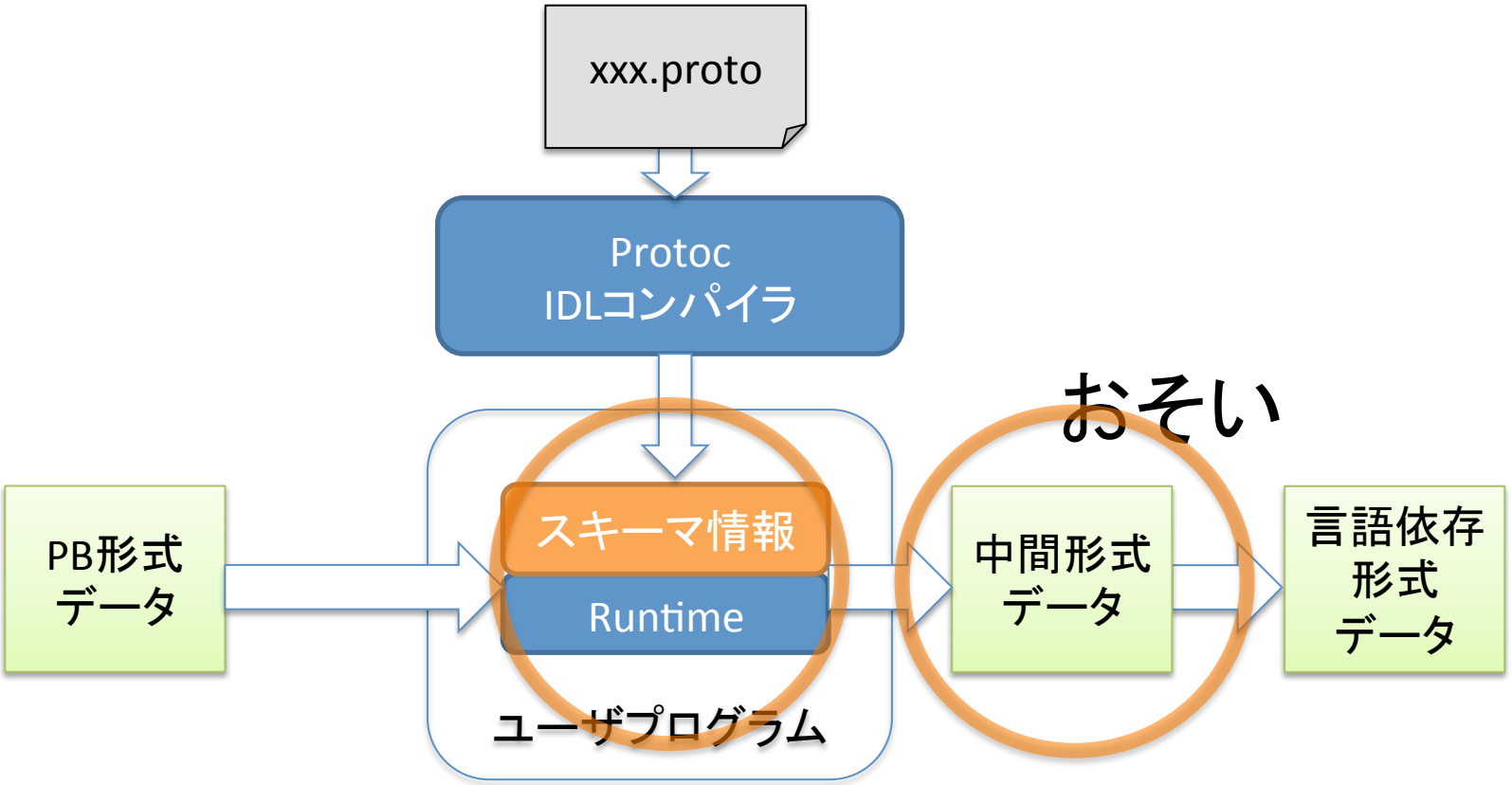
```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Protocの機能



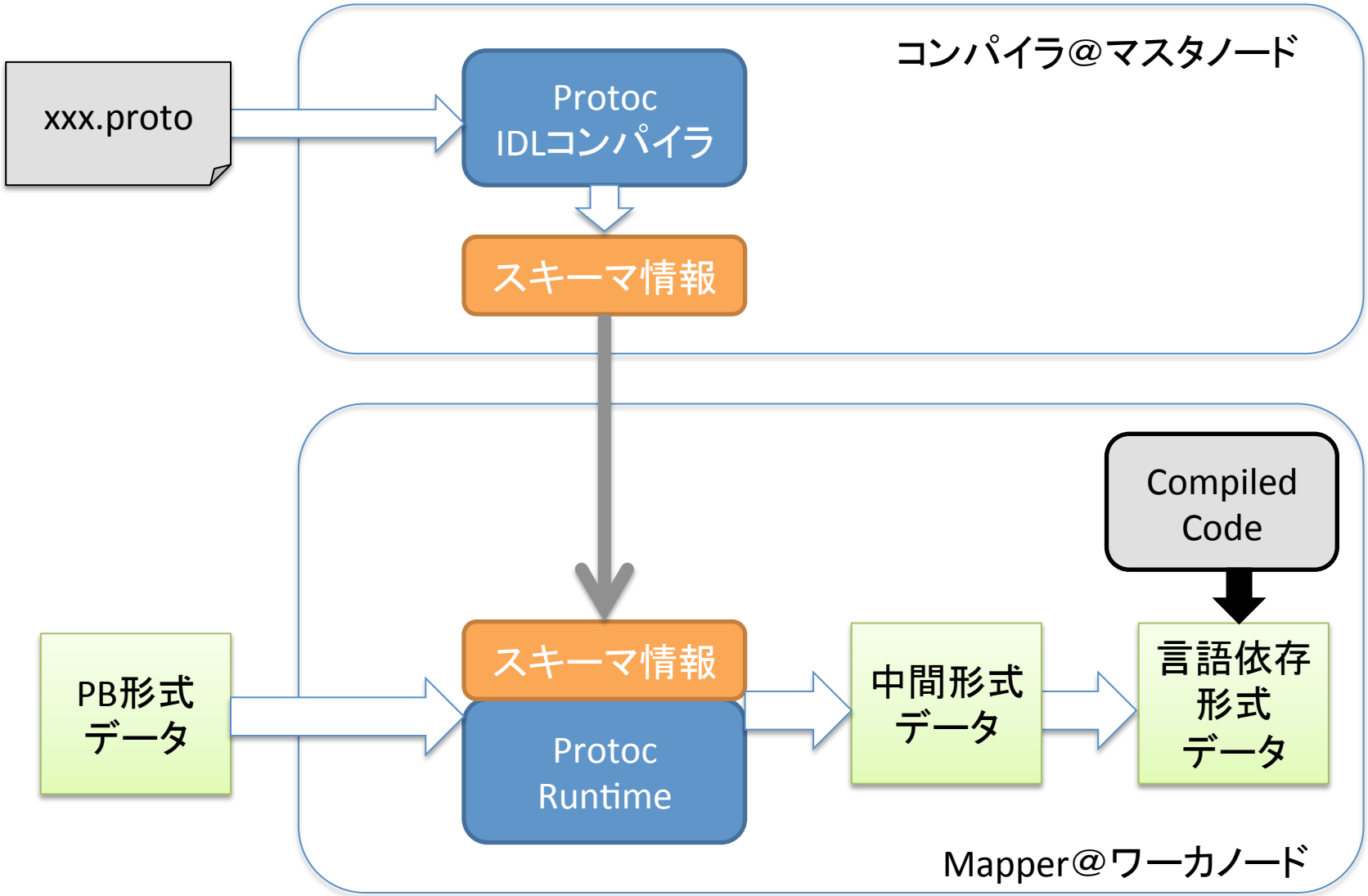
スキーマ情報にアクセスできない

Protocの機能（2）



スキーマ情報にアクセスできる

SawzallCloneでのProtocolBuffresの処理



評価

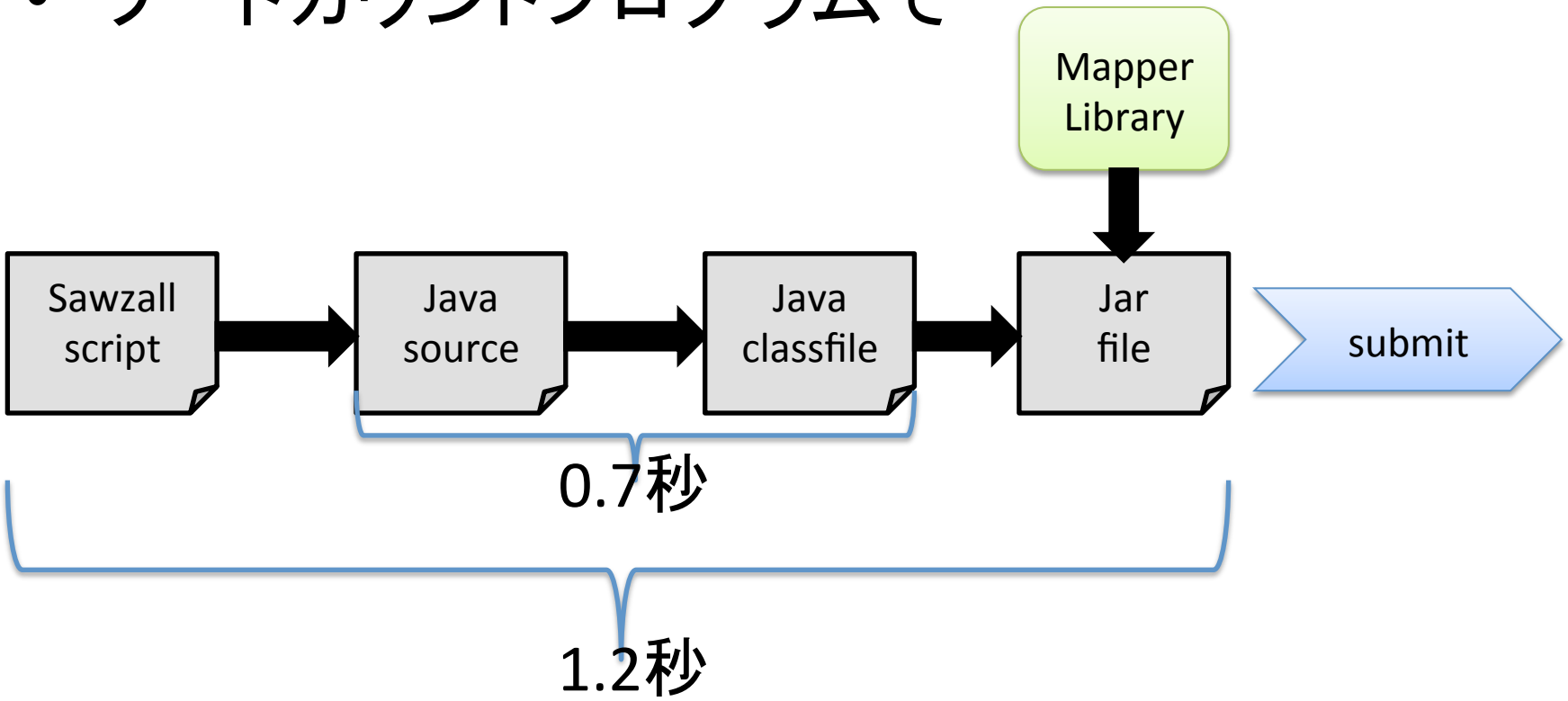
- コンパイル時間
