Journal of Information Processing Vol.25 841–851 (Aug. 2017)

[DOI: 10.2197/ipsjjip.25.841]

Regular Paper

A Generator of Hadoop MapReduce Programs that Manipulate One-dimensional Arrays

Reina Miyazaki[1,a)] Kiminori Matsuzaki[2,b)] Shigeyuki Sato[2,c)]

Received: December 14, 2016, Accepted: February 17, 2017

Abstract: MapReduce is a framework for large-scale data processing proposed by Google, and its open-source implementation, Hadoop MapReduce, is now widely used. Several language systems have been proposed to make developing MapReduce programs easier, for instance, Sawzall, FlumeJava, Pig, Hive, and Crunch. These language systems mainly target applications that can be naturally solved by using a MapReduce-like programming model. In this study, we propose a new MapReduce-program generator that accepts programs manipulating one-dimensional arrays. By using the proposed generator, users only need to write sequential programs to generate Hadoop MapReduce programs automatically. We applied some program optimization techniques to the generation of Hadoop MapReduce programs. In this paper, we also report our experiment results that compare programs generated by the proposed generator with hand-written MapReduce programs.

Keywords: large-scale data processing, MapReduce, Hadoop MapReduce, program generator

1. Introduction

Efficient parallel programs on clusters are more difficult to develop than sequential programs on single-node computers because we have to analyze the feasibility of parallel computing and design an appropriate manner of parallel computing together with synchronization and communication of intermediate results.

MapReduce [6] proposed by Google offers a programming model for large-scale distributed data processing. Using MapReduce to implement data processing on clusters is considered to be easier than using traditional programming models such as MPI. Actually, Hadoop MapReduce [*1], which is an open-source implementation of MapReduce, is now in industrial use [*2]. Thus, MapReduce programming is widespread throughout the world.

To make it easier to develop MapReduce programs, many higher-level languages and systems such as Sawzall [17], FlumeJava [4], Pig [*3], Hive [*4], and Crunch [*5] were developed. Most of these programming models are so close to the MapReduce model that their target applications are often limited to ones that can be straightforwardly implemented on top of MapReduce. Thus, they expose the MapReduce model to programmers and have programmers be aware of the MapReduce model. When implementing applications that are far from the MapReduce model, it becomes much cumbersome. In this work, we have tackled this problem by studying MapReduce-oblivious programming, in particular, by generating Hadoop MapReduce programs from sequential programs. Our main contribution is that we have developed a source-code generator from loop-based sequential programs that manipulate one-dimensional arrays to efficient MapReduce programs.

The organization of this paper is as follows: in Section 2, we describe skeletal parallel programming, which is the conceptual basis of our generator; in Section 3, we describe the MapReduce model and outline Hadoop MapReduce, on which our generator is based; in Section 4, we show an overview of our generator through examples; in Sections 5 and 6, we describe the design and implementation of our generator; in Section 7, we experimentally evaluate the performance of MapReduce programs generated through our generator compared to that of hand-coded ones; we discuss related work in Section 8 and conclude this paper in Section 9.

2. Skeletal Parallel Programming

Parallel skeletons [5] are abstract computational patterns that often appear in parallel and distributed programming. Skeletal parallel programming is a methodology of parallel programming in which we select and combine parallel skeletons for the target problems.

In this chapter, we first review three parallel skeletons related to the proposed generator, map, reduce, and scan, with their parallel algorithms. Then, we introduce the diffusion theorem [11] that derives parallel programs by using those parallel skeletons. We will denote programs in Haskell [2] in accordance with the paper [11].

---

[1] Graduate School of Engineering, Kochi University of Technology, Kami, Kochi 782–8502, Japan

[2] School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan

[a)] 195071u@gs.kochi-tech.ac.jp

[b)] matsuzaki.kiminori@kochi-tech.ac.jp

[c)] sato.shigeyuki@kochi-tech.ac.jp

[*1] http://hadoop.apache.org/

[*2] https://wiki.apache.org/hadoop/PoweredBy

[*3] http://pig.apache.org/

[*4] https://hive.apache.org/

[*5] https://crunch.apache.org/

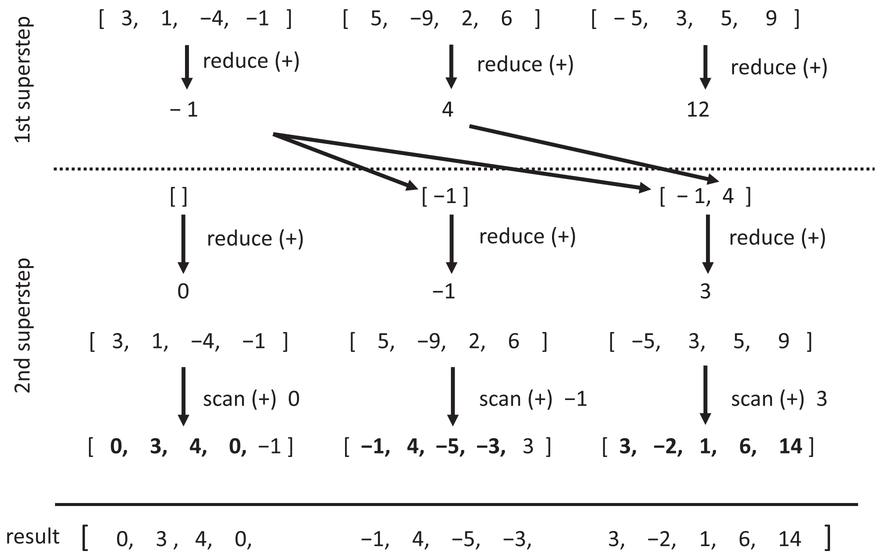


Fig. 1 The scanBSP algorithm.

2.1 Map

A parallel skeleton map applies a function k to each element of a list and is defined informally as follows.

$$map\ k\ [x_1, x_2, \ldots, x_n] = [k\ x_1, k\ x_2, \ldots, k\ x_n]$$

Parallel implementation of the map skeleton is simple. Let the input list be split into multiple sublists, and each processor independently applies function k to each element in the sublist.

2.2 Reduce

A parallel skeleton reduce joins the elements of a list and returns a value. The operator used in the reduce skeleton should be associative to ensure the correctness of parallel computation. Given an associative operator $\oplus$, the reduce skeleton is defined informally as follows.

$$reduce\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

The parallel implementation of the reduce skeleton consists of two steps. First, we join the elements of a sublist with the associative operator independently in parallel (local reduce). Then, we collect all the results on a processor and join them.

