An Implementation of Sawzall on Hadoop

Hidemoto Nakada*, Tatsuhiko Inoue† and Tomohiro Kudoh‡
*,‡ 1-1-1 National Institute of Advanced Industrial Science and Technology,
Umezono, Tsukuba, Ibaraki 305-8568, JAPAN
† Soum Corporation, 1-34-14 Hatagaya, Setagaya-ku, Tokyo, 151-0072, JAPAN

***Abstract*—Sawzall is a script language designed for batch processing of large amount of data, based on MapReduce parallel execution model, which is introduced by Google in 2006. Sawzall allows programmers only to program *mappers* to ease the burden for them. Sawzall provides a set of built-in *aggregators* that provides reducing function, from which programmers could pick and use. We have implemented a Sawzall compiler and runtime, called SawzallClone, which allows Sawzall scripts to run in parallel on Hadoop. We employed Scala language to leverage Scala's parser combinator libraries for Sawzall syntax parsing. It enabled easy implementation of parser and potential future extension of the language. This paper provides detailed implementation of the system. We performed evaluation on the system comparing with the Java programs that use native Hadoop API and szl, a Sawzall open source implementation from Google. We confirmed that overhead imposed by SawzallClone is small enough, and the execution speed is comparable with szl.**

I. Introduction

MapReduce [10] became very popular as a parallel programming paradigm and along with the wide adoption of Apache Hadoop [1], it is now used in practical business data processing as well as scientific data analysis.

Although they say programming in MapReduce is easy, and actually it is much better than MPI, it is not easy enough for engineers who are not familiar with programming. In Hadoop, programmers have to provide three different programs, namely, *mappers*, *reducers*, and *the main program*, which configure mappers and reducers, as shown in figure 1. Moreover, in Hadoop, static type checking for intermediate data does not work. It causes runtime error and complicates debugging the programs.

Google proposes a language called *Sawzall* [14] that is intended for large-scale log analysis with MapReduce. Sawzall ease the burden of programmers providing reducers as built-in capability of the language system. Programmers are only responsible for writing mappers. In 2010, Google published Sawzall system called *szl* [7] as open source software. It is, however, not for parallel execution. It just provides sequential execution so far.

We implemented a Sawzall, named SawzallClone, that leverage Hadoop as a runtime environment. SawzallClone is composed of compiler and runtime library both written in Scala. Sawzall scripts are compiled into Java source code that will do the task of mapper, and then compiled into Java byte code with Java compiler. Runtime library provides reducer capability.

We evaluated SawzallClone on Hadoop comparing with programs written with native Hadoop API. The result showed that overhead imposed by SawzallClone is negligible.

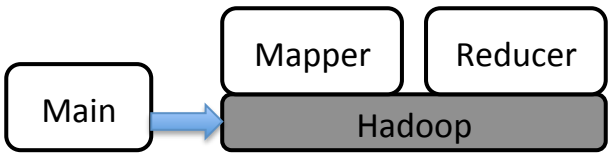

Fig. 1. Program for Hadoop.

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

Fig. 2. An Example of Hive QL Description.

The rest of paper is composed as follows: Section II introduce some related work. In section III, we give brief description of the Sawzall language. In section IV, we describe detailed implementations of SawzallClone. Section V provides evaluation compared with native Hadoop description. In Section VI, we conclude the paper with summary and future work.

II. Related work

Several language systems are proposed for easy processing on Hadoop.

a) Hive QL: Hive [2] [16] is a warehouse scale data base system that enables search with a declarative language called Hive QL, which resembles SQL. Hive is extensively utilized in Facebook. Hive stores serialized structured data on HDFS. Meta-information, such as database schemata and table locations on HDFS are managed by an RDB outside of Hadoop, which is called Metastore. While Hive provides a set of built-in functions for QL, users can define his/her own functions called UDF (User-Defined Functions), in Java. As shown in figure 2, Hive QL syntax is quite similar to the standard RDB SQL, making transition from RDB to Hive easy.

b) Pig Latin: Apache Pig [4] [13] is a data processing environment on Hadoop that uses a language called 'Pig Latin' for description. Pig Latin scripts are compiled into several cascading MapReduce processes that run on Hadoop. In Pig Latin, the data processing is described in imperative manner, in contrast with Hive QL that provides declarative description. The style is more intuitive for non-SQL programmers. Figure 3 shows an example of Pig Latin Script. Pig also provides way to extend the language with user-defined functions written in Java.

```
records = LOAD 'input/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records =
   FILTER records BY temperature !=  9999 AND
    (quality == 0 OR quality == 1);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

Fig. 3. An Example of Pig Latin Script.

