IPSJ Transactions on Programming Vol.7 No.1 18–27 (Jan. 2014)

Regular Paper

Accumulative Computation on MapReduce

Yu Liu[1,4,a)] Kento Emoto[2,b)] Kiminori Matsuzaki[3,c)] Zhenjiang Hu[4,d)]

Received: July 4, 2013, Accepted: October 9, 2013

Abstract: MapReduce programming model attracts a lot of enthusiasm among both industry and academia, largely because it simplifies the implementations of many data parallel applications. In spite of the simplicity of the programming model, there are many applications that are hard to be implemented by MapReduce, due to their innate characters of computational dependency. In this paper we propose a new approach of using the programming pattern *accumulate* over MapReduce, to handle a large class of problems that cannot be simply divided into independent sub-computations. Using this *accumulate* pattern, many problems that have computational dependency can be easily expressed, and then the programs will be transformed to MapReduce programs executed on large clusters. Users without much knowledge of MapReduce can also easily write programs in a sequential manner but finally obtain efficient and scalable MapReduce programs. We describe the programming interface of our *accumulate* framework and explain how to transform a user-specified *accumulate* computation to an efficient MapReduce program. Our experiments and evaluations illustrate the usefulness and efficiency of the framework.

Keywords: parallel programming, MapReduce, accumulative computation, automatic parallelization, algorithmic skeleton

1. Introduction

MapReduce [10] is a popular parallel programming framework, which is designed for parallel data processing such as clustering, mining or statistical analysis on large-scale data. The programming model of MapReduce is inspired by the functional programming languages and algorithms in MapReduce model are mainly restricted to using *map* and *reduce* functions[*1]. The MapReduce framework executes such functions in a massively parallel manner while dealing with failures automatically.

In spite of the simplicity, many problems are still difficult to be expressed in MapReduce model. As an example, consider the *elimSmallers* problem of eliminating all the *smaller* elements of a list to produce an ascending list (if an element is less than someone in the previous, it is *smaller*). For instance, given a list $[11, 15, 8, 9, 20, 25, 12, 23]$, then 8, 9, 12 and 23 are smaller ones, and thus the result is $[11, 15, 20, 25]$. A recursive function that solves this problem can be defined as follows, in Haskell [5].

$$\begin{aligned} &\mathit{elimSmallers}\ [\,]\ c = [\,] \\ &\mathit{elimSmallers}\ (x : xs)\ c \\ &\qquad\qquad = (\textbf{if}\ x < c\ \textbf{then}\ [\,]\ \textbf{else}\ [x]) \mathbin{+\!\!+} \\ &\qquad\qquad\quad \mathit{elimSmallers}\ xs\ (\textbf{if}\ x < c\ \textbf{then}\ c\ \textbf{else}\ x). \end{aligned}$$

In this function, recursively, numbers of input are compared with an accumulative parameter c (with initial value $-\infty$). The accumulative parameter c always holds the current maximum value and is used in the next recursion. If a number is no less than c then it is appended to the tail of the result, and otherwise dropped. In functional programming, such kind of computation pattern with accumulative parameters is called accumulative computation [15], [16], [18].

The recursive function *elimSmallers* clearly describes the computation (in $O(n)$ work, n is the length of input), but it cannot be easily mapped to MapReduce, because in the recursive function *elimSmallers*, every inner step of the recursion relies on the current maximum value, which is computed at the outer step. Such kind of recursive functions do not correspond to a simple divide-and-conquer algorithm. Developing an $O(n)$ work MapReduce algorithm for *elimSmallers* needs to resolve such computational dependency and avoid unnecessary and expensive I/O, which is not easy for many programmers. There are many applications (e.g., the *prefix-sum/scan* related problems [6]) having similar characters with *elimSmallers* and thus are also difficult to be resolved in MapReduce model.

In this paper, we propose a new programming framework for simplifying MapReduce programming on accumulative computations that cannot be expressed by using *map* and *reduce* functions in only one iteration of MapReduce processing. The programming interface[*2] is designed to help users defining recursive functions in the accumulative form, and then efficient and scalable MapReduce solutions can be automatically gained. Our main technical contributions can be summarized as follows.

[1] The Graduate University for Advanced Studies, Chiyoda, Tokyo 101–0062, Japan
[2] Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[3] Kochi University of Technology, Kami, Kochi 782–8502, Japan
[4] National Institute of Informatics, Chiyoda, Tokyo 101–0062, Japan
[a)] yuliu@nii.ac.jp
[b)] emoto@ai.kyutech.ac.jp
[c)] matsuzaki.kiminori@kochi-tech.ac.jp
[d)] hu@nii.ac.jp

[*1] The *map* and *reduce* functions in MapReduce are inspired by but not equivalent to those in Lisp or Haskell.

[*2] The accumulate skeleton has been implemented using MPI that is more flexible in programming model and does not have same constants as MapReduce has.

- We implemented a parallel programming framework modeled by the accumulate computation pattern [15], [16], [18][*3] for accumulative computations. The framework can produce efficient and scalable MapReduce jobs for dealing with large data. Two important technical points in the implementations are: (1) dealing with accumulative computations on ordered lists, in spite of the massively parallel execution manner of MapReduce and (2) providing generic and high-level programming interfaces for wrapping the low-level MapReduce APIs.
- We evaluated the framework with many interesting examples, e.g., *tag match*, *elimSmallers*, *maximum prefix sum*, and *line-of-sight*, on MapReduce clusters dealing with big input data. The experimental results show good efficiency and scalability of our accumulation framework.

The organization of this paper is as follows. In Section 2, we briefly explain the MapReduce model and accumulative computation patterns. In Section 3, we introduce our MapReduce algorithms for accumulative computations and describe the library we developed on Hadoop. We present the experimental results in Section 4, introduce the related work in Section 5, and conclude the paper in Section 6.

2. Background

2.1 Notations

To make our descriptions precise and clear, notations in this paper are mainly based on the functional language Haskell [5]. Function application is denoted with a space with its argument without parentheses, i.e., $f\ a$ equals to $f(a)$. Functions are curried and bound to the left, and thus $f\ a\ b$ equals to $(f\ a)\ b$. Function application has higher precedence than using operators, so $f\ a \oplus b = (f\ a) \oplus b$. We use the operator $\circ$ over functions and $(f \circ g)\ x = f\ (g\ x)$. Function *id* is the identity function. Tuples are written like (a, b) or (a, b, c). Function *fst* (*snd*) extracts the first (the second) element of the input tuple, and *top xs* returns the first element of a stack *xs*. Function *drop m xs* drops the first *m* elements from a list *xs*. The binary operator $\uparrow$ applies on two numbers and returns the larger one. We denote lists with square brackets, and use $[\,]$ to denote an empty list, and $+\!\!+$ to denote the list concatenation: $[3, 1, 4] +\!\!+ [1, 5] = [3, 1, 4, 1, 5]$. Function $[\cdot]$ takes a value and returns a singleton list with the value. The scan, map, reduce, zip are standard skeletons in the Bird-Meertens formalism [4], [26]. To distinguish the *map* / *reduce* functions in MapReduce form above skeletons, we use f_{MAP} and f_{REDUCE} for the parameter functions used in the MapReduce.