2.3 Scan

A parallel skeleton scan is an accumulative computation from the head of a list, and the result forms a list whose elements are the reduce of the prefix lists [*6]. Given an associative operator $\oplus$, the scan skeleton is defined informally as follows.

$$\begin{aligned} &scan\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] \\ &\quad = [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n] \end{aligned}$$

Note that the scan skeleton in this paper returns a list that has one more element than the input list has. We will use the unit of $\oplus$ for the initial value e unless otherwise mentioned.

There have been several algorithms for the parallel implementation of the scan skeleton. In this study, in accordance with the results of our prior work [15], we use an algorithm that is commonly used in the programming on the bulk synchronous parallel (BSP) model.

BSP is a model of parallel computation: computation consists of a sequence of supersteps separated by barrier synchronizations, and each process computes independently in a superstep. The scan algorithm in this paper, named scanBSP, computes in two supersteps. **Figure 1** illustrates how the scanBSP algorithm works, where the input list $[3, 1, -4, -1, 5, -9, 2, 6, -5, 3, 5, 9]$ is divided into three sublists. In the first superstep of the scanBSP algorithm, each process applies reduce computation to a sublist and sends the results to all the processes located on the right. In the second superstep, each process first applies reduce computation to the received values to compute the initial value of scan of the corresponding sublist and then computes scan locally starting from the initial value (the final element of the local scan would be dropped except for the last process).

2.4 Diffusion

Hu et al. [11] proposed the diffusion theorem stating that a recursive function over a list could be written as a combination of parallel skeletons if it satisfies some condition. The proposed program generator decomposes a sequential program into a combination of parallel skeletons based on the diffusion theorem and then generates a MapReduce program. Here, we briefly introduce only the theorem.

Theorem 1 (Theorem 1 in Ref. [11]) Let a function h over a list be written in the following form.

$$\begin{aligned} h\ [\,]\ c &= g_1\ c \\ h\ (a : x)\ c &= k\ (a, c) \oplus h\ x\ (c \otimes g_2\ a) \end{aligned}$$

If the operators $\oplus$ and $\otimes$ are associative and have unit $\iota_\oplus$ and $\iota_\otimes$, the function h can be decomposed as follows.

$$\begin{aligned} &h\ x\ c \\ &\ = \mathbf{let}\ cs' +\!\!+ [c'] = map\ (c\otimes)\ (scan\ (\otimes)\ \iota_\otimes\ (map\ g_2\ x)) \\ &\qquad\quad ac = zip\ x\ cs' \\ &\quad\ \mathbf{in}\ reduce\ (map\ k\ ac)\ \oplus\ g_1\ c' \end{aligned}$$

[*6] Therefore, it is also called prefix sums.

The keypoint of the theorem is: even if the computation on an element depends on its forward elements (through the accumulation parameter c of function h), we can compute in parallel by applying the `scan` skeleton to yield those accumulation parameters.

3. MapReduce

MapReduce [6] is a framework and also a programming model for distributed parallel processing proposed by Google. In this chapter, we review the MapReduce programming model. An important characteristic of it is that the process consists of two phases called Map and Reduce. The computation in the Map and Reduce phases is specified by the user. **Figure 2** illustrates the programming model.

In MapReduce, the input data are split and stored on a distributed filesystem (each set of split data is also called a *split*), and those split data are fed into the computation. In the Map phase, each Map task performs computation on a split and outputs key-value pairs as intermediate results. Then, the framework sorts the results of all the Map tasks to group them by their keys. In the following Reduce phase, each Reduce task computes on a set of intermediate results grouped by their keys (or a user-specified condition). The results of the Reduce phase are output as the final results. It is worth noting that the tasks in the Map and Reduce phases are executed independently of each other. To summarize, MapReduce provides a simple programming model but limits the computation that can be performed on it.

Hadoop MapReduce is a distributed-processing framework based on the MapReduce programming model. An important feature of Hadoop MapReduce is that we can develop a large variety of programs with many customization parameters and/or runtime parameters.

The customization provided in Hadoop MapReduce includes the `FileInputFormat` class that specifies the manipulation of an input split, secondary sorting to control grouping and sorting, extension of the `Writable` class, and so on. We observed in a prior work that the extended `Writable` class for sending data in bulk improved the performance when we sent a sequence between Map/Reduce tasks or between MapReduce jobs [15].

Hadoop MapReduce has hundreds of runtime parameters including the number of Map/Reduce tasks, upper bound of memory usage, condition of starting the Reduce phase, and size of virtual memory space[*7]. We observed in a prior work that the performance could be improved by setting appropriate values to the condition of starting the Reduce phase and the number of Reduce tasks [15].

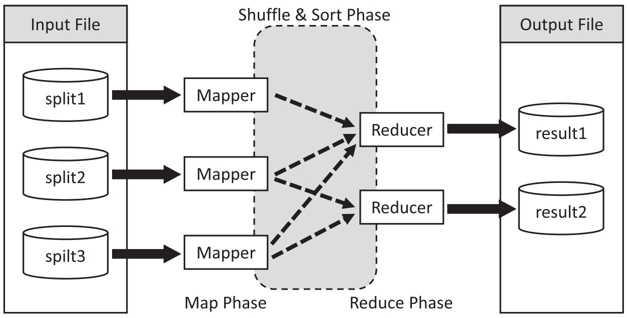


Fig. 2 Computation model of MapReduce.

4. Target Applications

The proposed program generator takes a sequential program that manipulates a one-dimensional array as its input and generates a Hadoop MapReduce program. In this chapter, we specify the input programs that the program generator accepts. We then show two examples of the target applications: log filtering and maximum number of concurrent users.

4.1 Specification of Input Programs

Table 1 shows the syntactic rules of the input programs.

We basically borrow the syntactic rules of Java to define those of the input program of the proposed generator. The entry point of the program is specified by the `main` function, and we can define other functions used in a program.

To specify the input/output data, we need to specify the types `Input<Type>` and `Output<Type>`, respectively, in the `main` function. The input data from the user will be stored in the variable declared with type `Input<Type>`. The output data in the variable declared with type `Output<Type>` will be output into a file as the results. These variables should be described in the `main` function.

In a program, we can describe multiple loops that scan the input one-dimensional array sequentially from the head. Those loops should be described in the `main` function with for statements, where the proposed generator requires the following conditions.

(1) Only unnested loops are available in the `main` function.
(2) The loop variable should have type `int` and name `i`.
(3) The range of the variable should be from `0` to the size of input array `input_array.length-1`.