```
mapReduce(
  { input:  {type: "hdfs", location: "sample.dat"},
    output: {type: "hdfs", location: "results.dat"},
    map:    fn($v) ( $v -> transform [$.x, 1] ),
    reduce: fn($x, $v)
          ( $v -> aggregate into {x: $x, num: count($)} )
  });
```

Fig. 4. A MapReduce program in Jaql

c) Jaql: Jaql [3] [9] is a language system to make searches on data stored on HDFS. It assumes that all the data stored is formatted in JSON (JavaScript Object Notation). JSON is a self descriptive format, i.e. no extra schema description required. Figure 4 shows a MapReduce example in Jaql, which reads structured data in `sample.dat`, groups it by attribute `x`, counts the number, and outputs the results to the file `results.dat`.

III. Sawzall

In this section, we briefly describe the language Sawzall based on the literature [14].

A. Overview

Sawzall [14] is a language based on MapReduce, which is said to be used in Google for log analysis. The goal of Sawzall is to make it even easier to write MapReduce program. The idea is to expose only the map portion to the programmers providing built-in *reducers* as a part of the language runtime. The programmers are only responsible for the mappers. It is intended to make non-expert users possible to write scripts for themselves.

Sawzall is a typed language. It is specialized for mapping, i.e., it processes all the target entries one by one. It is Similar to the AWK language [8], which processes text files one line by one line. There is, however, a fundamental difference. In AWK, there is a global state that could be shared and modified by each line processing. In Sawzall, there is nothing shared. Each entry is processed in completely independent way.

Sawzall provides a set of 'aggregators' to do the reducing job. Using the aggregators, programmers are freed from burden of writing reduction phase.

Sawzall implicitly assumes that all the structured data on the storage are serialized using Protocol Buffer, described below.

B. Protocol Buffers

Protocol Buffers [5] is a binary data representation of structured data used heavily in Google. It is language neutral and architecture neutral, i.e., binary data serialized with Protocol Buffers on a machine with a program are guaranteed to be readable no matter what machine and language is used by the reader. The goal is somewhat similar to XDR (External Data Representation) [15], but it is focusing on reducing data amount in the serialized format.

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

Fig. 5. An Example of proto files.

To use Protocol Buffers, users have to explicitly declare the data schemata with a language neutral IDL. The files contains the declaration are called *proto files*. Figure 5 shows an example of proto file.

The proto files are compiled into stub codes that includes language dependent declaration of the data structure and the serialization / deserialization codes that convert internal data representation from / to external (binary) representation.

Google made open source the Protocol Buffers compiler, called `protoc`, and the runtime libraries, which supports C++, Java and Python. Protoc also can produce 'meta-data' file that includes the declared schema in the Protocol Buffers binary format. A runtime library to read serialized data using the meta-data file is also provided.

C. Using Protocol Buffers in Sawzall

As mentioned above, Sawzall implicitly assumes that all the data in the storage is serialized in Protocol Buffers format. To parse the serialized data, the proto files are required. Sawzall provides `proto` statement to refer the proto files.

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

With this statement, data structures that correspond to the schema in the proto file are declared and implicit data conversion functions between the structures and byte array are defined, so that the data structure could be used in the Sawzall program.

D. Special Variable `input`

The Sawzall programs will receive data from a special variable named `input`; just like AWK scripts get each input line stored special variable `$0`.

The `input` variable is typed as byte array, often storing serializing some data structure with Protocol Buffers. To use the value as a structured data, it have to be deserialized. In Sawzall, data type conversion happens implicitly upon assignment. In the following code snippet, the incoming data is deserialized into a data structure typed P4ChangelistStats.

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

E. Tables and `emit`

Aggregators appear in Sawzall programs as special language structures called *tables*, defined with keyword `table`. Here is a sample declaration of table.

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

Note that `sum` is a predefined table type that means the table sums up all the data it gets. The declaration above defines an array of `sum` type table of `int`.

TABLE I
SAWZALL TABLES.

Table name	Description
collection	creates collection of all the emitted data
maximum	outputs largest items
sample	statistically samples specified number of items
sum	sums up all the data
top	outputs most frequent items
unique	statistically infers number of unique items