- Szl との比較
  - Szl : Google によるオープンソース実装
    - C++によるコンパイラとバイトコード処理実行系
    - 逐次実装しか存在しない
  - 逐次実行で比較
- SSSでのネイティブ実装との比較
  - 逐次実行
  - 並列実行

評価環境

- クラスタを使用
  - Number of nodes: 16 + 1 (master)
  - CPUs per node: Intel Xeon W5590 3.33GHz x 2
  - Memory per node: 48GB
  - OS: CentOS 5.5 x86_64
  - Storage: Fusion-io ioDrive Duo 320GB
  - NIC: Mellanox ConnectX-II 10G
- ソフトウェア
  - SSS
  - Hadoop 0.20.2
    - HDFSレプリカ数を1に設定
    - Nodeあたりのmapper数7

コンパイル時間

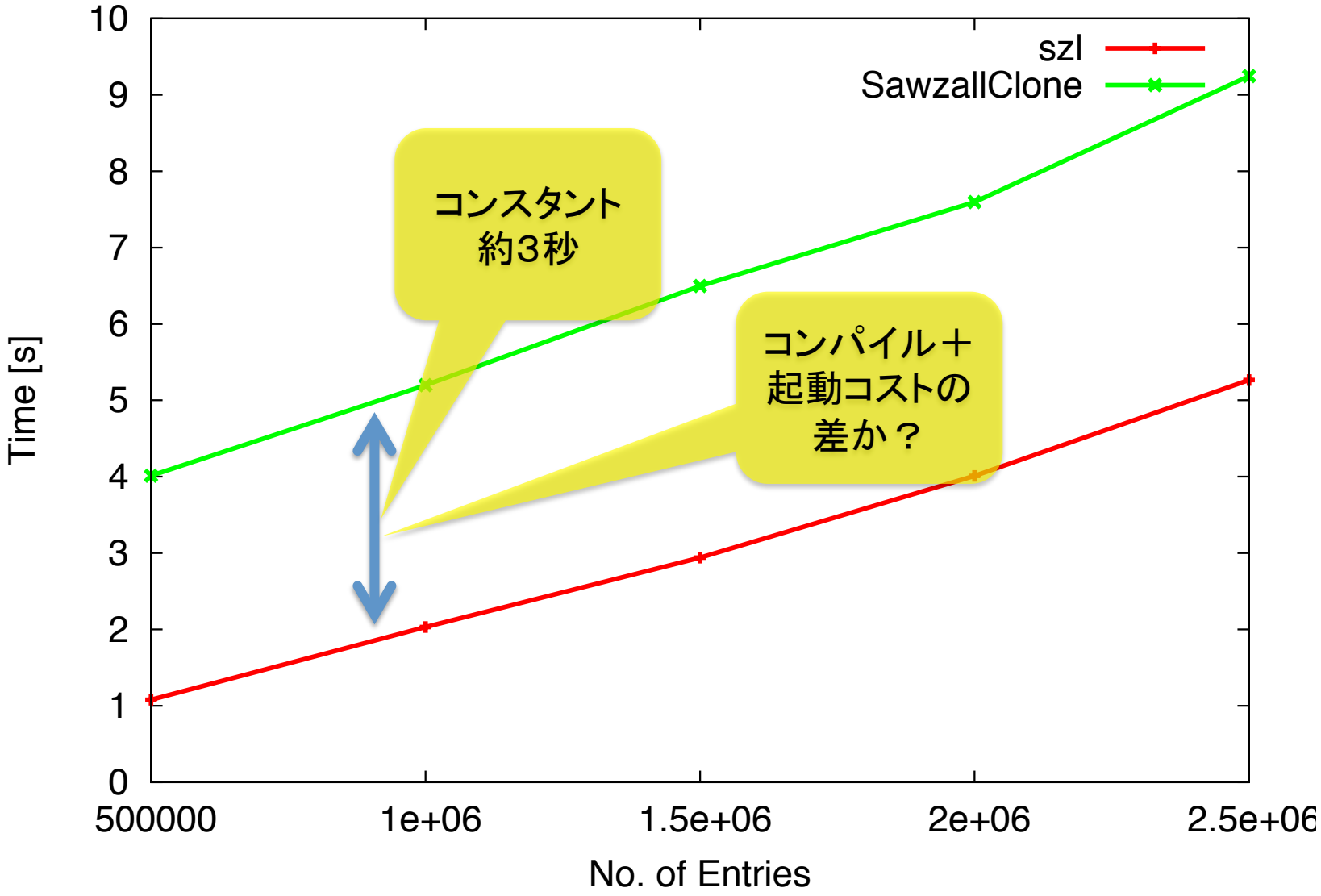
- Sawzall Clone は実行時にコンパイル
  - 2段階のコンパイルとJar の作成
- ワードカウントプログラムで



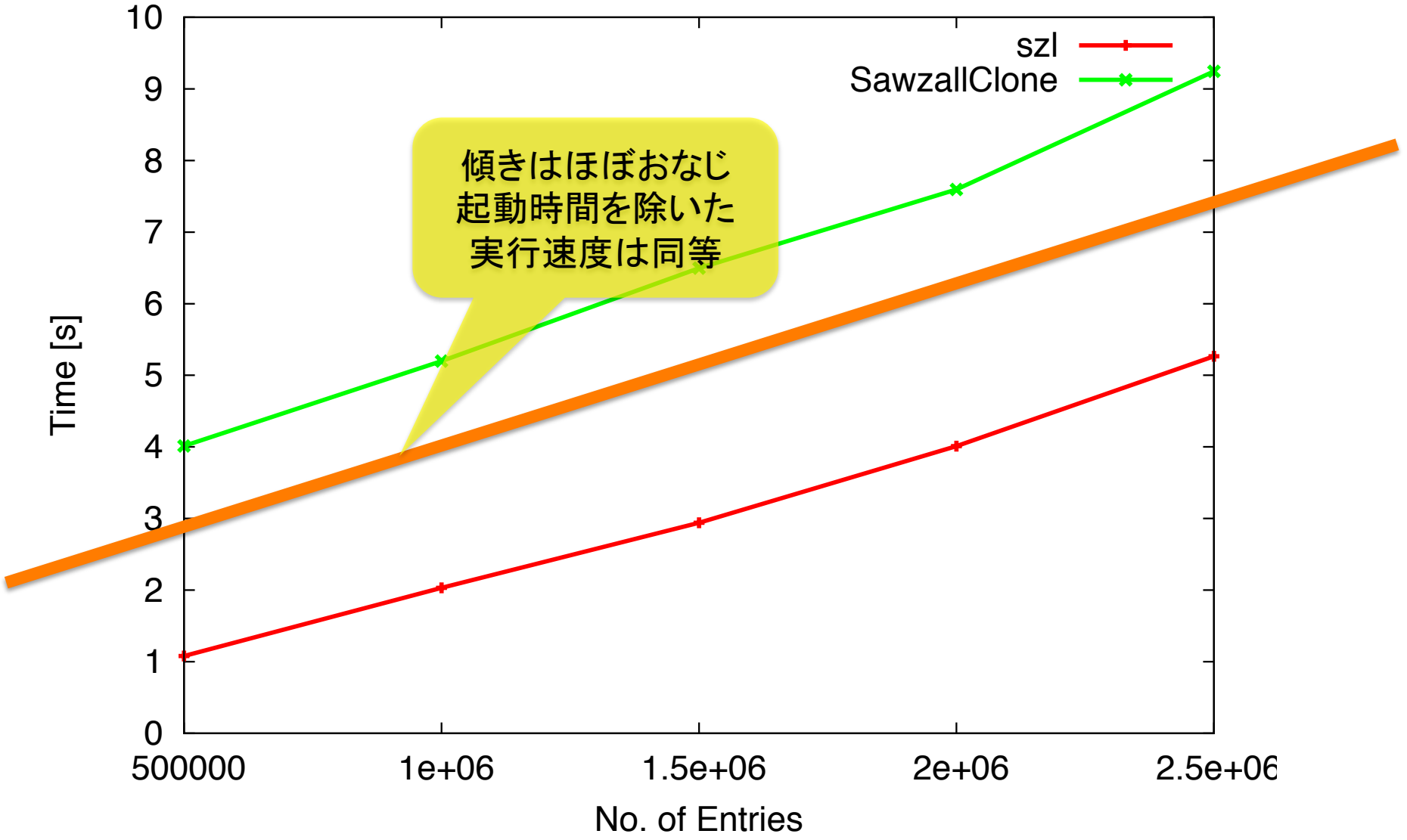