In addition to loops that apply independent computation to the elements, we can describe loops that accumulate the values from the head of the array. To describe such accumulations, we should use a statement in the following form to update an accumulation variable.

$$\langle\text{acc. variable}\rangle = \langle\text{acc. variable}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle$$

Here, we accepts associative binary operators (“`+`” and “`*`”) and corresponding inverse operators (“`-`” and “`/`”, where “`/`” is not available for integers) for the operator above. For the computation that accumulates the maximum or minimum value, we extract it separately from an if statement that does not satisfy this condition.

We now show two examples that satisfy the specifications of the input programs. In the following chapters, we will use a log-filtering example (Section 4.2) as our running example to see how the programs are transformed.

[*7] Those hundreds of runtime parameters were introduced over the history of Hadoop MapReduce. The latest version of Hadoop MapReduce is 2.7.x, and the internal processing mechanism and the job scheduler changed from version 1.x.x to 2.x.x. Version 2.x.x has a lot of runtime parameters to keep backward compatibility, and many of them would have a large effect on the execution or performance.

Table 1 Syntactic rules of input program (in EBNF).

```
program = ‘{’ , main , {function_def} , ‘}’ ;
main = ‘void’ , ‘main’ , ‘()’ , ‘{’ , input_decl , {var_decl} , output_decl , for_statement , ‘}’;
function_def = ? Function definition of a pure function interpreted as Java static method ?;
input_decl = ‘Input’ , ‘<’ , base_type , ‘[]’ , ‘>’ , input_identifier , ‘;’ ;
output_decl ::= ‘Output’ , ‘<’ , base_type , [‘[]’] , ‘>’ , identifier , ‘=’ , literal , ‘;’ ;
base_type = ‘int’ | ‘boolean’ | ‘double’ | ‘String’ | ... ;
var_decl = ? Declaration of a Java local variable ?;
for_statement ::= ‘for’ , ‘(’ , ‘int’ , ‘i’ , ‘=’ , ‘0’ , ‘;’ ,
                                        ‘i’ , ‘<’ , input_identifier , ‘.’ , ‘length’ , ‘;’ ,
                                        ‘i’ , ‘++’ , ‘)’ , ‘{’ , {statement} , ‘}’ ;
statement = (simple_assign | acc_assign | java_expr) , ‘;’ | if_statement ;
simple_assign = identifier [ ‘[’ , ‘i’ , ‘]’ ] , ‘=’ , java_expr ;
acc_assign = min_max | identifier , ‘=’ , identifier , assoc_op , java_expr ;
if_statement = ‘if’ , ‘(’ , <java_expr> , ‘)’ , ‘{’ , {statement} , ‘}’ ,
               [ ‘else’ , ‘{’ , {statement} , ‘}’ | ’else‘ if_statement ];
java_expr = ? Expression interpreted in Java ?;
assoc_op = + | - | * | ... (* associative operators and corresponding inverse operators *);
min_max = ? if-statement that compares two variables and assigns either of them to the other ?;
input_identifier = identifier ;
identifier = ? identifier interpreted in Java ?;
```

4.2 Log Filtering

Let us extract logs of connection failures (the logs include “false”) from a large amount of logs, where we would like to filter one from every ten occurrences. The following is a sequential program that solves this log-filtering problem.

```
void main() {
  Input<String[]> logs;
  int count = 0;
  Output<String> output = "";
  for (int i = 0; i < logs.length; i++) {
    if (cond(logs[i])) {
      count = count + 1;
      if (count % 10 == 0) {
        output = output + logs[i];
      }
    }
  }
}
boolean cond(String val) {
  return val.indexOf("false") >= 0;
}
```

In this program, the variable `logs` denotes the input list of logs, and the variable `output` denotes the user’s output. The variable `count` denotes the number of “false” logs up to the position.

4.3 Maximum Number of Concurrent Users

Let us extract the logs of logins (lines including “login”) and logouts (lines including “logout”) and compute the maximum number of concurrent users who are online at a time. The following is a sequential program that solves this problem.

```
void main() {
  Input<String[]> logs;
  int count = 0;
  Output<int> max = 0;
  for (int i = 0; i < logs.length; i++) {
    if (isLogin(logs[i])) {
      count = count + 1;
    } else if (isLogout(logs[i])) {
      count = count - 1;
    }
    if (count > max) {
      max = count;
    }
  }
}
boolean isLogin(String val) {
  return val.indexOf("login") >= 0
}
boolean isLogout(String val) {
  return val.indexOf("logout") >= 0
}
```

This program processes each element of the input `logs` in order, and increments or decrements the variable `count` when the element includes “login” or “logout,” respectively. Thus, the variable `count` denotes the number of concurrent users at the position. The program calculates the maximum value of `count` with an if statement and stores it in the variable `max`.

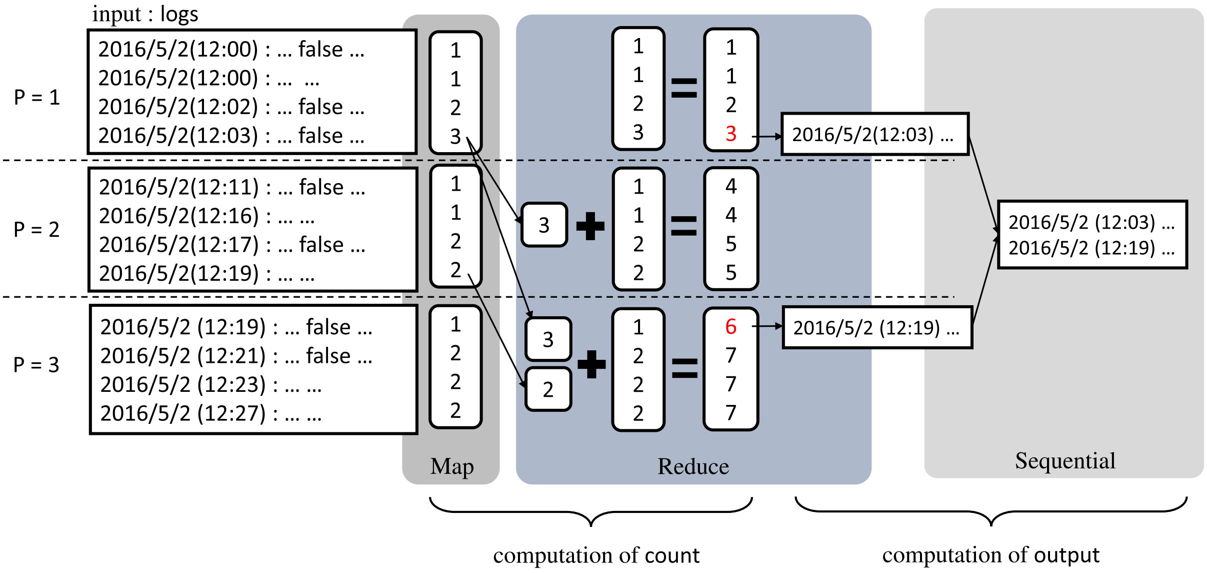
5. Compilation Pipeline

The proposed program generator takes a sequential program that manipulates a one-dimensional array (Section 4) and automatically generates a parallel program that runs on Hadoop MapReduce. In this chapter, first, we briefly show the execution of the output program generated from the input program in Section 4.2 and then show the compilation pipeline of the generator.

