Int J Parallel Prog (2017) 45:362–381
DOI 10.1007/s10766-016-0414-9

Functional Models of Hadoop MapReduce with Application to Scan

Kiminori Matsuzaki[1]

Received: 27 August 2015 / Accepted: 16 March 2016 / Published online: 5 April 2016

Abstract MapReduce, first proposed by Google, is a remarkable programming model for processing very large amounts of data. An open-source implementation of MapReduce, called Hadoop, is now used for developing a wide range of applications. Although developing a correct and efficient program on MapReduce is much easier than developing one with MPI etc., it is still nontrivial if the target application requires involved functionalities of Hadoop MapReduce. Under these situations, functional models for MapReduce computation play important roles because we can utilize them for better understanding, proving the correctness, and even optimization of MapReduce programs. In this paper, we develop two functional models, a low-level one and a high-level one, which capture the semantics of Hadoop MapReduce computation. We discuss the detailed semantics mainly in terms of the following two computations: the computation of `Mapper` and `Reducer` classes and the computation in the Shuffle phase with the secondary-sorting technique. In addition, we develop MapReduce algorithms for the scan computational pattern (prefix sums) on the newly proposed models.

Keywords MapReduce · Functional model · Hadoop

1 Introduction

MapReduce, first proposed by Google [8], is a remarkable programming model as well as an infrastructure for processing very large amounts of data on large clusters. From the viewpoint of programming, it provides a simple data-parallel programming

✉ Kiminori Matsuzaki
matsuzaki.kiminori@kochi-tech.ac.jp

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

model, in which users should specify in principle two parameter functions: `map` and `reduce`. From the viewpoint of infrastructure, it provides nice mechanisms such as task mapping, load balancing, and fault tolerance.

Several MapReduce-like implementations have been developed, for instance, Hadoop [1,25], Phoenix [22], Spark [27], and SSS [18]. Among them, Hadoop, an open-source implementation of MapReduce, is now widely used in very many companies dealing with large amounts of data, such as Yahoo!, IBM, Amazon, Facebook, and Twitter [2]. It is used for not only processing data from the web but also developing a wide range of applications [16].

Although Google's original MapReduce and Hadoop MapReduce were implemented in C++ and Java, they attracted the interest of researchers in the functional programming community and algorithm community. There have been several studies on modeling MapReduce programming models in a more formal manner. Among them, the most highly cited work was by Lämmel [15], who first discussed a functional model of Google's MapReduce. There have been several others from a theoretical point of view, for instance, Feldman et al. [11] and Karloff et al. [14] proposed algorithmic classes that MapReduce can deal with. Pace [20] compared MapReduce computation with the BSP (bulk synchronous parallel) model [24].

Functional models are important for several reasons.

- *Understanding the computation* Since MapReduce uses the same terms, *map* and *reduce*, in a different way from functional languages like Lisp or Haskell, some people misunderstood or were misled by the actual computation model of MapReduce [7]. A clear functional model helps us to understand the computation correctly.
- *Proof of correctness* Correctness of programs is an important property, and certified parallel programming is now an important topic in parallel programming [17]. Functional models are very helpful for proving the correctness of MapReduce programs. On this topic, Ono et al. [19] and Jiang et al. [13] used a simpler model of MapReduce to prove MapReduce programs on the Coq proof assistant.
- *Preventing unsafe usage* Even in the conventional MapReduce framework, we could share states or communicate between Map tasks through the back-door, since imperative code can make it easy. However, such an unintended or unapproved usage is often unsafe. Developing algorithm on functional models prevents such an unsafe usage of the framework.
- *Program calculation* After getting suitable functional models, we can apply the program transformation (or program calculation) techniques (e.g. [12]) to obtain better programs from specifications.
- *Cost model* We can also develop cost models [10] based on those functional models. A cost model plays an important role in optimization: we can predict the performance of MapReduce programs before execution or with a little profiling of execution. On this topic, Dörre et al. [10] wrote down the computation of Hadoop MapReduce and developed a cost model for MapReduce programs.

In this study, we develop two functional models that capture the semantics of Hadoop MapReduce computation. Since Hadoop is now the de facto standard implementation of the MapReduce framework, we take into account the Hadoop-

specific mechanism and implementation. We write down a low-level model based on the implementation of Hadoop MapReduce and then modify it into a high-level one.

The contributions of the paper are summarized as follows.