2.2 MapReduce

Google's MapReduce [10] is a popular programming model for processing large data sets in a massively parallel manner. In the MapReduce programming model, parallel computations are represented in the paradigm of a parallel Map processing followed by a Reduce processing[*4]. Between the Map and Reduce phases, there is a Shuffle/Sort phase. **Figure 1** shows the typical data-processing flow of MapReduce. Note that Map tasks are executed independently (no direct way for one Map task to communicate/synchronize with another one, and so do the Reduce tasks. The only global synchronization in MapReduce is the barrier between Map and Reduce. Usually an instance of Map (Reduce) task is also called a mapper (reducer). The types of the two basic functions of MapReduce are defined as follows.

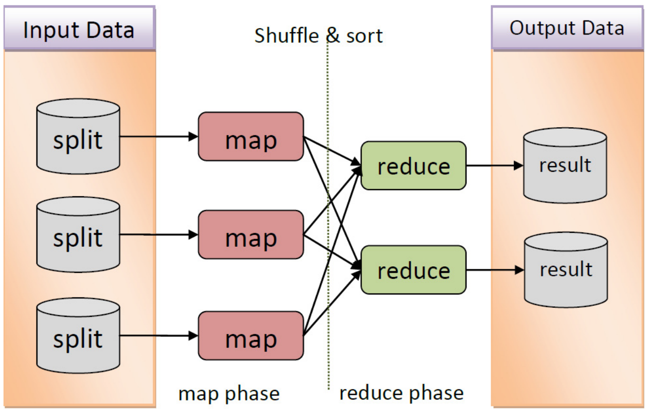


Fig. 1 Data-processing Flow of MapReduce.

- Function f_{MAP}.

$$f_{\mathrm{MAP}} :: (k_1, v_1) \rightarrow (k_2, v_2)$$

This function is invoked during the Map phase, and it is applied on each key-value pair of input and returns an intermediate key-value pair.

- Function f_{REDUCE}.

$$f_{\mathrm{REDUCE}} :: (k_2, [v_2]) \rightarrow (k_3, v_3)$$

This function is invoked during the Reduce phase, and it takes a key and a list of values associated to the key and merges the values.

Nowadays, several free, realistic implementations of MapReduce are available. In particular, Hadoop [2] is a famous open-source implementation using Java as its primitive language. Our implementation is based on Hadoop.

2.3 Accumulative Computations

Accumulative computation [15] plays an important role in describing a computation on an ordered list from left or right, when a later computation depends more or less on this computation. The innate character of data dependency can be captured by using an accumulative parameter that holds and delivers some information through the whole computation.

2.4 General Accumulative Computation Pattern

The accumulate skeleton abstracts a typical pattern of *recursive functions* with an accumulative parameter, which can be defined as a function h in the following form.

$$\begin{aligned} &h\ [\,]\ c = g\ c \\ &h\ (x : xs)\ c = p\ (x, c) \oplus h\ xs\ (c \otimes q\ x). \end{aligned}$$

This definition provides a natural way to describe computations with data dependencies and can be understood as follows.

- If the input list is empty, the result is computed by applying

[*3] http://code.google.com/p/diffusion-mapreduce/
[*4] The Reduce processing can be done in parallel or sequentially.

some function g to accumulative parameter c.

- If the input list is not empty and its head and tail parts are x and xs respectively, then the result is generated by combining the following two values using some binary operator $\oplus$: the result of applying p to x (head value) and c (the accumulative parameter), and the recursive call of h to xs (the rest part of the input list) with its accumulative parameter updated to $c \otimes q\ x$.

Because h is uniquely defined by g, p, $\oplus$, q, and $\otimes$, so we write h with special parentheses $[\![\ \ ]\!]$ as:

$$h\ xs\ c = [\![\, g,\ (p,\oplus),\ (q,\otimes)\,]\!]\ xs\ c.$$

Note that $(p,\oplus)$ and $(q,\otimes)$ correspond to two basic recursive forms *foldr* and *foldl* [14] respectively. The *elimSmallers* discussed in the introduction can be also written as follows.

```
$elimSmallers\ xs\ c = [\![\, g,\ (p,\oplus),\ (q,\otimes)\, ]\!]\ xs\ c$
    where  $g\ c = [\,]$
           $p\ (x,c) = \textbf{if}\ x < c\ \textbf{then}\ [\,]\ \textbf{else}\ [x],\ \ \oplus = +\!\!+,$
           $q\ =\ id,\ \ \otimes = \uparrow .$
```

Since the function h in the above form represents the most natural recursive definition on lists with a single accumulative parameter, it is general enough to capture many algorithms [15] as seen below.

Scan

Given a list $[x_1, x_2, x_3, x_4]$ and an associative binary operator $\odot$ with an identity element $\iota_\odot$, a function scan computes all its prefix sums yielding

$$[\iota_\odot,\ x_1,\ x_1 \odot x_2,\ x_1 \odot x_2 \odot x_3,\ x_1 \odot x_2 \odot x_3 \odot x_4].$$

As mentioned in the introduction, scan can be defined in terms of accumulate (by giving an initial value $\iota_\odot$ to the accumulative parameter c):

```
$scan\ [\,]\ c = [\cdot]\ c$
$scan\ (x : xs)\ c = ([\cdot] \circ snd)(x,c) +\!\!+ scan\ xs\ (c \odot\ (id\ x)).$
```

The function scan is very useful in algorithm design and is also a primitive operator in lots of parallel computations [6]. For example, lexical analysis, quick sort, and regular-expression matching can be implemented by using scan.

Line-of-Sight Problem

The well known line-of-sight problem [6] is that given a terrain map in the form of a grid of altitudes and an observation point, find which points are visible along a ray originating at the observation point. For instance, we use a pair (d,a) to represent a point, where a is the altitude of the point and d is its distance from the observation point. The function $\angle\ (d,a) = a/d$ computes the tangent of an angle. If the list is $[(1,1),(2,5),(3,2),(4,10)]$, then the point (3,2) is invisible. The function *los* [18] solves a simplified line-of-sight problem which counts the number of visible points.

```
$los\ xs\ c = [\![\, g,\ (p,+),\ (q,\uparrow)\, ]\!]\ xs\ c$
    where  $g\ c = 0$
           $p\ (x,c) = \textbf{if}\ c \le \angle\ x\ \textbf{then}\ 1\ \textbf{else}\ 0$
           $q\ x = \angle\ x.$
```

Maximum Prefix Sum Problem

Intuitively, the maximum prefix sum problem is to compute the maximum sum of all the prefixes of a list. Given a list $[\underline{3, -4, 9, 2}, -6]$ the maximum of the prefix sums is 10, to which the underlined prefix corresponds. We can define a function *mps* that solves this problem, in terms of accumulate.