5.1 Execution of Input and Output Programs

In this section, we show the execution of the Hadoop MapReduce program generated from the input program in Section 4.2. **Figure 3** gives the outline of the execution of the output program. (For simplicity, one “false” log is extracted for every three occurrences in Fig. 3. One log was extracted for every 10 occurrences in Section 4.2.)

In the sequential program for the log-filtering problem, we compute the values of `count` and extract a log to `output` simultaneously. We can compute `count` and `output` simultaneously in sequential processing, but in parallel processing we cannot filter logs until we know the number of forward “false” logs because the input data are split and processed independently in parallel. Therefore, in the generated program in Fig. 3, we first compute `count` (the number of “false” logs) for all the elements, share the results among the split data to obtain the `count` values for the



NB. In the paper "false" logs are extracted once for every 10 occurrences, while once for every 3 occurrences in this figure.

Fig. 3 Computation of Hadoop MapReduce program for log-filtering problem.

whole data, and then filter the logs to `output`.

As we can see in Fig. 3, the first Map phase counts the "false" logs for each split of the input data. This value is closed in each split. Then, the number of "false" logs is sent to all the processes that will manipulate split addressing later, and each process obtains the number of "false" logs before the corresponding split. The following Reduce phase recomputes the values of `count`, which are correct over the whole data. By using the values of `count`, the program selects the logs to be extracted. Finally, a sequential program joins the logs extracted by the processes to obtain the final result. What we have described so far is the outline of the Hadoop MapReduce program generated for the log-filtering problem.

The proposed generator takes a file containing an input program and outputs multiple files for Hadoop MapReduce programs and a sequential program executed at the last step if needed. The user then compiles the generated Hadoop MapReduce programs using the commands in Hadoop 2.7.1 to generate a jar file, puts the input data on the HDFS, and executes the jar file with the parameters specifying the input/output directories. If there exists a sequential computation step as in Fig. 3, the user compiles the sequential program with the Java compiler and executes it with a parameter specifying the directory to which the Hadoop MapReduce programs output the results.

5.2 Steps of Program Translation

A sequential program manipulating a one-dimensional array is translated into a parallel program on the Hadoop MapReduce in the following steps.

(1) Analyze input program.
(2) Decompose loop structures based on dependency among variables.
(3) Translate to MapReduce algorithms.
(4) Generate Hadoop MapReduce programs.

In the first step, we parse the input program and generate an abstract syntax tree (AST). We then analyze the program to extract data dependencies among the variables. We use existing techniques for this analysis and briefly explain them later.

In the second step, we decompose a loop containing dependent (not parallelizable) statements into multiple parallelizable loops. For example, as we have seen in Section 5.1, the parallel program for the log-filtering problem should compute `count` for all the elements first and then compute the `output` using the results. The loop in the input program that computes `count` and `output` simultaneously is divided into two loops: one for `count` and the other for `output`.

In the third step, to compose MapReduce jobs, we assign a Map phase and/or a Reduce phase for each of the loops decomposed in the second step. Then, we optimize the MapReduce jobs by rearranging those phases.

Finally, in the fourth step, we generate Hadoop MapReduce programs for the MapReduce jobs obtained in the third step. We use the efficient programs studied in our prior work [15] as the template for code generation.

5.3 Analysis of Input Program

We use existing common techniques to analyze the input program. Here, we briefly review them.

To prepare to translate to the Hadoop MapReduce programs, we extract control dependencies and data dependencies. Consider a list of input data is split and computation is applied independently to the splits. We cannot compute correctly if the computation depends on the value in other (forward or backward) splits. In the log-filtering example, we cannot judge whether a "false" log is 10th unless we know the number of "false" logs in the forward splits. To obtain a parallel algorithm from a program that sequentially scans a one-dimensional array, we need to perform the computation with dependency (such as `count`) in advance. We extract control dependencies and data dependencies from the input program to judge the computation.

Control dependency "statement *t* depends on condition *s*" means that condition *s* determines whether statement *t* is executed [8]. For example, the following program has a control dependency "the statement in line 2 depends on the condition in line 1".

```
1: if (a == 0) {
2:   b = b * 10;
3: }
```

Data dependency "statement *t* depends on statement *s* through variable *w*" means that variable *w* is defined at statement *s* and is used at statement *t* [8]. For example, the following program has a data dependency "the statement in line 2 depends on the statement in line 1 through variable `a`."

```
1: int a = 0;
2: int b = a + 4;
```

In this study, we extracted control dependencies and data dependencies from the input program. We did not deal with circular dependencies over multiple variables.

6. Translation to MapReduce Programs

In this chapter, we show how we translate an input sequential program to Hadoop MapReduce programs.

6.1 Decomposing Loop Structures Based on Dependencies among Variables

As we saw for the computation of `count` in the log-filtering example, the computation depending on forward input elements can be easily described by a sequential program but cannot easily be described by a parallel program. In particular, the computation of `output` depends on the values of `count`, and it requires the number of "false" logs until each position over the whole input data. In the sequential program, we described the computations of `count` and `output` in a loop. In the parallel computation, we required two steps of computation: computing `count` for each split and then sharing the partial results among splits. By using these partial results (numbers of "false" logs), we could compute `output` to obtain the same results as the sequential program. As we have seen, to compute problems similar to the log-filtering problem in parallel, we need to decompose loops so that all the computations with forward dependency are executed in advance.

Such computations with forward dependency are formalized as accumulation over lists in functional programming, and the `scan` skeleton is used as the parallel computational pattern for those computations. The diffusion theorem [11] in Section 2 transforms a linear recursive function over lists to a parallel program by extracting the `scan` skeleton. The proposed generator decomposes loops based on the diffusion transformation with the dependencies obtained from the analysis of the input program. In the following part of this section, we demonstrate how the generator transforms the example program based on the diffusion transformation.