Szlとの比較

- 文献で紹介されているログ解析プログラムを使用
  - ログエントリをプロトコルバッファでエンコードしたものを入力とする
- 逐次実行
  - SawzallCloneも単一VM内でMapperとReducerを実行
- ログアイテムの数を50万 － 250万に変化させ挙動を観察

Szlとの比較


10
9
8
7
6
5
4
3
2
1
0
Time [s]
500000
1e+06
1.5e+06
2e+06
2.5e+06
No. of Entries
szl
SawzallClone
コンスタント
約3秒
コンパイル＋
起動コストの
差か？

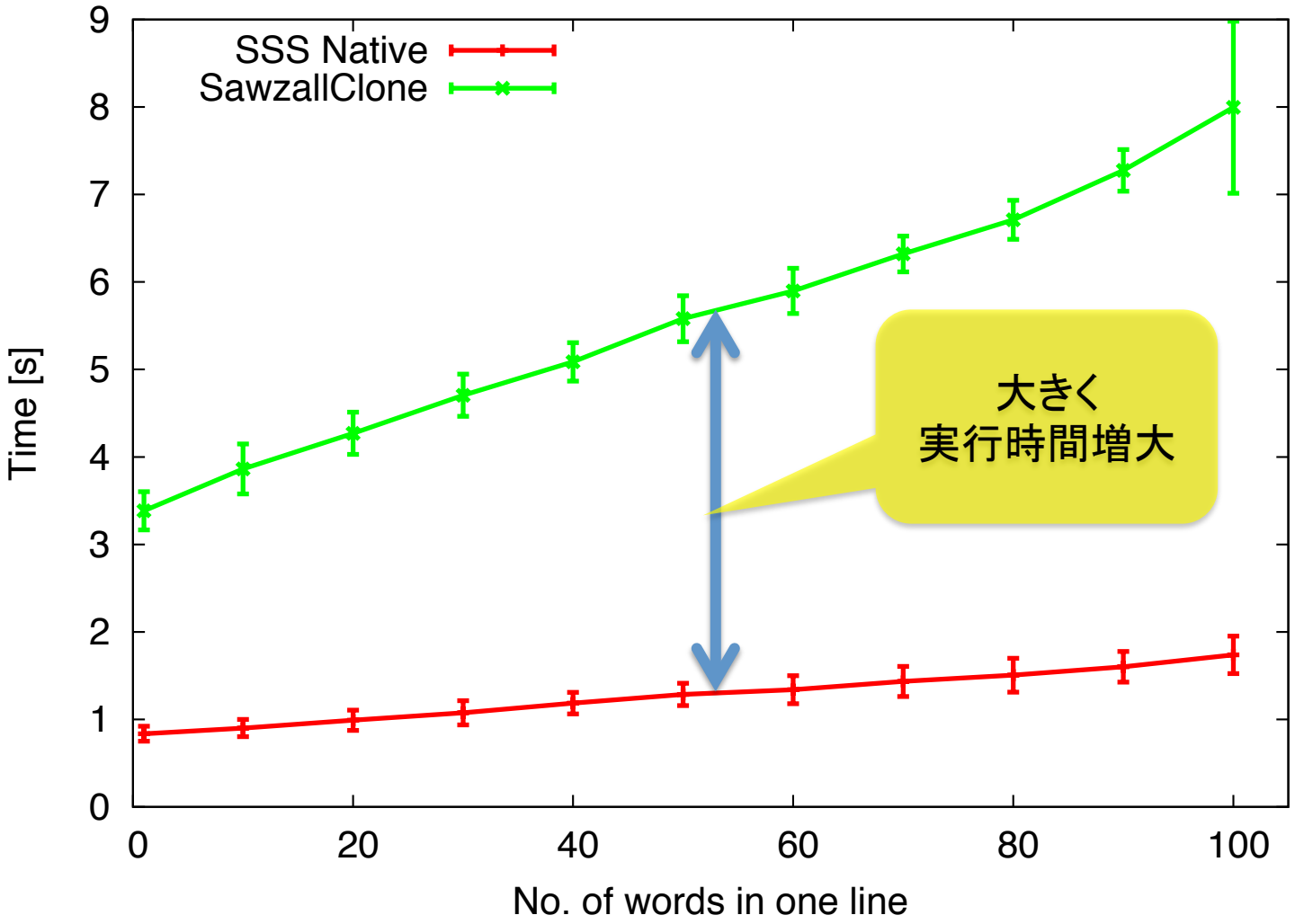
Szlとの比較


10
9
8
7
6
5
4
3
2
1
0
Time [s]
500000
1e+06
1.5e+06
2e+06
2.5e+06
No. of Entries
szl
SawzallClone
傾きはほぼおなじ
起動時間を除いた
実行速度は同等

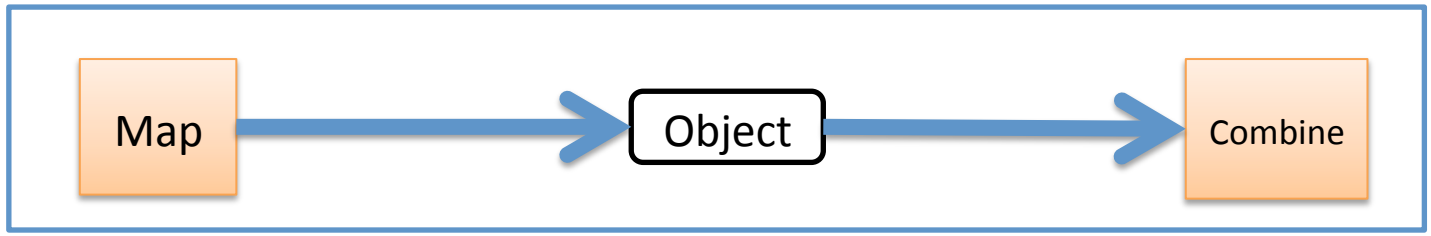
逐次SSSネイティブAPIとの比較

- ワードカウントを使用
- 各ラインのワード数を増減
  - スクリプトは各行に対して起動される