$$mps\ xs\ c = [\![\, id,\ (snd,\uparrow),\ (id,+)\,]\!]\ xs\ c$$

Tag Matching Problem

The tag matching problem is to check whether the tags are well matched or not in a document, e.g., an XML file. There is an accumulative function *tagmatch* introduced by Ref. [18] for the tag matching problem.

```
$tagmatch\ xs\ cs = [\![\, isEmpty,\ (p,\wedge),\ (q,\otimes)\, ]\!]\ xs\ cs$
    where
        $p\ (x,cs)$
            $= \textbf{if}\ isOpen\ x\ \textbf{then}\ True$
              $\textbf{else if}\ isClose\ x\ \textbf{then}$
              $notEmpty\ cs\ \ \wedge\ \ match\ x\ (top\ cs)$
              $\textbf{else}\ True$
        $q\ x = \textbf{if}\ isOpen\ x\ \textbf{then}\ ([x],1,0)$
              $\textbf{else if}\ isClose\ x\ \textbf{then}\ ([\,],0,1)$
              $\textbf{else}\ ([\,],0,0)$
        $(s_1,n_1,m_1) \otimes (s_2,n_2,m_2)$
            $= \textbf{if}\ n_1\ \le\ m_2\ \textbf{then}\ (s_2,\ n_2,\ m_1 + m_2 - n_1)$
                              $\textbf{else}\ (s_2\ +\!\!+\ drop\ m_2\ s_1,$
                                     $n_1 + n_2 - m_2,\ m_1).$
```

3. Parallel Accumulation on MapReduce

Due to the different infrastructures of MapReduce and MPI, doing parallel accumulative computation on MapReduce is quite different and challenging. In typical MapReduce programming environments like Hadoop, or some MapReduce-like ones such as Dryad [17] and Spark [28], there is no option for users to use peer-to-peer communication like MPI_Send or MPI_Recv, and the synchronization of parallel processes can be only implemented by making use of the barrier between Map phase and Reduce phase.

There are two strategies for implementation of the accumulate skeleton. One is extending the existing MapReduce framework, by adding new peer-to-peer communication functions and barrier functions, so that we can do implementation in a similar way as Ref. [18] did. The other way is just making use of existing high-level API of a MapReduce framework such as *map* and *reduce* functions (and the necessary API of the distributed file system). Actually we have implemented and evaluated both, based on the state-of-the-art open-source MapReduce framework Hadoop. The prior one has advantages in the performance (several times faster) but significantly affected the fault tolerance mechanisms of MapReduce and it is not compatible with vanilla MapReduce frameworks [10]. On the contrary, the latter way, although it is not as fast as the prior one, enjoys all features of MapRedce's system design and has good portability (between different instances of

MapReduce frameworks). In this paper we select the latter way as our main solution and introduce the algorithm and implementation of it. The readers who are interested in the solution of prior way can refer the source code in the package of our framework.

Users of our framework can directly use the accumulate as a building block to solve their computations. Our implementation is based on Hadoop but could be easily ported to other MapReduce engines such as Spark [28].

3.1 Input Data Model

The accumulative computations take lists as input and obey the order of elements in the input list, while the input data of MapReduce are represented as a set of records (key-value pairs) stored in the distributed file system. This means we need a file-based *list* data structure for computations of accumulate. One issue for processing a file-based *list* is that the massively parallel processing manner of MapReduce could cause the accumulative computations being out-of-order.

For simplicity of discussion, we suppose the input data to an accumulative computation are stored as a list of records in one binary file (could be very huge) in the distributed file system (DFS). Each record of the file is a serialized Java object[*5] that represents an element of input list, and when detribalized to memory, each record will be transformed to a key-value pair. The key part of the pair is the corresponding Java object and the value part is a null object.

If big enough, the input file will be split to several splits by DFS (each split is called a chunk in the DFS) [13] and distributed to several DataNodes, and the DFS knows the offsets of each splits [10], [13], which can be seen as the indices of the splits. When data are loaded to multiple mappers, users of MapReduce cannot control which mapper to load which splits. In order to keep the total order of computation, the output of each mapper must be associated with the offset of its input, so that when merging the results from mappers, the order can be carefully manipulated by making use of such offsets and the sorting function.

3.2 A 2-pass MapReduce Algorithm for Accumulation

From the diffusion theorem [15], [18], an accumulative function $h = [\![\, g, (p,\oplus), (q,\otimes) \,]\!]$ can be transformed into the following compositional form using the parallel skeletons scan, map, reduce and zip.

$$\begin{aligned} &h\ xs\ c = \mathsf{reduce}\ (\oplus)\ (\mathsf{map}\ p\ as) \oplus g\ b \\ &\quad \textbf{where}\ bs +\!\!+ [b] = \mathsf{map}\ (c\,\otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ q\ xs)) \\ &\qquad\qquad as = \mathsf{zip}\ xs\ bs. \end{aligned}$$

In this form, map $(c\,\otimes)$ (scan $(\otimes)$ (map $q\ xs$)) can be firstly computed to get $bs +\!\!+ [b]$, then zip $xs\ bs$ to obtain as, and finally reduce $(\oplus)$ (map $p\ as) \oplus g\ b$ to get the result. However, directly doing in this way will generate a lot of intermediate data such as $bs +\!\!+ [b]$, as (these are much bigger than the input), so that it is uncomputable in the MapReduce-like environments where input data are usually in terabytes. The previous MPI implementation [18] was based on this form but using a fusion technique to avoid generating large intermediate data, but mapping that fusion to MapReduce is difficult because MapReduce lacks flexible communication/synchronization mechanisms as MPI. So we developed a new "fusion" algorithm on MapReduce to efficiently compute the above compositional form.

3.2.1 The MapReduce Implementation for General Accumulation

Our approach is to divide the computation into two MapReduce phases and restrain the data transportation between the two. Suppose input list xs is split to p sublists, i.e., $xs = chk_1 +\!\!+ chk_2 +\!\!+ \ldots chk_p$. The k^{th} split chk_k has m elements $[x_1^k, x_2^k, \ldots, x_m^k]$ and its offset is seg_k. Our two-pass MapReduce algorithm (shown in **Fig. 2**) actually avoids generating large intermediate data and thus it is efficient. We introduce the details in the following paragraphs.