6.1.1 Decomposition of Loops Based on Computation Order

The diffusion theorem transforms a computation over lists described in a certain form into a combination of three parallel skeletons: `map`, `reduce`, and `scan`. The basic idea of the theorem is that if some computation depends on forward elements and the values are used in later computation (like `count` in the log-filtering problem), we compute the values in advance over the whole input data.

The proposed generator finds the variables and their computation order by utilizing the dependencies among variables as follows.

(1) Extract all the statements that update variables from a loop.
(2) Compute the dependencies for each statement.
(3) Extract statements that do not depend on other statements.
(4) Create a loop structure for the statements extracted in step 3 with the program-slicing technique.
(5) Remove the statements extracted in step 3 and go back to step 2. Finish if all the statements have been extracted.

We demonstrate these procedures with the input program in Section 4.2. First, we extract all the statements that update variables; we obtain the statement in line 7 (updating `count`) and the one in line 9 (updating `output`). We compute the dependencies for the two statements. The statement in line 7 (updating `count`) does not depend on the other statement, but the statement in line 9 (updating `output`) depends on the value of `count` through the branch in line 8. With these pieces of information, we now know that the values of `count` should be computed in advance.

Then, we create a loop structure to compute the values of `count` in advance. To do this we use the program-slicing technique; more concretely, we extract the statements or conditions that the statement depends on. For the statement in line 7 (updating `count`), we extract the for statement in line 5 and the if statement in line 6 and construct a new loop for computing the values of `count`. Since the remaining statement (updating `output`) refers to the value of `count` at each step, it is necessary to store the values of `count` as an array. Therefore, we convert the variable `count` to an array and put the unit of +, which is `0`, as the initial value in the array. The following is the result of program slicing for the statement updating `count`.

```
int[] count;
count[0] = 0;
for (int i = 0; i < logs.length; i++) {
  if (cond(logs[i])) {
    count[i+1] = count[i] + 1;
  } else {
    count[i+1] = count[i];
  }
}
```

After removing the statements related to `count`, we go back to step 2. Since all the remaining statements are related to `output`, we rearrange those statements to form a loop. As we did for `count`, we put the unit of the updating operator + for `output`, which is `""`, as the initial value. The following is the result of the transformation.

```
Output<String> output = "";
for (int i = 0; i < logs.length; i++) {
  if (cond(logs[i])) {
    if (count[i+1] % 10 == 0) {
      output = output + logs[i];
    }
  }
}
```

6.1.2 Unification of Variable-Updating Statements

To apply the diffusion theorem, we need to linearize the recursive calls, that is, we remove the if branches to unify the recursive calls into a single call. In this study, we only deal with loops not general recursive functions; the target statements to be linearized are those updating variables in the loops.

Here, we look at the loop for the variable `count`. When `cond(logs[i])` is `true`, the value of `count` is incremented. When it is `false`, the value of `count` does not change. Therefore, by writing down all the update statements in the branches, we obtain the following statements.

```
count[i + 1] = count[i] + 1
count[i + 1] = count[i]
```

The goal here is to represent all the statements as a single statement. We move the if statements into functions so that the different parts are managed by the returned value of the functions. Before moving the if statements into functions, we derive a unified form of those two statements as follows.

```
count[i + 1] = count[i] + 1
count[i + 1] = count[i] + 0
```

We can perform this transformation by searching the same references or operators in the corresponding subtrees in the AST. Here, we use the subtree corresponding to the first statement `count[i + 1] = count[i] + 1` as the initial template and match it to the subtree corresponding to the second statement `count[i + 1] = count[i] + 0`. We begin the matching at the self-reference to the updated variable (`count[i]`). We compare the AST nodes for operators. If the operators differ, we add operators and the missing operands (the unit of the operator) to the template, considering the priority of the operators. We continue this process until all the statements are in the form of the template.

With the transformation so far, we know the position of values that may differ in the if branches. For each of such values, we create a function that includes the if statement of the original program and returns the (possibly) different values. After creating a function and replacing a value with a function call, we obtain the following program.

```
int[] count;
count[0] = 0;
for (int i = 0; i < logs.length; i++) {
  count[i + 1] = count[i] + func0(logs[i]);
}
int func0(String val0) {
  if (cond(val0)) {
     return 1;
  }
  return 0;
}
```

The function `func0` shown above is created from the body of the original loop, where updating statements are replaced with statements returning the corresponding values. Note that we add the parameter `String val0` since it is used as the condition of the if statement.

By applying the program transformations above, we can unify the statements that update the same variable in a loop. Note that those program transformations are variants of those applied to recursive functions in the original diffusion transformation.

We expect that the programs obtained after decomposition of loops are computed in parallel. The remaining task is to extract associative operators for the scan and/or reduce skeletons. Since we limited the updates of accumulation values only with associative binary operators, we just extract those operators from the AST nodes next to the self-reference to the variable in the RHS of the statements.

For example, let the statement be `count = count + 1`; the operator next to the self-reference to `count` in the RHS is +, which is used in the computation of the corresponding reduce and/or scan skeleton. If the statement is `count = count - 1`, we use the operator "+" instead of the operator "-". Similarly, for the operators "/" and "*" in the programs, we use "*" for the reduce and/or scan.

The loop for the variable `output` is transformed as follows.

```
Output<String> output = "";
for (int i = 0; i < logs.length; i++) {
  if (cond(logs[i])) {
    if (count[i+1] % 10 == 0) {
      output = output
               + func1(logs[i], count[i+1]);
    }
  }
}
String func1(String var0, int var1) {
  if (cond(val0)) {
    if (val1 % 10 == 0) {
      return val0;
    }
  }
  return "";
}
```

The proposed generator outputs the implementation of operators. The following is the program corresponding to the operators for `count` and `output`.

```
// operator for composing count
int countOp(int val0, int val1){
  return val0 + val1;
}
// operator for composing output
String outputOp(String val0,
                String val1){
  return val0 + val1;
}
```

We can obtain the decomposed program by performing the transformations so far:

(1) decomposition of loops,
(2) unification of variable-updating statements, and
(3) identification of operators.

Although we had the fourth step that optimized the operators in the original diffusion transformation, we have not implemented it in the proposed generator.

6.2 Transformation to MapReduce Algorithms

With the transformations in Section 6.1, we decomposed loop structures so that we can compute them in parallel. In this section, we show how we assign Map and/or Reduce phases for each loop structure. We also optimize the structure of phases to improve the performance on MapReduce.