```
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) +
8               60*(hourof(t) + 24*(dayofweek(t)-1))
9 emit submitsthroughweek[minute] <- 1;
```

Fig. 6. An Example of Sawzall Script

To output to tables, `emit` statements are used. The data emitted to a table are handled by an aggregator that corresponds to the table type. Table I shows principal tables in Sawzall.

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

F. An Example of Sawzall script

Code shown in figure 6 is an example of Sawzall script taken from [14], which analyze web server logs to get histogram of request frequency for each minute.

In line 1, a `proto statement` imports a proto file to define input data type. The line 2 defines an array of tables of sum type for int values. The line 4 reads data from variable `input` and converts it into structure type `P4ChangelistStats`, which is defined in the proto file above, and assigns it to the variable named `log`. The line 6,7 and 8 pulls out recorded time from the structure and calculate minutes. In line 9, the `emit` statement outputs 'one' to table corresponds to the calculated minutes. The table is typed as `sum`, so it sums up one for each emission. As a result, it counts up the emission.

IV. IMPLEMENTATION OF SAWZALLCLONE

A. Overview of SawzallClone

The SawzallClone system is composed of following four components as shown in figure 7;

- Sawzall compiler
- Mapper Library
- Aggregators
- The main component as the entry point

The Sawzall compiler reads Sawzall source code, generates Java source code, compile it into Java byte code using `javac` and create jar for it. The mapper library will be linked with the compiled code and works as the Hadoop mapper. The aggregators are implemented as the Hadoop Reducer module.

The main component works as the entry point for Hadoop. It invokes Sawzall compiler and sets up configurations for the system.

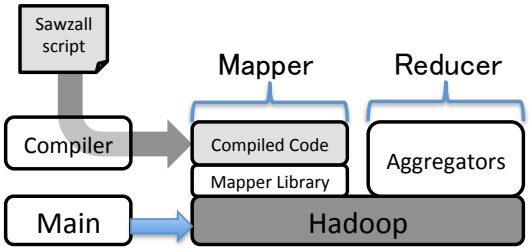

Fig. 7. Overview of Implementation.

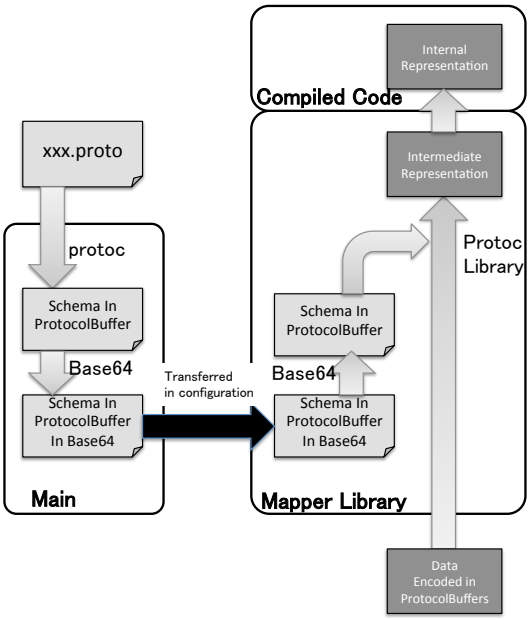

Fig. 8. Processing of Protocol Buffers message.

B. Including Protocol Buffers Schema

As described in III-C, Sawzall must handle Protocol Buffers schemata. We leveraged Protocol Buffers implementation from Google.

Upon compilation of the Sawzall source code, the compiler finds `proto` statements and invokes `protoc` on the specified file to generate compiled schema definition in Protocol Buffers format. The definition is encoded with base64 to make it human readable string and embedded in the Hadoop configuration structure and passed to the SawzallClone mapper library.

The SawzallClone mapper library will extract the encoded schema and decode it. Using the schema, it reads the byte array from HDFS and decodes them into intermediate data structure defined by Protocol Buffers implementation. The data structure will be converted into SawzallClone native internal representation.

C. Parser Combinators

We choose the Scala language for implementation, because of two reasons. One is that, since Scala programs are compiled into Java byte-codes, it is easy to use it with Hadoop, which is implemented in Java.