- *Functional models* We develop two functional models, a low-level one and a high-level one, for Hadoop MapReduce. The Haskell test code is available at http://www.info.kochi-tech.ac.jp/~kmatsu/MRModel/
- *List scan on MapReduce* Based on the functional models, we develop algorithms of the scan (prefix sums) computation on lists. The development of BSP-inspired scan algorithm on MapReduce have not been reported in literature as far as the author knows.

The rest of the paper is organized as follows. In Sect. 2, we introduce notations and basic functions used in the paper. In Sect. 3, we start by briefly reviewing the programming model of MapReduce. We propose a low-level functional model in Sect. 4 based on the implementation of Hadoop MapReduce. We then modify the model into a high-level one in Sect. 5 in terms of the Shuffle phase. We develop two scan algorithms on the proposed functional models in Sect. 6. We review related work in Sect. 7 and finally conclude the paper in Sect. 8.

2 Preliminaries

In this paper, we basically borrow the notation of Haskell [4] for describing the functional models and the algorithms. The line after "−−" is dealt with as comment in Haskell.

2.1 Basic Notation

A function application is denoted with a space with its argument without brackets: $f\ a$ means $f(a)$. Functions are curried and bind to the left: $f\ a\ b$ means $(f\ a)\ b$. Function composition is denoted by $\circ$, and the identity function is id: $(f \circ g)\ x = f\ (g\ x)$. The last expression could be written without brackets using the "$" operator: $f\ (g\ x) = f\ \$\ g\ x$. Anonymous functions (lambda expressions) are denoted with "\" and "→": $f = \backslash x \to 2 * x$ is the same as $f\ x = 2 * x$. We can use "_" for a parameter to show that we do not care about its value. For a binary operator $\oplus$, we can treat it as a function by sectioning: $(\oplus)\ a\ b = (a \oplus)\ b = (\oplus b)\ a = a \oplus b$.

In this paper, we use two type classes, Eq and Ord, to clarify requirements on datatypes. For any datatype that belongs to Eq, we have the operator "==". For any datatype α that belongs to Ord, we have the function *compare* of type $compare :: \alpha \to \alpha \to Ordering$. Here, the type *Ordering* has three constructors, which represent "less than," "equal," and "greater than," respectively.

$$\textbf{data}\ Ordering = LT \mid EQ \mid GT$$

The function *comparing* is used to apply a function before comparing two values.

$$comparing\ f\ a\ b = compare\ (f\ a)\ (f\ b)$$

Tuples consist of a finite number of values, for example, pair (a, b) or triple (a, b, c). The function *fst* takes the first element of the pair; *snd* takes the second element. The following function *applyW* applies the parameter function and makes a pair with the output and the input.

$$applyW :: (\alpha \to \beta) \to \alpha \to (\beta, \alpha)$$
$$applyW\ f\ a = (f\ a, a)$$

2.2 Lists and Functions Manipulating Lists

A list is an ordered sequence of elements of the same type. We denote a list with square brackets. The type of list with elements of type α is denoted by $[\alpha]$. The empty list is denoted by [ ], and list concatenation is denoted by binary operator $+\!\!+$. The function *head* takes the first element in the list, and *last* takes the last element. For a list *xs*, *xs* !! *n* returns the *n*th value in *xs*. The function *init* returns all the elements in the list except for the last element. For a list *xs*, *take n xs* returns the first *n* elements in *xs*, and *drop n xs* removes the first *n* elements.

In the functional programming community, Bird-Meertens Formalism [5] is known as one of the programming theories for lists (and other data structures). Here are some important functions for lists used in the paper. Their definitions are given in Fig. 1.

The function *map* takes a function and applies it to every element in the input list. The function *foldl* collapses the input list from left to right using a binary operator. The function *zip* takes two lists and makes pairs of the corresponding elements. The

$$map\ f\ [a_0, a_1, \ldots, a_{n-1}] = [f\ a_0, f\ a_1, \ldots, f\ a_{n-1}]$$
$$foldl\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] = (\cdots((e \oplus a_1) \oplus a_2) \oplus \cdots) \oplus a_n$$
$$zip\ [a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_n] = [(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$$
$$scan\ (\oplus)\ e\ [a_0, a_1, \ldots, a_{n-1}] = [y_0, y_1, \ldots y_{n-1}]$$
$$\quad \textbf{where}\ y_i = e \oplus a_0 \oplus \cdots \oplus a_{i-1}$$

$$flatten :: [[\alpha]] \to [\alpha]$$
$$flatten = foldl\ (+\!\!+)\ [\,]$$

$$sortBy :: (\alpha \to \alpha \to Ordering) \to [\alpha] \to [\alpha]$$
$$\text{-- omitted}$$

$$partition :: (\alpha \to Bool) \to [a] \to ([a], [a])$$
$$partition\ p\ [\,] = [[\,], [\,]]$$
$$partition\ p\ (a : as) = \textbf{let}\ (ts, fs) = partition\ p\ as$$
$$\textbf{in if}\ p\ x\ \textbf{then}\ (a : ts, fs)\ \textbf{else}\ (ts, a : fs)$$