6.2.1 Composing MapReduce tasks for Loop Structures

We compose MapReduce tasks for each of the two loop structures obtained in Section 6.1.2. We show the target loop structures again for reference.

```
01: int[] count;
02: count[0] = 0;
03: for (int i = 0; i < logs.length; i++) {
04:   count[i + 1] = count[i] + func0(logs[i]);
05: }
06: int func0(String val0) {
07:   if (cond(val0)) {
08:      return 1;
09:   }
10:   return 0;
11: }
```

```
01:  Output<String> output = "";
02:  for (int i = 0; i < logs.length; i++) {
03:    if (cond(logs[i])) {
04:      if (count[i+1] % 10 == 0) {
05:        output = output
                  + func1(logs[i], count[i+1]);
06:      }
07:    }
08:  }
09:  String func1(String var0, int var1) {
10:    if (cond(val0)) {
11:      if (val1 % 10 == 0) {
12:        return val0;
13:      }
14:    }
15     return "";
16:  }
```

6.2.2 Application to MapReduce

First, we categorize the loops based on the computation in them. The first loop for `count` is the computation of the scan skeleton because the computation of `count[i+1]` uses the value in the previous step `count[i]` and `count` has values accumulated from the head of the array. If the update of a variable depends on its previous value, the computation in the loop is either reduce or scan; the computation is scan if the accumulated values are stored in an array. If the statement is `count = count + 1`, for example, then the computation is reduce.

Consider the second loop for `output`. Looking at line 5, the variable `output` is updated with the value of itself. Thus, the computation in the loop is either reduce or scan. Since the result is stored in a single variable `output`, the computation in the second loop is reduce.

We have categorized the computation in the loops. We now compose MapReduce tasks for the computation in each loop.

Let us compose MapReduce tasks for the first loop (scan). Figure 1 illustrates the computation of the scan skeleton. The first superstep above the dashed line (including the judgment of "false" logs with the cond function) can be implemented in a Map phase. The arrows crossing the dashed line correspond to the Shuffle&Sort phase after the Map phase. The number of "false" logs in a split is sent to the later splits, and then all the splits can compute the total number up to the split. By using this total number of "false" logs and the list of marks of "false" positions, we compute the scan in the split to obtain the values of `count`. The computation below the dashed line is implemented in a Reduce phase. Therefore, computation of a scan skeleton is implemented in single Map and single Reduce phases (as in this example of `count`).

Let us compose MapReduce tasks for the second loop (reduce). The computation of the reduce skeleton consists of two steps: the values in each split are joined with an associative operator into a single value and the resulting values from all the splits are then collected and joined. For the target loop, the first step is to take a split of `logs` and filter logs based on the values of `count`. This first step is implemented in the Map phase. The results of `output` are transferred to a single process and concatenated. The transfer of data is implemented in a Shuffle&Sort phase, where a single key is used to collect data. The final join is implemented in the Reduce phase. Therefore, computation of the reduce skeleton is also implemented in single Map and single Reduce phases.

6.2.3 Optimization of MapReduce Tasks

As we stated in the previous sections, the proposed generator first decomposes a loop so that we can compute it in parallel and then composes MapReduce tasks for each of the decomposed loops. This means that the more loops we have after decomposition the more MapReduce tasks we have. The MapReduce framework brings a large cost for starting up the MapReduce jobs, starting up the Map or Reduce phases, and the input and output. Therefore, it is important to minimize the number of MapReduce jobs/tasks for good performance. In this section, we show how we optimize MapReduce tasks with our running example.

As we have seen in Section 6.2.2, the log-filtering problem was implemented as a two-pass MapReduce program: the first pass is for the scan skeleton for `count`, and the second pass is for the reduce skeleton for `output`. Here, we focus on the Reduce phase in the first pass and the Map phase in the second pass. In the Reduce phase in the first pass, we compute the numbers of "false" logs and store them in `count`. Note that the computation in the following Map phase, joining logs locally based on the values of `count`, can be executed simultaneously in the same Reduce phase. With this fusion, the result of the first MapReduce pass is joined logs (`output`) for each split. The remaining task is to join these results. Note that the number of results after the first MapReduce pass is just the number of splits, and we expect it to be small enough. Therefore, the join of the results could be implemented as a sequential program, which does not have the overhead of the MapReduce framework. With these transformations, the program for the log-filtering problem is optimized from a two-pass MapReduce program to a one-pass MapReduce program with a sequential program.

6.3 Generating Hadoop MapReduce Programs

Finally, we generate Hadoop MapReduce programs based on the MapReduce algorithms. As we briefly reviewed in Section 3, we proposed an efficient implementation [15] for the programs obtained by the diffusion transformation. In this study, we use those implementation techniques as templates for generating Hadoop MapReduce programs.

The proposed generator outputs the following six files.

DataFunction.java analyzes the input split and converts it into an array. In the current implementation, we expect that a single element is given in a line.

Function.java defines the functions used in the map, reduce, and scan skeletons. These functions are generated from the transformed programs where the types of variables are changed to `Writable` types on Hadoop MapReduce.

HadoopMainProg.java is the main program executed on Hadoop MapReduce. This program starts up MapReduce jobs and controls the directories for input and output files.

MRSCode.java implements `Mapper` and `Reducer` functions for the map, reduce, and scan skeletons. The structure of tasks follows that in Section 6.2. This program also implements functions that generates MapReduce jobs (these functions are called in `HadoopMainProg.java`.).

CustomWritable.java provides a customized `Writable` class used for transferring the input array and/or intermediate results of map and scan skeletons (Hadoop MapReduce requires us to implement a customized `Writable` class if we want to transfer structures or values with different types between phases).

GlobalReduce.java joins the intermediate results of reduce obtained from the MapReduce jobs for splits. It takes the directory where the intermediate results are stored.

The following programs are used in the above programs.

CustomPartitioner.java, **CustomGroupingComparator.java**, **CustomSortComparator.java**, and **CustomKeyWritable.java**: These programs implement the secondary-sort mechanism on Hadoop MapReduce. Since the input array is order sensitive, we use the secondary-sort mechanism to transfer data between the Map and Reduce tasks keeping the order of splits.

SplitFileInputFormat.java and **SplitFileRecordReader.java**: These programs are used to split the input data by the specified size and execute a Map task for each split. In this study, we assume that the elements of the input array are stored in a file as one element in a line. These two programs support each Mapper to take the corresponding input split as a list. In these programs, `LineReader` reads the lines until the amount of read data reaches the specified size and puts the elements into a `List`. Then, a Mapper starts with a key-value pair where the key is the position in a file and the value is the `List`.

7. Experiments

In this chapter, we evaluate our program generator experimentally by comparing the performance of a generated program with that of a hand-optimized one for solving the log-filtering problem described in Section 4.2.