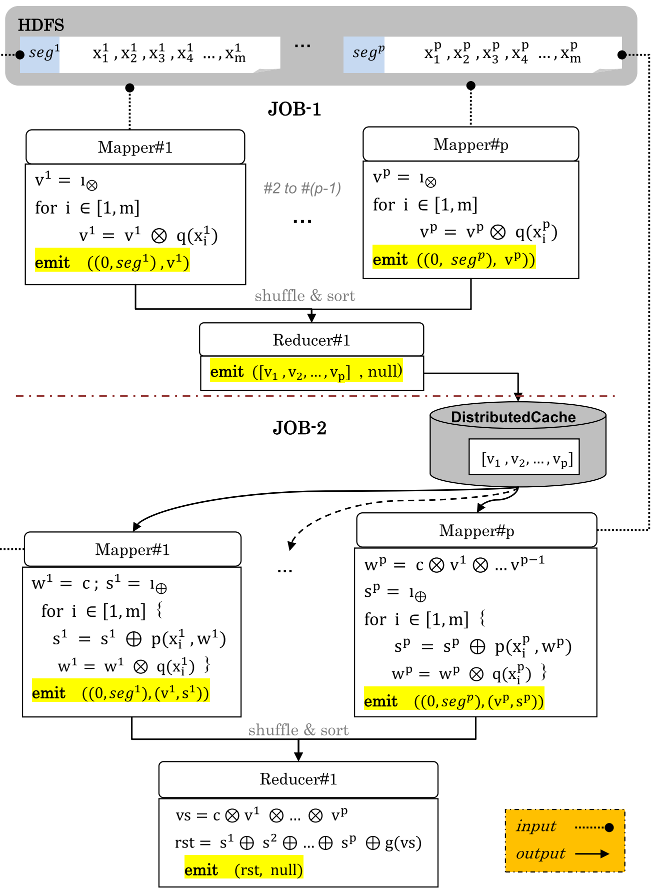
The first MapReduce job

There are p Map tasks spawned for each split, in the first MapReduce job. In general, for each sublist chk_k ($k \in [1, p]$), the first MapReduce computes:

$$mapRed_{map}\ chk_k = \mathsf{reduce}\ (\otimes)\ (\mathsf{map}\ q\ chk_k).$$

We do the above computation during Map phase and just use one reducer to collect the result. In detail, each Map task iterates over the elements of its input and applies the following f_{MAP} function on each input record $(x_i^k,\ \_\ )$ $(i \in [1, m])$.

$$f_{\text{MAP}}\ (x_i^k,\ \_\ ) = (\ (0, seg_k)\ ,\ q(x_i^k)).$$



HDFS
seg^1 $x_1^1, x_2^1, x_3^1, x_4^1 \ldots, x_m^1$
…
seg^p $x_1^p, x_2^p, x_3^p, x_4^p \ldots, x_m^p$
JOB-1
Mapper#1
$v^1 = \imath_\otimes$
for $i \in [1,m]$
$v^1 = v^1 \otimes q(x_i^1)$
emit $((0, seg^1), v^1)$
#2 to #(p-1)
…
Mapper#p
$v^p = \imath_\otimes$
for $i \in [1,m]$
$v^p = v^p \otimes q(x_i^p)$
emit $((0, seg^p), v^p)$
shuffle & sort
Reducer#1
emit $([v_1, v_2, \ldots, v_p]$, null)
JOB-2
DistributedCache
$[v_1, v_2, \ldots, v_p]$
Mapper#1
$w^1 = c$; $s^1 = \imath_\oplus$
for $i \in [1,m]$ {
$s^1 = s^1 \oplus p(x_i^1, w^1)$
$w^1 = w^1 \otimes q(x_i^1)$ }
emit $((0, seg^1), (v^1, s^1))$
…
Mapper#p
$w^p = c \otimes v^1 \otimes \ldots v^{p-1}$
$s^p = \imath_\oplus$
for $i \in [1,m]$ {
$s^p = s^p \oplus p(x_i^p, w^p)$
$w^p = w^p \otimes q(x_i^p)$ }
emit $((0, seg^p), (v^p, s^p))$
shuffle & sort
Reducer#1
$vs = c \otimes v^1 \otimes \ldots \otimes v^p$
$rst = s^1 \oplus s^2 \oplus \ldots \oplus s^p \oplus g(vs)$
emit (rst, null)
input
output

Fig. 2 The 2-pass MapReduce Accumulation.

[*5] In practice, serialization systems like Avro [1] can be used to improve performance.

Different with general MapReduce applications, here once the f_{MAP} function was applied on an input pair $(x_i^k,\ \_\ )$, the output did not be emitted immediately, but aggregated to a value v^k: $v^k = \iota_\otimes \otimes q(x_1^k) \otimes q(x_2^k) \otimes ... \otimes q(x_m^k)$. After the iterations, each Map task emits only one key-value pair: $(\ (0, seg_k)\ ,\ v^k)$. Here the key itself is a pair consisting of a constant value 0 and the offset seg_k. The outputs of Map tasks are grouped (by the constant value) and sorted by the offset.

In the reduce phase we only spawn one reducer. We use a special group function that groups records by first element of keys, so that the reducer collects all $(\ (0, seg_k)\ ,\ v^k))$ $(k \in [1, p])$ and sort them by the offsets seg_k. Then the reducer just emits a key-value pair $([v^1, v^2, ..., v^p],\ \_\ )$ (the key is a list and value part is useless) to the distributed file system. The f_{REDUCE} function is defined as follows.

$$f_{\mathrm{REDUCE}}\ (0,\ [v^1, v^2, ..., v^p]\ ) = ([v^1, v^2, ..., v^p],\ \_\ ).$$

Then the final result of the first MapReduce is a list that contains p elements, say $vs = [v^1, v^2, ..., v^p]$, and vs is guaranteed being in the correct order.

The second MapReduce job

After the first MapReduce, we initialize the second MapReduce: each second-Map task reads $vs = [v^1, v^2, ..., v^p]$ (the result list of the first reducer[*6]) from HDFS, in addition to the same input data as the first Map task. After initialization, in general, for each sublist chk_k each Map task in the second MapReduce computes:

$$\begin{aligned} &mapRed_{map}\ chk_k = \mathsf{map}\ p\ (\mathsf{zip}\ chk_k\ ws)\\ &\qquad \textbf{where}\\ &\qquad\qquad ws = \mathsf{map}\ (vk\ \otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ q\ chk_k))\\ &\qquad\qquad vk = \mathsf{reduce}\ (\otimes)\ [c, v^1, v^2, \ldots v^{k-1}]. \end{aligned}$$

The only one Reduce task in the second MapReduce computes:

$$\begin{aligned} &mapRed_{red}\ ss = (\mathsf{reduce}\ (\oplus)\ (ss)) \oplus\ g(vp)\\ &\qquad \textbf{where}\\ &\qquad\qquad ss = [s^1, s^2, \ldots s^p]\\ &\qquad\qquad vp = \mathsf{reduce}\ (\otimes)\ [v^1, v^2, \ldots v^p]. \end{aligned}$$

In detail, each Map task computes in a loop $s^k = p(x_1^k, w^k) \oplus p(x_2^k, w^k \otimes q(x_1^k)) \oplus ... \oplus p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$, where $w^k = c \otimes v^1 \otimes\ ... \otimes v^{k-1}$.