Fig. 1 Definitions of basic functions for lists

function *scan* (also called *prescan*) takes an associative binary operator with unit, initial value, and a list and returns a list of the same length whose elements are prefix sums of the input list.

The function *flatten* takes a list of lists (a nested list) and concatenates all the inner lists. The function *sortBy* takes a comparison function and sorts the elements in the input list in terms of the comparison function. The function *partition* takes a predicate and splits the input list into two lists where one list includes all the elements satisfying the predicate, and the other list includes all the others.

2.3 Bags (Multisets) and Functions Manipulating Bags

A bag (multiset) is an unordered sequence of elements of the same type. The type of bag with elements of type α is denoted by $Bag\ \alpha$ [1]. We may use Nil_{B} and $Cons_{\mathrm{B}}$ for pattern matching with an empty bag and an element in a bag similarly to the case of lists. Conversion from a list to a bag and vice versa are done by the functions *list2bag* and *bag2list*, where we assume that the elements are aligned in any order in the result list of *bag2list*.

We can define functions for bags similarly to the case of lists (suffix $_{\mathrm{B}}$ is added to these functions). We will use the following functions for bags.

$$
\begin{aligned}
&map_{\mathrm{B}} :: (\alpha \to \beta) \to Bag\ \alpha \to Bag\ \beta \\
&flatten_{\mathrm{B}} :: Bag\ [\alpha] \to Bag\ \alpha \\
&head_{\mathrm{B}} :: Bag\ \alpha \to \alpha \\
&sortBy_{\mathrm{B}} :: (\alpha \to \alpha \to Ordering) \to Bag\ \alpha \to [\alpha] \\
&partition_{\mathrm{B}} :: (\alpha \to Bool) \to Bag\ \alpha \to (Bag\ \alpha, Bag\ \alpha)
\end{aligned}
$$

3 MapReduce Programming Model in Nutshell

MapReduce [8] provides a simple data-parallel programming model suitable for processing large amounts of data. In this section, we briefly review the programming model [15] that is commonly behind Google's MapReduce and its variants. More details on Hadoop MapReduce will be given in the next section.

Figure 2 depicts a simple model of MapReduce computation. The input and output of MapReduce computation are put on a distributed filesystem (DFS), where data are split into smaller chunks. Each fragment of data forms a key-value pair throughout the MapReduce computation.

MapReduce computation consists of three phases: the Map phase, Shuffle phase, and Reduce phase[2].

[1] It is not trivial to give a definition of *Bag* supporting its nondeterministic behavior. One way is to define the structure in the same way as the list with simulating the behavior by permutation of elements.

[2] We will use the term "phase" for the models of computation and "task" for the implementation. In the implementation, MapReduce consists of two sets of tasks: the mapper tasks work from the input to the end of the Map phase, and the reducer tasks work from the Shuffle phase to the end of the output.

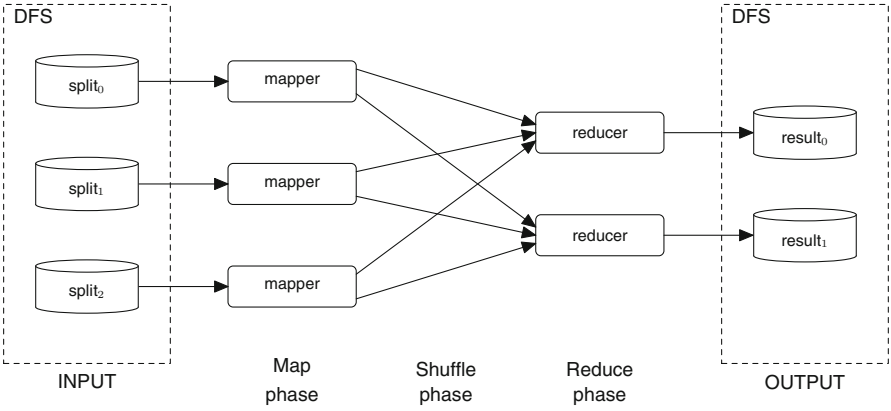

Fig. 2 Illustration of simple MapReduce model

- Map phase: For each fragment of data split by the system, a user-defined function *mapper* is applied independently in parallel. The function should take a key-value pair and return one or more (or possibly no) intermediate results as key-value pairs (the types of key and/or value can differ).
- Shuffle phase: The intermediate results of the same key are grouped and passed to the following reduce phase.
- Reduce phase: For each set of intermediate results of the same key, the user-defined function *reducer* is applied in parallel. The function should take a key and a list of values and generate one or more final results, which will be stored on the DFS.