```
WhenStatement =
 'when' '(' {QuantifierDecl} BoolExpression ')' Statement.
QuantifierDecl = var_name ':' quantifier Type ';'.
quantifier = 'all' | 'each' | 'some'.
```

Fig. 9. when Statement in BNF.

```
def whenStmt     = ("when" ~> "(" ~> whenVarDefs ~
                    expr ~< ")" ~ statement ) ^^@ WhenStmt
def whenVarDefs  = rep1(whenVarDef)
def whenVarDef   = (varNames ~< ":" ~ whenMod ~
                    dataType ~< ";")         ^^@ WhenDefVar
def whenMod: WhenMod[Parser] = "some" ^^^@ SomeMod() |
                               "each" ^^^@ EachMod() |
                               "all"  ^^^@ AllMod()
```

Fig. 10. When implementation with Parser Combinator.

Another reason is the Parser Combinators library that is included in Scala [11] [6] standard libraries. Parser Combinators is a technique common in functional languages to describe parsers. Where, parsing functions, which correspond to each syntax component, are combined by *combinator functions* composing parsing function for the whole language. With the Parser Combinators, parser functions could be described in similar way with the description in BNF.

While the technique Parser Combinators is not specific for Scala and could be used in any language with first-order functions and operator overloading, Scala provides Parser Combinators as a part of standard libraries comes with the language.

To demonstrate parser combinator, implementation of *when* statement is shown in figure 10. Note that this is a plain Scala program, not a parser generator source code. When statement is a Sawzall specific language construct that combines *while* loop statement and *if* condition statements. A BNF based description is shown in figure 9. Although keywords, such as `WhenStatement` and `whenStmt`, are different, it is obvious that the Scala description resembles the BNF definition.

D. Compilation

We employed Java as the intermediate code and avoided to directly compile the Sawzall code into Java byte code. One of the reasons of the design decision is to simply make the implementation easy. Another reason is that, since Sawzall programs are usually not so big, the increased compilation time is ignorable.

Figure 11 shows a part of the generated code for the word count code shown in figure 13. The compiled code implements `SCHelpers.Mapper` interface, which will be called from the mapper library of SawzallClone.

V. Evaluation

We have evaluated SawzallClone from several point of view; compilation time, comparison with programs with Hadoop native API, and comparison with szl. Furthermore, we compared parallel execution performance with program written in Hadoop native API.

A. Evaluation Environment

For the following evaluations, we used 16 nodes PC cluster each with two sockets of four core Xeon 5580 and 48 GB memories and SATA HDDs connected with 10Gbit Ethernet. We used Hadoop version 0.20.

B. Measurement of Compilation time

We have measured time to compile the word count script shown in figure 13. The whole compilation took 1.2 seconds including the Java compilation phase that took 0.7 seconds. Given that typical MapReduce computation lasts more than tens of minutes, the compilation time is ignorable.

C. Comparison with Hadoop Native Program

Here, we compare SawzallClone with Java program with Hadoop API. We employed the word counting program for comparison. Figure 13 and 14 show word counting programs in Sawzall and Java with Hadoop API, respectively. The former is in just 6 lines, while the latter is in about 60 lines. This proves the efficiency of Sawzall in writing scripts.

We compared the execution speed using the programs. As the input files we generated files with 69,906 lines, with 1 to 100 words in each line. The mapper is invoked on each line. It means that the loop in the script will iterate number of the words times for each invocation. The evaluation was performed with just one worker and reducer, since the goal of the experiment is to know the overhead of the Protocol Buffers. Note that the Sawzall version uses Protocol Buffers encoded file although it does not explicitly appears in the script, while the Java version uses plain text.

Figure 12 shows the result. The x-axis denotes number of words in each line and the y-axis denotes the time spent. Each experiment was performed 5 times and the average time is plotted in the graph. The error bar in the graph denotes the standard deviation of the value.

SawzallClone is substantially slower than the Java program. The overhead could be divided into Sawzall compiled code overhead and Protocol Buffers overhead. To divide these two, we modified the Java program so that it also uses Protocol Buffers. The result is also shown in figure 12. The difference between Sawzall and Java with Protocol Buffers is quite small, proving that the overhead mainly comes from Protocol Buffers.

D. Comparison with Szl

We compared SawzallClone with szl, the open source Sawzall implementation from Google. Since szl does not support parallel execution so far unfortunately, we compared them with sequential execution. For comparison, we employed the sample code appeared in the paper [14], shown in figure 6. We generated log files with 0.5, 1, 1.5, 2, and 2.5 million records in Protocol Buffers format and run SawzallClone and szl on them. [1]

The result is shown in figure 15. While SawzallClone is substantially slower than szl, note that the difference is constant and does not depends on the number of records in the file. This means that the execution speed of SawzallClone is comparable with szl and the difference comes from start up overhead, presumably due to Java VM initialization.

[1]The protocol buffer support in szl is intentionally turned off. We slightly modified the code to turn on the support for evaluation.

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

Fig. 11. Compiled Code.

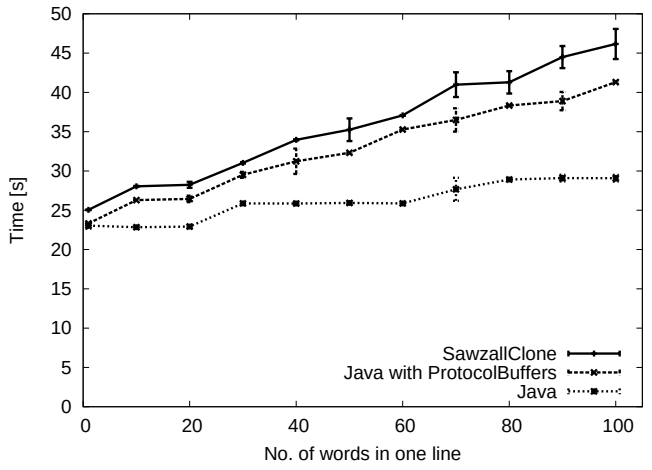

Fig. 12. Wordcount Comparison.