The output of a Map task is a nested key-value pair whose key is the same $(0, seg_k)$, and the value is (v^k, s^k). The f_{MAP} function is defined as follows.

$$f_{\mathrm{MAP}}\ ((x_i^k,\ \_\ ) = (\ (0, seg_k)\ ,\ p(x_i^k, w^k \otimes q(x_{i-1}^k))\ ).$$

Similar to the first pass MapReduce, the outputs of all Map tasks are grouped/sorted, and we spawn a single reducer in the second MapReduce. The final result is $s^1 \oplus s^2 \oplus \cdots \oplus s^p \oplus g(c \otimes v^1 \otimes v^2 \otimes \cdots \otimes v^p)$. The f_{REDUCE} function is defined as follows.

$$f_{\mathrm{REDUCE}}\ (0,\ ss) = (\mathsf{reduce}\ (\oplus)\ (ss +\!\!+\ g(vp)\ ),\ \_\ ).$$

Here $ss = [s^1, s^2, \ldots s^p]$, and $vp = \mathsf{reduce}\ (\otimes)\ [v^1, v^2, \ldots v^p]$.

[*6] We use the *DistributedCache* function of Hadoop to implement such initialization.

An example

As a concrete example, let us demonstrate the above algorithm to compute the *elimSmallers* problem on a two-nodes cluster. An input list is given as $[11, 15, 8, 9, 20, 25, 12, 23]$, the initial value of parameter $c = -\infty$, and the list is split to two (with the offset 0 and 10, respectively). The processing is represented in the following tabular form, step by step.

	node^1	node^2
input	$0 : [11, 15, 8, 9]$	$10 : [20, 25, 12, 23])$
1st Map	$-\infty \uparrow 11 \uparrow 15 \uparrow 8 \uparrow 9 = 15$ **output** $= ((0, 0), 15)$	$-\infty \uparrow 20 \uparrow 25 \uparrow 12 \uparrow 23 = 25$ **output** $=((0, 10), 25)$
1st Reduce	emit directly **output** $= ([15, 25], \_)$	N/A
2nd Map	$p(11 \uparrow \underline{-\infty}) +\!\!+ p(15 \uparrow \underline{11}) +\!\!+ p(8 \uparrow \underline{15}) +\!\!+ p(9 \uparrow \underline{15}) = [11, 15]$ **output** $=((0, 0), (15, [11, 15]))$	$p(20 \uparrow \underline{-\infty}) +\!\!+ p(25, \uparrow \underline{20}) +\!\!+ p(12 \uparrow \underline{25}) +\!\!+ p(23 \uparrow \underline{25}) = [20, 25]$ **output**$=((0, 10), (25, [20, 25]))$
2nd Reduce	$[11, 15] +\!\!+ [20, 25] +\!\!+ g(-\infty \uparrow 15 \uparrow 25)$ **output** $=([11, 15, 20, 25], \_)$	N/A

Discussions on efficiency

Our two-pass MapReduce algorithm for $\mathsf{accumulate}$ $[\![\, g,\ (p, \oplus),\ (q, \otimes)\,]\!]$ has two parallel Map phases and two sequential Reduce phases, and it only generates p intermediate data $(v^1, v^2, ..., v^p)$ and duplicates them p times through networks (copied to p Map tasks using *parallel-copy*). Consider that p i.e., the number of input splits, is not a very huge value (for 1 TB data, if the chunk size of HDFS is 128 MB then the p is 7813), so that if all v^k and s^k are in small constant size, then the two Reduce phases will not be bottlenecks and also the communication cost is low. This algorithm has been proved to be efficient and scalable by our evaluations shown in Section 4.2. However, there is still a restriction on operators $\otimes$ and $\oplus$, in practice. Under the assumption that the input data are larger than the storage capability of any single node in the cluster, that $\otimes$ must not be $+\!\!+$ (or any other that has similar effect), otherwise in the Map phases of the first MapReduce job, the result of $v^k = \iota_\otimes \otimes q(x_1^k) \otimes q(x_2^k) \otimes ... \otimes q(x_m^k)$ may be too large to be stored in the DistributedCache nor be transported via networks, unless function q can filter out (returns an empty list) most of the elements of the input. If $\otimes$ is not $+\!\!+$ but $\oplus$ is $+\!\!+$, then whether the $\mathsf{accumulate}$ is efficient depends on the size of s^k. Here $s^k = p(x_1^k, w^k) \oplus p(x_2^k, w^k \otimes q(x_1^k)) \oplus ... \oplus p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$, and $w^k = c \otimes v^1 \otimes\ ... \otimes v^{k-1}$. For function p, if it can filter out most of its input then using only one reducer in the second MapReduce will not be a big problem, otherwise we have to do special optimization for such case by using multiple reducers.

3.2.2 The Optimized MapReduce Implementation for Specialized Accumulation

If the emitted intermediate data are small enough, then they can be efficiently transferred to one reducer via network otherwise the computation will be very costive or the data are too large to be manipulated by only one reducer.

In order to improve the performance for some special cases such that (in the Map phase of the second MapReduce job), $s^k = p(x_1^k, w^k) +\!\!+ p(x_2^k, w^k \otimes q(x_1^k)) +\!\!+ ... +\!\!+ p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$ is a long list (suppose the input is split to p sub-

Listing 1: Scan Representation

```
public class Scan<T> {
  public AssociativeBinaryOP<T> oplus;

  public Scan(AssociativeBinaryOP<T> op) {
    this.oplus = op;
  }
}
```

lists), we have optimized the implementation. We do not group all output of Map phase to one reducer but use multiple reducers instead. The number of reducers can be adjusted to fit the practical problems and data. Same as the general case in Subsection 3.2.1, output of Map are sorted by seg_k $(k \in [1, p])$, but grouped to t reducers. Intuitively, let $r = p/t$, reducer^k receives $[s^{k*r+1}, s^{k*r+2}$ $..., s^{(k+1)r}]$, and emits $s^{k*r+1} +\!\!+\, s^{k*r+2} \ldots +\!\!+\, s^{(k+1)r}$, $(k \in [0, r-1))$. The reducer^t receives $[s^{p-r+1}, s^{p-r+2}$ $..., s^p]$ and reads $[v^1, ..., v^p]$ from *DistributedCache*, and computes $w = c \otimes v^1 \otimes v^2 \otimes \cdots \otimes v^p$. At last the reducer^t emits $s^{k*r+1} +\!\!+\, s^{k*r+2} \ldots +\!\!+\, s^{(k+1)r} +\!\!+\, g(w)$,. Each output from the t reducers contains part of the final result.

The Optimized Implementation for Scan

The scan skeleton is a special case of accumulate: $scan = [\![\, [\cdot], ([\cdot] \circ snd, +\!\!+), (id, \otimes) \,]\!]$, i.e., $g = [\cdot]$, $p = [\cdot] \circ snd$, $\oplus = +\!\!+$, and $q = id$. The MapReduce implementation of scan can be optimized and efficiently computed, if $\otimes$ is not $+\!\!+$. Because the result of scan is $s^1 +\!\!+\, s^2 +\!\!+ \cdots +\!\!+\, s^p$ we do not need the Reduce phase in the second MapReduce, and just let each mapper emit (seg_k, s^k) $(k \in [1, p])$. The seg_k denotes the offset of sublist handled by the k^{th} mapper, so that these pairs can be sorted by seg_k and form the final result.