7.1 Experimental Setting

We used a 16-node cluster that consisted of 9 nodes with Intel® Core$^{\mathrm{TM}}$ i5-2500 (3.30 GHz) and 7 nodes with Intel® Core$^{\mathrm{TM}}$ i5 CPU 760 (2.80 GHz); all nodes were equipped with 8-GB memory (DDR3-1333) and 500-GB SATA HDD (7,200 rpm) and connected with 1000BASE-T. We used Hadoop 2.7.1 and JDK 1.8.0.

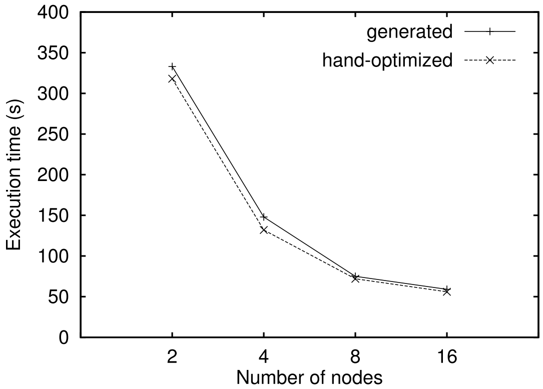


Fig. 4 Execution time of MapReduce job for log-filtering problem.

Table 2 Execution time (in seconds) plotted in Fig. 4.

number of nodes	2	4	8	16
generated	333	148	75	59
hand-optimized	318	132	72	56

Table 3 Average execution time of Map, Shuffle, and Reduce tasks.

	generated	hand-optimized
Map	12	11
Shuffle	40	36
Reduce	26	4

The input file was 100 million lines of 6-GB random text biased such that about 20% lines contained `false`. A line of the file corresponded to an element of the input array `logs`. The output file was 2 million lines of 2% of the input file.

Regarding the parameters of Hadoop MapReduce, we set the split size to 128 MB. The number of Map tasks created for this split size was 47. As in the setting of our prior work [15], we tuned other parameters so that Reduce tasks started to run after 90% of the Map tasks ended. The number of cores in the CPUs was four and we limited the number of Map tasks and the number of Reduce tasks to two.

7.2 Experimental Results

Figure 4 and **Table 2** show the execution time of a program generated through our system and a hand-optimized program. These execution times include only the time for the single MapReduce job of solving the log-filtering problem and exclude that for the following sequential concatenation of distributed `output` values.

Table 3 shows the averages of each execution time of Map, Shuffle, and Reduce tasks with 8 computing nodes. The Map phase took about 3 times the average Map task time because we had 47 Map tasks. Note that Map tasks and Reduce tasks ran in parallel, and the results in Table 3 do not include the waiting time.

7.3 Evaluation

As seen from Fig. 4, the generated program had almost the same performance as the hand-optimized one. This is because there the only difference of code between them was whether if statements formed functions. The optimization of our system brought the same construction of MapReduce jobs as the hand-optimized one. This has demonstrated that our system can generate efficient programs that consist of map, reduce, and scan such as the one for solving the log-filtering problem.

8. Related Work

The advantages of MapReduce as a data processing tool are summarized as the following three points [7]: fine-grained fault tolerance of large-scale jobs; making data processing on heterogeneous distributed storage systems easy; enabling us to unrestrictedly call complex functions that are not given in SQL in databases. An offset of this unrestricted nature in programming is the cumbersome programming of implementing even common useful functions, which SQL natively supports, through the Map and Reduce APIs. Therefore, systems that generate MapReduce programs from higher-level languages and APIs were developed to enable lightweight programming while preserving the first and second advantages.

Sawzall [17] and Pig [*8] were procedural domain-specific language systems for ad hoc log analysis. Pig was designed to enable us to specify the pipelines of primitives of relational databases. Sawzall was designed to enable us to describe aggregate computations briefly through for loops. Sawzall is similar to our system in the sense that for loops are central to language design. However, our target loops iterate over one-dimensional arrays, while Sawzall's loops iterate over sets or collections. Therefore, Sawzall did not deal with order-sensitive computations that scan can represent.

FlumeJava [4] was a Java library that offers a programming framework equipped with both pipelines of data processing and loops over collections. Since FlumeJava was a Java library, it was designed for more general-purpose use. As Sawzall, FlumeJava did not deal with computations on one-dimensional arrays. As its main feature, FlumeJava was equipped with optimizations by dynamically transforming graphs of pipelines. Our optimization described in Section 6.2.3 is orthogonal to the ones of FlumeJava.

Although Sawzall and FlumeJava did not deal with one-dimensional arrays, it is possible in principle to address our target computations on top of them. Because they were able to handle tables (i.e., key-value collections), we can encode order-sensitive computations on arrays into the ones on tables through tricky programming on keys. This tricky programming of encoding causes cumbersome low-level MapReduce programming, which we would have liked to avoid. The main contribution of our system is to provide both the advantages of MapReduce as a data processing tool and high-level programming for computations on one-dimensional arrays.

Liu et al. [13], [14] developed two list skeleton libraries on top of MapReduce; one [14] was based on list homomorphism [3], which is a composition of map and reduce, and the other [13] was based on accumulate [12], which is a composition of scan, map, and reduce. These libraries assumed that the user gave appropriate operators satisfying the algebraic conditions required by these skeletons. In contrast, our system is designed to take sequential loops described without caring about algebraic conditions and parallelize them as much as possible. Thus, ours burdens the user less.

A recent representative study on making MapReduce programming easier was a tool developed by Smith and Albarghouthi [19] for synthesizing MapReduce programs from given input-output examples on the basis of search and verification. Their approach first prepares a predefined set of data-parallel skeletons and operator templates and then extends it gradually with their compositions while verifying whether generated instances satisfy the specifications of given input-output pairs; eventually the desired programs are obtained through this iteration. Our system also generates programs after composing skeletons. However, they limited targets to MapReduce programs containing aggregation with associative commutative operators. Thus, they did not deal with accumulative computations like scan. Their approach assumed that the whole of a generated program would be a composition of skeletons. Our system extracts loops to be converted to skeletons from a given loop and thus can generate a best-effort parallel program even when we cannot represent the whole as a composition of skeletons.

This work has an aspect of automatic parallelization, which has a long history and was originated from PTRAN [1]. In fact, program dependence graphs [8], which we used in this work, are commonly used in the context of automatic parallelization. A recent study on automatic parallelization was performed by Fonseca et al. [10]. Their compiler parallelized general Java programs by extracting task parallelism with dependence analysis. It was able to parallelize recursive functions straightforwardly because the generated programs were task-parallel programs assuming work-stealing schedulers. However, parallelization focusing on task parallelism is not appropriate for selectively parallelizing large-scale data processing that we would like to use with MapReduce. In this work, we have focused only on massive data parallelism for MapReduce. Therefore, by limiting targets to loops over one-dimensional arrays, we developed a tool capable of generating efficient MapReduce programs.