```
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;
}
```

Fig. 13. Word Count implementation in Sawzall

E. Evaluation with Parallel Execution

We have performed evaluation of parallel execution speed using Hadoop. We employed the log analysis program shown in figure 6. For comparison we wrote Java version of the program using Hadoop API. Note that we used Protocol Buffers for both of them.

We used 1 to 16 workers for mappers and reducers. As the target log files, use set up 64 log files, about 40MB each, replicated 3 times on HDFS.

The result is shown in figure 16. The x-axis denotes number of workers and the y-axis denotes elapsed time. Figure 17 shows the SawzallClone execution time normalized by the execution of the Java version.

The overhead imposed by SawzallClone is less than 10% for all the number of workers. Given that Sawzall makes implementation of the scripts easy, this overhead could be said quite small.

```
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 StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
 }

 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));
   }
 }

 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);
 }
}
```

Fig. 14. Word Count Implementation with Hadoop API.

VI. Conclusion

We implemented SawzallClone that executes Sawzall scripts on Hadoop. The system is implemented in Scala to leverage its Parser Combinator library to make the implementation easy. We evaluated the system comparing with programs written using native Hadoop API and confirmed that imposed

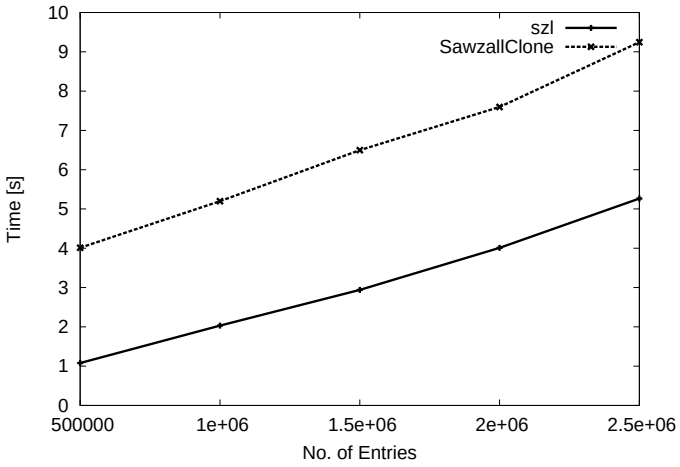

Fig. 15. Comparison with szl.

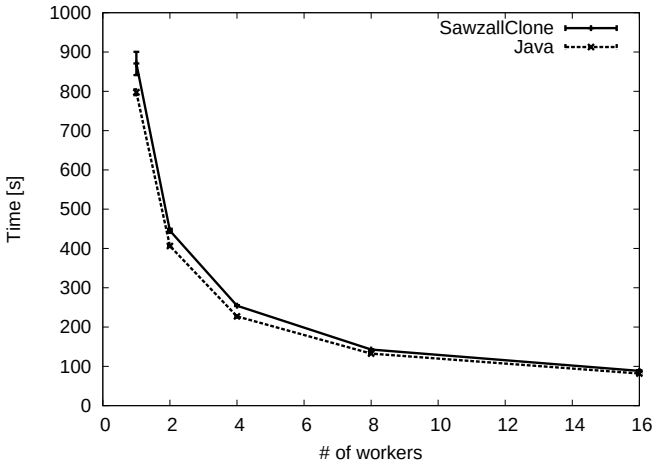

Fig. 16. Parallel Execution Comparison with Hadoop.

overhead by the language layer is small enough. We also compared it with Google's open source implementation, which is implemented in C++, and found that execution speed is comparable.

The followings are the future work:

- **Compatibility Improvement** SawzallClone has some compatibility issue with szl, the open source implementation from Google, including lack of function definition capability. We will fix the issue.