3.3 The Programming Interfaces

We provide two parallel skeletons scan and accumulate in our framework. These two skeletons and also the related binary operators are represented as Java classes, in the object-oriented style.

3.3.1 Scan Interface

Listing 1 shows the representation Java class for scan. Users need to define the associative binary operator to create an instance of the scan computation. There is an example of an associative binary operator `Plus` in Listing 2, which adds two integers and returns the sum. The `evalute` method takes two arguments and returns one value. The `ie` method returns the identity element. To execute a scan on MapReduce cluster, users need to write the client codes like Listing 2. The Java class `ScanExample` extends `ScanMRHelper` and overrides the method `createScanIns` which creates an instance of the scan computation. In the `main` function, the method `runScanMR` which takes two arguments — one is an instance of scan, and the other is the `args` (the input, output paths given by users) from `main`, — will execute the scan computation on the Hadoop cluster.

3.3.2 Accumulation Interface

An accumulate can be defined by implementing the abstract class `Accumulation` (as shown in Listing 3). There are five functions/operators according to the definition of the accumulate, and an accumulative parameter `c`. The Java class `MapReduceExample` (Listing 4) which extends `MRAccHelper` shows how to write the client code. Similarly to scan, the method

Listing 2: An Example of Using the Scan Programming Interface

```
public class ScanExample extends MRScanHelper<Int> {
  // type Int is a wrapping class for Java int
  public class Plus extends AssociativeBinaryOP<Int> {
    public Int evaluate(Int a, Int b) {
      return new Int(a.val + b.val);
    }
    public Int id() {
      return new Int(0);
    }
  }

  @Override
  public void createScanIns() {
    // instance an scan computation
    this.scan = new Scan(new Plus());
  }

  public static void main(String[] args) throws
       Exception {
    // Create and run on MapReduce
    int res = runScanMR(new ScanExample(), args);
    System.exit(res);
  }
}
```

Listing 3: Accumulation Representation

```
public abstract class Accumulation<T0, T1, T2> {
  public T1 c;
  public UnaryFunction<T1, T2> g;
  public AssociativeBinaryOP<T2> oplus;
  public AssociativeBinaryOP<T1> otimes;
  public BinaryOperator<T0, T1, T2> p;
  public UnaryFunction<T0, T1> q;
}
```

Listing 4: An Example of Using the Accumulation Programming Interface

```
public class MapReduceExample extends MRAccHelper<Int,
    Int,IntList> {
  public void createAccuIns( ) {
    //instance an accumulate computation
    this.accumulate = new ElimSmallers(new Int(50));
  }
  public static void main(String[] args)
                          throws Exception {
    //Create and run on MapReduce
    int res = runAccMR( new AExample(), args);
    System.exit(res);
    }
  }
}
```

`createAccuIns` needs to be override, in which an instance of accumulate is created. In the `main` function, the `runAccMR` method should be invoked to execute the accumulate. The Listing 5 shows an example to define the *elimSmallers* computation.

4. Programming Examples and Evaluations

We have developed several examples by using the parallel skeletons scan and accumulate provided by our framework, and evaluated them on Hadoop clusters.

4.1 Example Programs

Table 1 lists five examples developed on our framework. More examples of accumulative computations can be found in the source packages of the framework. The *scan* (+) is an application of prefix-sum on numbers using the binary operator +. The

Listing 5: Difination of Accumulation for ElimSmallers

```
public class ElimSmallers extends Accumulation<Int, Int,
      IntList> {

  public ElimSmallers(Int x) {
    c = x;// new Int(50);

    oplus = new AccociBinaryOP<IntList>() {
      @Override
      public IntList evaluate(IntList left, IntList
           right) {
        if (left == null || left.get() == null)
          left = new IntList(new ArrayList<Int>());
        if (right != null && right.get() != null
            && right.get().size() > 0) {
          left.get().addAll(right.get());
        }
        return left;
      }
      @Override
      public IntList id() {
        return new IntList();
      }
    };

    otimes = new AccociBinaryOP<Int>() {
      @Override
      public Int evaluate(Int left, Int right) {
        if (right.get() < left.get())
          return left;
        else
          return right;
      }
      @Override
      public Int id() { // id * x = x
        return new Int(Integer.MIN_VALUE);
      }
    };

    g = new UnaryFunction<Int, IntList>() {
      @Override
      public IntList evaluate(Int obj) {
        return new IntList();
      }
    };

    p = new BinaryOperator<Int, Int, IntList>() {
      @Override
      public IntList evaluate(Int x, Int c) {
        ArrayList<Int> val = new ArrayList<Int>();
        if (x.get() < c.get()) {
          return null;
        } else {
          val.add(x);
          return new IntList(val);
        }
      }
    };

    q = new UnaryFunction<Int, Int>() {
      @Override
      public Int evaluate(Int da) {
        return da;
      }
    };

  }
}
```

elimSmallers, *los*, *tagmatch* and *mps* are those introduced in Section 2. Each program requires a particular type of input list, such as list of numbers, list of tags or list of pairs. Table 1 also gives the type of the input lists for each program.

4.2 Evaluation

We evaluated the performance and scalability of the example programs with manually generated data sets shown in Table 1.

Table 1 Example programs for five problems and data types of their input.

Problem	*scan* (+)	*elimSmallers*	*los*	*tagmatch*	*mps*
Input Data	Numbers	Numbers	Pairs	Tags	Numbers

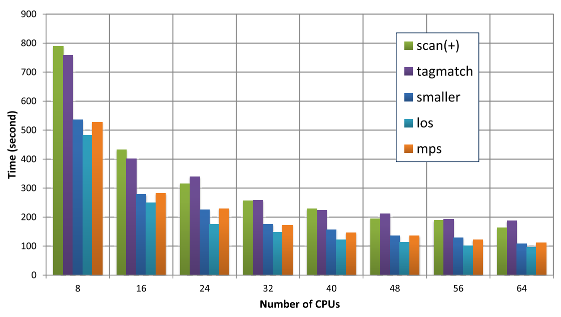


Fig. 3 Running Times of Each Accumulative Programs.

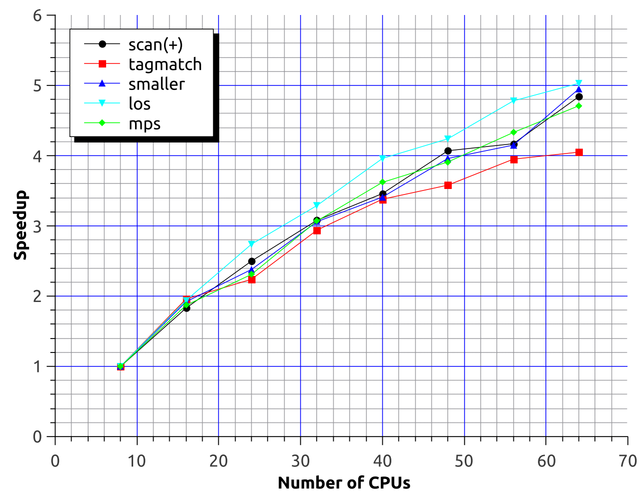


Fig. 4 Relative Speedup Calculated with Respect to The Result on 8 CPUs.