The main task of programmers in the programming model of MapReduce is to provide the two parameter functions *mapper* and *reducer*. The MapReduce system is responsible for data distribution, communication, and fault tolerance.

In the MapReduce computation, there is another feature called Combiner, which could be inserted just after the Map phase. Combiners are useful for reducing network communication cost without changing the result. The use of Combiner has been widely discussed including the model by Lämmel [15]. We basically omit the discussion of Combiner from the functional models in the paper.

4 Low-Level Model of Hadoop MapReduce

In this section, we develop a functional model from the implementation of Hadoop MapReduce. In Hadoop MapReduce, there are a lot of parameters to control the execution and performance of MapReduce. Here, we will discuss some that mainly affect the computation (results).

4.1 Overview

The most important two classes in MapReduce programs are `Mapper` and `Reducer`, which specify the main computation in MapReduce programs. We start by giving their types.

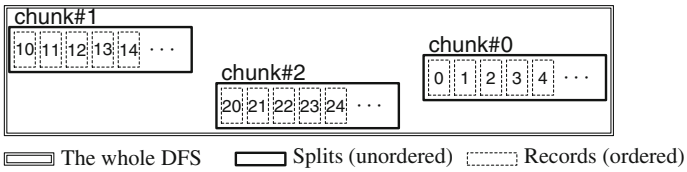

Fig. 3 Nested input-data model in Hadoop: Data are put on DFS

In Hadoop MapReduce, a large input file is divided in two stages (Fig. 3). Firstly, the whole of the data is divided into *splits* of a size that a single computer can deal with, and then each split is divided into smaller *records* (e.g. by lines). Splits often correspond to data chunks on the DFS; we cannot take care of the order among them. In contrast, a split is processed on a single computer, so we can mind the order among the records. This is the first key point in our model: we use a *list* for a set of records but a *bag* for a set of splits.

A *mapper* (`Mapper`) takes a split for its input: a list of key-value pairs. The output of the *mapper* is a list of key-value pairs where the types of keys and/or values may differ from those of the input.

$$mapper :: [(k_1, v_1)] \to [(k_2, v_2)]$$

Before the Reduce phase, values of the same key are arranged into a list (we will discuss later how these keys are evaluated and merged). In a similar way to the *mapper*, some of the intermediate results will be given to a *reducer*: the input type of the *reducer* becomes a *list* of pairs of a key and a *list* of values. The output of the *reducer* is also a list of key-value pairs with types that may be different.

$$reducer :: [(k_2, [v_2])] \to [(k_3, v_3)]$$

Other important parameters in Hadoop MapReduce programs are those for controlling the Shuffle phase. We can specify three classes (functions) through `setPartitionerClass`, `setSortComparatorClass`, and `setGroupingComparatorClass`. Hereafter, we denote the three functions set by the above three functions as *hashP*, *compS*, and *compG*, respectively. Note that the `Partitioner` class should have a function `getPartition` whose signature is[3]:

```
int getPartition(KEY k, VALUE v, int n),
```

and the other two `Comparator` classes should have a function `compare` whose signature is[4]:

```
int compare(byte[] b1, int s1, int l1,
            byte[] b2, int s2, int l2).
```

[3] The function `getPartition` (*hashP*) can take a value as well as a key. The author found hardly any applications in which the function uses the value. In the high-level model given in Sect. 5, we do not use the value for partitioning.