Prominent work on extracting reduce and scan in the context of automatic parallelization was performed by Fisher and Ghuloum [9]. Their approach formalizes a given loop body as a function and parallelizes it by discovering a computation for constructing its composite functions. They succeeded in parallelizing loops containing complicated conditionals by applying heuristic search and pruning it. Their idea of parallelization was formalized as quantifier elimination and generalized to cover recursive functions by Morihata and Matsuzaki [16]. Sato and Iwasaki [18] presented a lightweight method based on Fisher et al.'s paper [9] that formalizes a given loop body as matrix multiplication over semiring after extracting maximum operators. Complicated loops of reduce and scan that these studies dealt with are not in the scope of our system. A parallelization strategy of extracting loops to be converted to skeletons, which we have adopted, has been presented [18]. In this sense, our contribution to this work is not on automatic parallelization techniques. In fact, our implementation of diffusion transformation [11] is so straightforward that it has made no technical extension. Our contribution of this work is, as mentioned before, tool development based on existing tech-

---

[*8] http://pig.apache.org/

niques for making MapReduce data processing easier.

9. Conclusion

In this paper, we have presented a system that takes an input program manipulating a one-dimensional array and generates an efficient Hadoop MapReduce program. We also have experimentally demonstrated that our generated program can achieve the same performance as a hand-optimized one. Our presented system enables the user to describe computations on one-dimensional arrays without caring about the MapReduce model and handle large-scale data efficiently with the advantages of Hadoop MapReduce such as fault tolerance.

A direction of future work is to relax the current restrictions on input programs and improve the expressiveness of them. Many miscellaneous restrictions will be removed by adding normalization passes. Our system will become able to deal with manipulation of multiple (a constant number of) one-dimensional arrays by introducing a common list skeleton called zip. In such cases, naturally independent tasks can occur and would bring room for optimization of MapReduce programs. Not only for manually described loops, it is worth introducing MapReduce-style primitives such as groupByKey and filter and generating MapReduce tasks that put it all together. We would then be able to evaluate our system more practically with input examples of Sawzall and FlumeJava.

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Number 26280020.

References

[1] Allen, F.E., Burke, M.G., Charles, P., Cytron, R. and Ferrante, J.: An overview of the PTRAN analysis system for multiprocessing, *Proc. 1st International Conference on Supercomputing* (*ICS '87*), Vol.194-211, Springer (1988).

[2] Bird, R.: *Introduction to Functional Programming using Haskell*, Prentice Hall (1998).

[3] Bird, R.S.: An Introduction to the Theory of Lists, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series F, Vol.36, pp.3–42, Springer (1987).

[4] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R. and Weizenbaum, N.: FlumeJava: Easy, Efficient Data-Parallel Pipelines, *Proc. 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '10*), pp.363–375, ACM (2010).

[5] Cole, M.I.: *Algorithmic Skeletons: Structural Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, MIT Press (1989).

[6] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation* (*OSDI '04*), pp.137–149, USENIX Association (2004).

[7] Dean, J. and Ghemawat, S.: MapReduce: A Flexible Data Processing Tool, *Commun. ACM*, Vol.53, No.1, pp.72–77 (2010).

[8] Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Program. Lang. Syst.*, Vol.9, No.3, pp.319–349 (1987).

[9] Fisher, A.L. and Ghuloum, A.M.: Parallelizing Complex Scans and Reductions, *Proc. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (*PLDI '94*), pp.135–146, ACM (1994).

[10] Fonseca, A., Cabral, B., Rafael, J. and Correia, I.: Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime, *Int. J. Parallel Prog.*, Vol.44, No.6, pp.1337–1358 (2016).

[11] Hu, Z., Iwasaki, H. and Takeichi, M.: Diffusion: Calculating Efficient Parallel Programs, *Proc. 1999 ACM SIGPLAN International Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (*PEPM '99*), pp.85–94 (1999).

[12] Iwasaki, H. and Hu, Z.: A New Parallel Skeleton for General Accumulative Computations, *Int. J. Parallel Prog.*, Vol.32, No.5, pp.389–414 (2004).

[13] Liu, Y., Emoto, K., Matsuzaki, K. and Hu, Z.: Accumulative Computation on MapReduce, *IPSJ Trans. Prog.*, Vol.7, No.1, pp.18–27 (2014).

[14] Liu, Y., Hu, Z. and Matsuzaki, K.: Towards Systematic Parallel Programming over MapReduce, *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, Lecture Notes in Computer Science, Vol.6853, pp.39–50, Springer (2011).

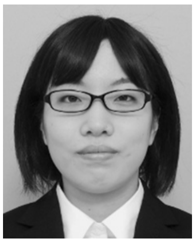
[15] Miyazaki, R. and Matsuzaki, K.: A Study of How Implementations of Accumulative Computation for Lists Affect Performance on Hadoop, *IPSJ Special Interest Group on Programming 107th Meeting* (2016). in Japanese.

[16] Morihata, A. and Matsuzaki, K.: Automatic Parallelization of Recursive Functions Using Quantifier Elimination, *Proc. 10th International Conference on Functional and Logic Programming* (*FLOPS '10*), pp.321–336, ACM (2010).

[17] Pike, R., Dorward, S., Griesemer, R. and Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall, *Sci. Program.*, Vol.13, No.4 (2005).

[18] Sato, S. and Iwasaki, H.: Automatic Parallelization via Matrix Multiplication, *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '11*), pp.470–479, ACM (2011).

[19] Smith, C. and Albarghouthi, A.: MapReduce Program Synthesis, *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '16*), pp.326–340, ACM (2016).



Reina Miyazaki was a Master-course student of Graduate School of Engineering, Kochi University of Technology, and passed through the course on 2017. Her research interest includes implementing libraries and language systems to support high-level parallel programming.

Kiminori Matsuzaki is an Associate Professor of Kochi University of Technology in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming, algorithm derivation, and game programming. He is also a member of ACM, JSSST, IEEE.

Shigeyuki Sato acquired his Ph.D. in engineering from The University of Electro-Communications in 2015 and has been a postdoctoral researcher at Kochi University of Technology since 2016. His research interest is on compilers, parallel programming, automatic parallelization, data-parallel skeletons, and domain-specific languages. He is a member of the JSSST.