Table 2 Data sets for evaluating examples on Hadoop clusters with different number of working nodes.

Program	Input Length	Input Size
scan (+)	$5{,}000 \times 2^{20}$	9.77 GB
elimSmallers	$5{,}000 \times 2^{20}$	9.77 GB
los	$5{,}000 \times 2^{20}$	10.54 GB
tagmatch	$5{,}000 \times 2^{20}$	9.77 GB
mps	$5{,}000 \times 2^{20}$	9.77 GB

We configured Hadoop (cdh3u5) clusters with up to 32 virtual machines (VMs) inside the *EdubaseCloud* system in National Institute of Informatics. Each VM has 2 CPUs (a CPU is one core of the Xeon E5530@2.4 GHz), 6 GB RAM. The total parallel-task slots in Hadoop are configured to be equal to the total number of CPUs in the cluster[*7].

Scalability

The experiment results are summarized in **Figs. 3** and **4**. Letting the input data be fixed size (shown as **Table 2**) while increasing the working nodes of the cluster (i.e., increasing VMs), all programs have almost twice speedup when the number of CPUs increases from 8 to 16. This indicates the good scalability of our framework. When the number of CPUs keeps increasing, the running-time becomes shorter and approaches to a constant value

[*7] We made this configuration in order to simplify the analysis of scalability. In fact, optimizing the configurations of the Haddop cluster, e.g., allowing more mappers running simultaneously in each VM, can obtain much better performance.

Table 3 Large data sets for comparing two kinds of implementation: with/without using our framework.

Input Data	Length	Size
Numbers	$1\times10^5\times 2^{20}$	195.32 GB
Pairs	$1\times10^4\times 2^{20}$	233.21 GB
Tags	$1\times10^5\times 2^{20}$	195.32 GB

Table 4 In comparison of length of code and performance to vanilla Hadoop programs (using the data sets listed in Table 3).

Problems	Lines (*vanilla*)	Lines	Time (vanilla)	Time
scan (+)	163	29	1,995 s	1,988 s
elimSmallers	368	107	2,012 s	2,013 s
los	348	81	1,583 s	1,587 s
tagmatch	346	107	2,902 s	2,903 s
mps	347	75	1,793 s	1,791 s

which is the time of fixed sequential parts computation and system overhead of Hadoop. As a summary, the relation between speedup (y) and number of CPUs (x) approximately fits to a linear curve $y = Ax + B$, e.g, in case of *scan* (+), $A = 6.49 \times 10^{-2}$, $B = 0.779$.

In Comparison with Vanilla Hadoop Programs

Finally we discuss the programmability and relative efficiency by comparing the programs written by using our framework and those written by directly using the Hadoop API. As mentioned before, programs written by using our *accumulate* API will be transformed to programs that are exactly equivalent to those manually written by using vanilla Hadoop (i.e., without using our framework). The comparison we gave just shows the benefits that how much our framework saves programmers efforts and low overhead of the high-level abstraction in our framework.

For each problem in Table 1, we made a new version using only vanilla Hadoop, which implemented the same two-phases MapReduce algorithm in Section 3.2. **Table 4** shows the comparison of length of source code and running times of the two classes of programs (on the same 64-CPU cluster). The source code is formatted by the Eclipse code formatter, and counted by using Google CodePro Analytix*[8]. We used much larger data sets listed in **Table 3** as input for the evaluation.

In Table 4, the column *Lines (vanilla)* is for the lengths of programs implemented without using our framework, and the column *Lines* is for the lengths of programs implemented by using our framework. The column *Time (vanilla)* and the column *Time* are running times of the two versions.

The results show that all vanilla Hadoop programs are much longer than programs written by using our *accumulate* API (3.2–5.6 times longer). The system overhead caused by the generic abstraction and wrapping of Hadoop API can be almost negligible. In addition, programs implemented by using our framework can still handle the larger input data (nearly 20 times larger compared to each data set in Table 2) very well.

Generally, the main difficulty for a Hadoop programmer to implement a MapReduce algorithm for problems such as *elimSmallers*, is about finding the scalable divide-and-conquer algorithm. Furthermore, even when he knows the algorithm, the implementation of the cumbersome Hadoop code is still probably very time consuming.

*[8] https://developers.google.com/java-dev-tools/codepro/

5. Related Work

Algorithmic skeletons for parallel programming have been well studied from 1989 [9], and a lot of frameworks have been developed to provide those algorithmic skeletons [3], [7], [8], [21], [24]. Not only the programming frameworks, we have studied a systematic way to develop parallel programs using those skeletons. In particular, scan is a very useful skeleton because it enables us to reuse the partial results in reduce. For example, In [19] we showed that we can solve a set of *maximum marking problems* on lists with scan skeletons. We also showed that we can perform the matching of a regular expression over a single large document in parallel [20]. With the scan or accumulate computations developed in this paper, we can extend the technique to retreive substrings.

The MapReduce was firstly introduced by Google to handle very large raw data form Internet. The programming model of MapReduce is inspired by the concepts from functional programming [10], [22]. MapReduce gains a big success in both industry and academia, because of its functionality and simplicity. But there is still a gap between a nontrivial problem and the MapReduce paradigm because MapReduce programming model is relatively low-level that leads many difficulties in piratical programming. Many studies such as Sawzall [25], PigLatin [12], and DryadLINQ [27] tried to provide more user-friendly domain specific languages to address the programmability problem, and our previous work [11], [23] introduced the approaches of calculation theorems for list homomorphisms into MapReduce, for the similar purpose but in different methodology.

The accumulate as an algorithmic parallel computation pattern has been implemented by using MPI [18] and now is a part of the Sketo library [24] that provides a simple programming interface and efficient parallel implementation. To our knowledge, there was no completable MapReduce implementation for the accumulative computing*[9] before present study.

6. Conclusion

The research on parallel skeletons [9], [15], [18] and list homomorphisms [23] illustrates a systematic and constructive way to high-level parallel programming, by which this work was inspired. In this paper, we have described how to implement and use the two parallel skeletons scan and accumulate, in MapReduce. We provide a Hadoop-based framework with high-level programming interfaces. A large class of computations that are originally difficult to be programmed directly with MapReduce APIs can be easily implemented on our framework and enjoy the merits of MapReduce. The implementation is efficient and scalable, because it is based on the result of applying the fusion transformation to accumulate, which eliminates unnecessary intermediate data structures.