- **Adoption to other MapReduce systems** The system could be implemented on any MapReduce based system. We are developing a MapReduce system called SSS [12], based on distributed key-value store, which is designed to be faster than Hadoop. We will adopt SawzallClone to the system.
- **Improve Aggregator** While Sawzall is designed to allow optimized aggregator implementations [14], SawzallClone just provide naively implemented ones. We will optimize the aggregator implementation.
- **Improve Language Construct** Sawzall focuses on mappers only making program easy to understand and write. While this concept made it unique solution, apparently there is room to improve. We will extend the language so that it can also handle reducers, not only mappers.

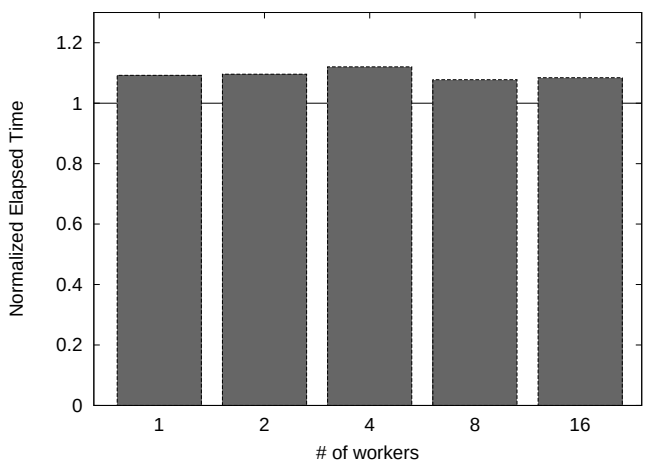

Fig. 17. Normalized Execution Time.

ACKNOWLEDGEMENT

This work was partly funded by the New Energy and Industrial Technology Development Organization (NEDO) Green-IT project.

REFERENCES

[1] Hadoop. http://hadoop.apache.org/.
[2] Hive. http://hive.apache.org/.
[3] jaql: Query Language for JavaScript(r) Object Notation. http://code.google.com/p/jaql/.
[4] Pig. http://pig.apache.org/.
[5] Protocol buffers - google's data interchange format. http://code.google.com/p/protobuf/.
[6] Scala. http://scala-lang.org/.
[7] Szl - a compiler and runtime for the sawzall language. http://code.google.com/p/szl/.
[8] A. V. Aho, Z. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.
[9] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. 2010.
[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.
[11] M. Odersky, L. Spoon, and B. Venners. Programming in Scala, Second Edition, 2010.
[12] H. Ogawa, H. Nakada, R. Takano, and T. Kudoh. Sss: An implementation of key-value store based mapreduce framework. In *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))*, pages 754–761, 2010.
[13] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD 2009*, 2009.
[14] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):227–298, 2005.
[15] I. Sun Microsystems. Xdr: External data representation standard. RFC 1014, June 1987.
[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE 2010: 26th IEEE International Conference on Data Engineering*, 2010.