  - ワード数はスクリプト内のループに相当
  - 実行時間の変化を観察

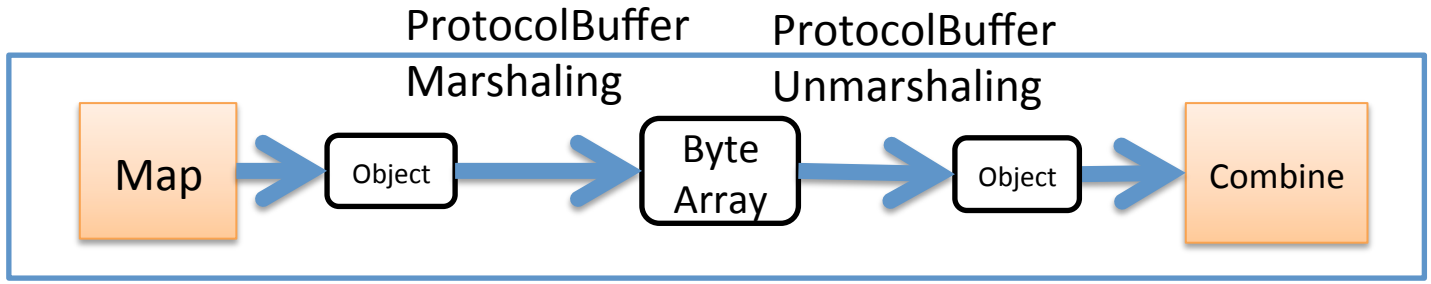
逐次SSSネイティブAPIとの比較



Mapper-Combiner 間の マーシャリングが原因？



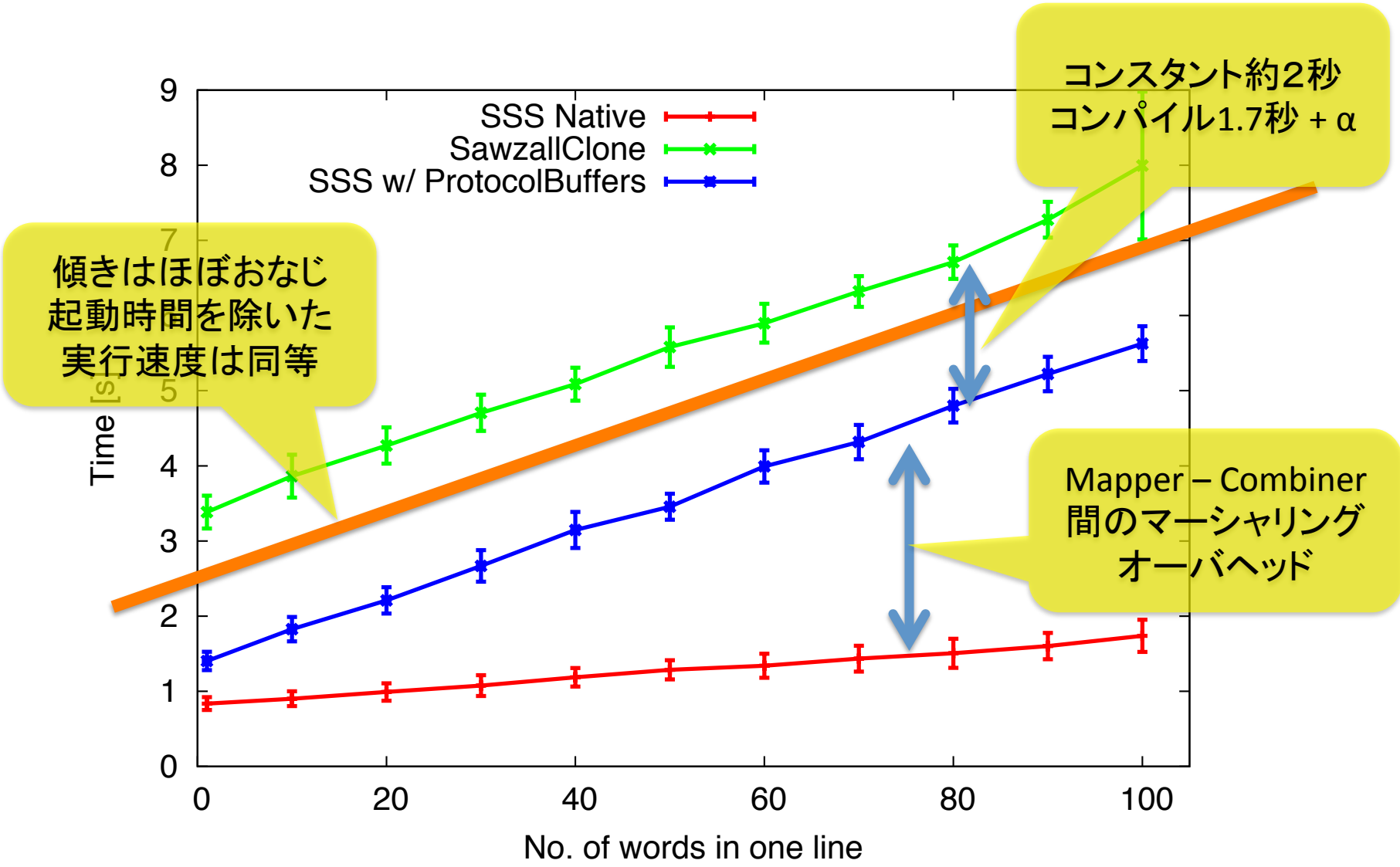
SSS Native Application



SawzallClone Implementation

ネイティブ版でもマーシャリングを行うようにして検証

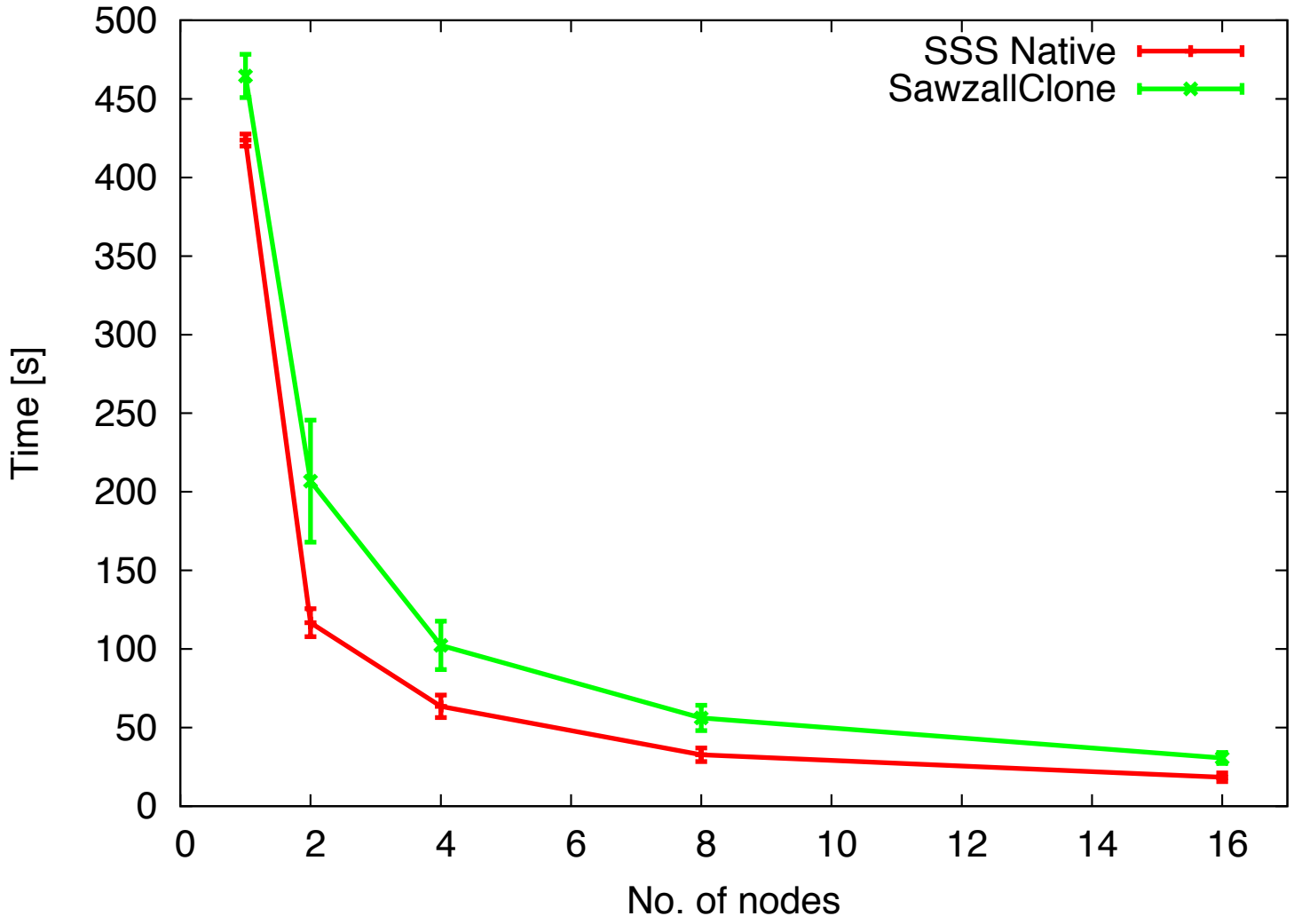