Although we limited our discussion to *lists* in this paper, the diffusion theorem can be extended to *trees* [16] and other general recursive data types. We plan to implement more algorithmic

*[9] Some MapReduce frameworks, Pig [12] and Spark [28] also have so-called "accumulate" interface, but such "accumulate" is not in the same sense of the functional programming pattern accumulate.

skeletons on MapReduce to simplify the parallel programming and large-scale data processing.

Acknowledgments We would like to thank Hideya Iwasaki, Shigeyuki Sato, and Le Duc Tung, for their helpful advices and constructive comments on this paper. This work was partially supported by JSPS KAKENHI Grant Number 24700025, and JST/ANR Grant Numbers 10102704/ANR-2010-INTB-0205-02.

References

[1] Apache Software Foundation: Avro, available from ⟨http://Avro.apache.org⟩.

[2] Apache Software Foundation: Hadoop, available from ⟨http://hadoop.apache.org⟩.

[3] Benoit, A., Cole, M., Gilmore, S. and Hillston, J.: Flexible skeletal programming with eskel, *Proc. 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pp.761–770, Berlin, Heidelberg, Springer-Verlag (2005).

[4] Bird, R.: An introduction to the theory of lists, *Proc. NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pp.5–42 (1987).

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

[6] Blelloch, G.E.: Scans as primitive operations. *IEEE Trans. Comput.*, Vol.38, No.11, pp.1526–1538 (1989).

[7] Botorog, G.H. and Kuchen, H.: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, *Proc. 5th International Symposium on High Performance Distributed Computing* (*HPDC-5*), pp.243–252, IEEE Computer Society (1996).

[8] Botorog, G.H. and Kuchen, H.: Efficient high-level parallel programming, *Theoretical Computer Science*, Vol.196, pp.71–107 (1997).

[9] Cole, M.: Algorithmic skeletons: A structured approach to the management of parallel computation, *Research Monographs in Parallel and Distributed Computing* (1989).

[10] Dean, J. and Ghemawat, S.: Mapreduce: Simplified data processing on large clusters, *Commun. ACM*, Vol.51, No.1, pp.107–113 (2008).

[11] Emoto, K., Fischer, S. and Hu, Z.: Filter-embedding semiring fusion for programming with mapreduce, *Formal Aspects of Computing*, Vol.24, pp.623–645 (2012).

[12] Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S. and Srivastava, U.: Building a high-level dataflow system on top of map-reduce: The pig experience, *Proc. VLDB Endow.*, Vol.2, No.2, pp.1414–1425 (2009).

[13] Ghemawat, S., Gobioff, H., and Leung, S.-T.: The google file system, *SIGOPS Oper. Syst. Rev.*, Vol.37, No.5, pp.29–43 (2003).

[14] Gibbons, J.: The third homomorphism theorem, *Journal of Functional Programming*, Vol.6, No.4, pp.657–665 (1996).

[15] Hu, Z., Iwasaki, H. and Takeichi, M.: An accumulative parallel skeleton for all, *11th European Symposium on Programming* (*ESOP 2002*), pp.83–97, Lecture Notes in Computer Science 2305, Springer Verlag (2002).

[16] Hu, Z., Takeichi, M. and Iwasaki, H.: Diffusion: Calculating efficient parallel programs, *Proc. 1999 ACM SIGPLAN International Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (*PEPM'99*), BRICS Notes Series NS-99-1, pp.85–94 (1999).

[17] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks, *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp.59–72, ACM (2007).

[18] Iwasaki, H. and Hu, Z.: A new parallel skeleton for general accumulative computations, *International Journal of Parallel Programming*, Vol.32, pp.389–414 (2004).

[19] Matsuzaki, K., Hu, Z. and Takeichi, M.: Derivation of parallel programs for maximum marking problems on lists, *IPSJ Trans. Programming*, Vol.49, No.3, pp.16–27 (2008).

[20] Matsuzaki, K., Emoto, K., and Liu, Y.: Parallelization of regular expression matching and its evaluation on hadoop, *IPSJ Trans. Programming*, Vol.4, No.4, pp.1–11 (2011).

[21] Kuchen, H.: A skeleton library, *Proc. 8th International Euro-Par Conference* (*Euro-Par2002*), volume 2400 of *Lecture Notes in Computer Science*, pp.620–629, Springer-Verlag (2002).

[22] Lämmel, R.: Google's MapReduce programming model — Revisited, *Science of Computer Programming*, Vol.70, No.1, pp.1–30 (2008).

[23] Liu, Y., Hu, Z. and Matsuzaki, K.: Towards systematic parallel programming over mapreduce, *Proc. 17th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'11. Springer-Verlag (2011).

[24] Matsuzaki, K., Iwasaki, H., Emoto, K. and Hu, Z.: A library of constructive skeletons for sequential style of parallel programming, *Proc. 1st International Conference on Scalable Information Systems*, *InfoScale '06*, ACM (2006).

[25] Pike, R., Dorward, S., Griesemer, R. and Quinlan, S.: Interpreting the data: Parallel analysis with sawzall, *Sci. Program.*, Vol.13, No.4, pp.277–298 (2005).

[26] Skillicorn, D.B.: The bird-meertens formalism as a parallel model, *Software for Parallel Computation*, volume 106 of *NATO ASI F*, pp.120–133, Springer-Verlag (1993).

[27] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K. and Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language, *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'08*, pp.1–14, USENIX Association (2008).

[28] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *Proc. 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp.2–2, USENIX Association (2012).

Yu Liu is a Ph.D. student in the Graduate University for Advanced Studies (Japan), and is a Research Assistant of National Institute of Informatics (NII). He received his B.S. from Xi'an Jiao Tong University (China) in 2004, and began doctoral study since 2009. Before entered Ph.D. course of the Graduate University for Advanced Studies, he worked as a softer engineer in China and Japan. His current research interest is parallel programming and parallel algorithms. He is a member of JSSST.

Kento Emoto is an Assistant Professor of Kyushu Institute of Technology (Kyutech) in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2004, 2006 and 2009, respectively. He was a Researcher (2007–2009) and an Assistant Professor (2009–2013) in The University of Tokyo, before joining Kyutech as an assistant professor in 2013. His main interest is in programming languages and software engineering in general, and parallel programming and algorithm synthesis in particular. He is a member of ACM, JSSST, IPSJ.

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 and algorithm derivation. He is a member of ACM, JSSST, IEEE.

Zhenjiang Hu is a Professor of National Institute of Informatics (NII) in Japan. He received his B.S. and M.S. from Shanghai Jiao Tong University in 1988 and 1991 respectively, and Ph.D. degree from The University of Tokyo in 1996. He was a Lecture (1997–1999) and an Associate Professor (2000–2007) in The University of Tokyo, before joining NII as a Full Professor in 2008. His main interest is in programming languages and software engineering in general, and functional programming, parallel programming and bidirectional model-driven software development in particular. He is a member of ACM, JSSST, IPSJ.