逐次SSSネイティブAPIとの比較



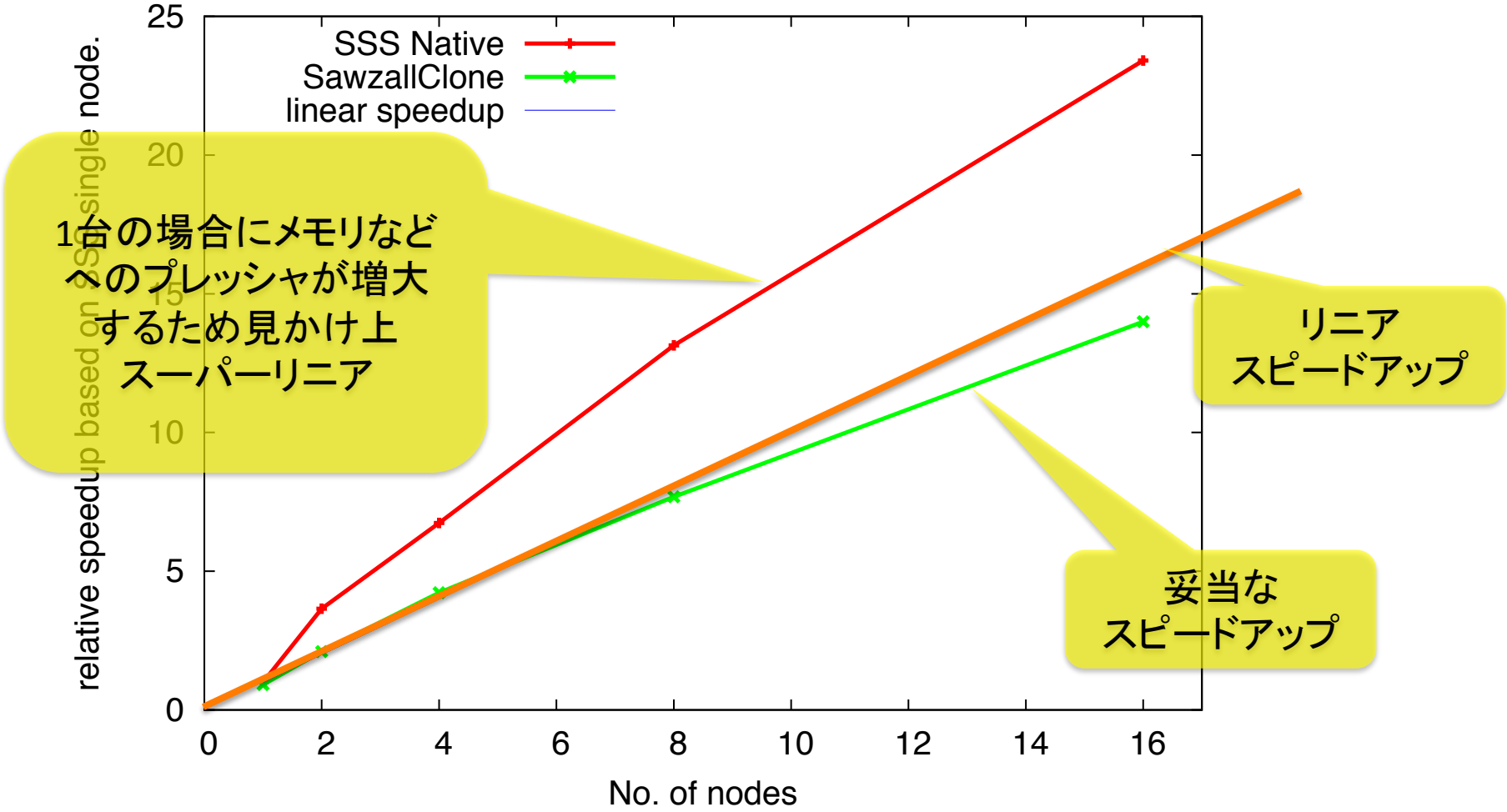
並列SSSネイティブAPIとの比較

- ログ解析プログラムを実行
  - データ量を固定
    - ログ1億レコード
  - ノード数 1,2,4,8,16

並列SSSネイティブAPIとの比較



並列SSSネイティブAPIとの比較



まとめ

- Sawzall をSSS / Hadoop 向けに実装
  - ScalaによるJavaへのコンパイラ
  - ランタイム
- 評価
  - Szlとの逐次性能比較
    - コンパイルオーバヘッドはあるが実行性能自体は同等
  - SSSのネイティブAPIとの比較
    - 逐次・並列とも言語オーバヘッドを確認
    - Mapper – Reducer間で余分なmarshaling/をしていることが原因

今後の課題

- Mapper – Combiner間の余分なマーシャリングの除去
- スケーラブルなAggregator 実装
- 言語機能の拡張
  - 複数段のMapReduceのワークフローを記述

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.

ありがとうございました