[4] Triples (`b1`, `s1`, `l1`) and (`b2`, `s2`, `l2`) represent two values to be compared and stored on byte streams.

```
mapReduceL :: ([(k1, v1)] → [(k2, v2)])                    -- mapper
            → ([(k2, [v2])] → [(k3, v3)])                  -- reducer
            → ((k2, v2) → Integer)                         -- hashP
            → (k2 → k2 → Ordering)                         -- compS
            → (k2 → k2 → Ordering)                         -- compG
            → Bag [(k1, v1)]                               -- input
            → Bag [(k3, v3)]                               -- output
mapReduceL mapper reducer hashP compG compS input
  = let aftMap = map_B mapper input
        bfrRed = shuffleMR hashP compG compS aftMap
    in  map_B reducer bfrRed
```

Fig. 4 Low-level model of MapReduce computation: *mapReduceL*

In accordance with these signatures, we specify the type of those functions as follows.

$$\begin{aligned} &\mathit{hashP} :: (k_2, v_2) \to \mathit{Integer} \\ &\mathit{compS} :: k_2 \to k_2 \to \mathit{Ordering} \\ &\mathit{compG} :: k_2 \to k_2 \to \mathit{Ordering} \end{aligned}$$

With these parameter functions, we give a rough model of the computation of Hadoop MapReduce as shown in Fig. 4. The following are the key points in this model.

- A Hadoop MapReduce program takes five parameter functions.
- The input and output consist of a bag of *splits* (unordered), and a split consists of a list of key-value pairs (ordered).
- The computation in Hadoop MapReduce consists of three phases, each given in each line in the definition.
- The two map_{B}'s represent the possibility of parallelism: since we do not care about the order among splits, we can compute in parallel.

In the following two subsections, we will give the details of the definitions of *shuffleMR*, *mapper*, and *reducer*.

4.2 Shuffle Phase

In the Shuffle phase, basically the key-value pairs that are output from *mapper* are grouped together based on their keys. In the implementation of this grouping, the key-value pairs are sorted by their keys. In fact, the Shuffle phase in Hadoop MapReduce consists of the following three subphases in this order.

1. Partitioning: For each key-value pair, the index of the reducer task that will receive it is computed using the parameter function *hashP*.
2. Sorting: All the key-value pairs in the same reducer task are sorted with respect to the comparison function *compS*.
3. Grouping: After the sorting, the key-value pairs that have the same key with respect to the comparison function *compG* are grouped together.

```
shuffleMR :: ((k2, v2) → Integer)                                  -- hashP
           → (k2 → k2 → Ordering)                                  -- compS
           → (k2 → k2 → Ordering)                                  -- compG
           → Bag[(k2, v2)]                                         -- input
           → Bag[(k2, [v2])]                                       -- output
shuffleMR hashP compS compG input
  = let aftP = grpByID $ map_B (applyW hashP) $ flatten_B input
        aftS = map_B (sortByKey compS) aftP
    in  map_B (grpByKey compG) aftS

grpByID :: Eq α ⇒ Bag (α, β) → Bag (Bag b)
grpByID Nil_B = Nil_B
grpByID (Cons_B a x) = let (xa, xo) = partition_B (\b → fst a == fst b) x
                       in Cons_B (map_B snd (Cons_B a xa)) (grpByID xo)

sortByKey :: (k2 → k2 → Ordering) → Bag (k2, v2) → [(k2, v2)]
sortByKey compS = sortBy_B (\(k, v)(k', v') → compS k k')

grpByKey :: (k2 → k2 → Ordering) → [(k2, v2)] → [(k2, [v2])]
grpByKey compG [] = []
grpByKey compG ((k, v) : x) = grpByKey' compG [] k [v] x
grpByKey' compG rs k vs [] = rs ++ [(k, vs)]
grpByKey' compG rs k vs ((k', v') : x)
    | compG k k' == EQ      = grpByKey' compG rs k' (vs ++ [v']) x
    | otherwise             = grpByKey' compG (rs ++ [(k, vs)]) k' [v'] x
```

Fig. 5 Low-level definition of Shuffle phase: *shuffleMR*

Figure 5 shows a definition of *shuffleMR*. In the first line that computes *aftP*, we compute the reducer index by *hashP* with which we group the key-value pairs. Then, in the following two lines, we apply sorting and grouping. Note the following three details.

The implementation of *grpByID* is different from that in Hadoop. In the real implementation, we know the number of reducer tasks, and we directly put the key-value pairs into the slot of the corresponding reducer index. Instead, our definition of *grpByID* is generic so that it can be used again later.

There is only one sorting subphase. Even when we would like to use so-called *secondary sorting*, sorting is executed only once and is executed *before* grouping. We will discuss this matter in the next section.

Our definition of *grpByKey* follows the implementation in Hadoop. Grouping by *grpByKey* assumes that the data are correctly sorted in advance. If not or if the comparison functions for sorting and grouping are inconsistent, then key-value pairs may not be fully merged. In addition, the last key is used for the next comparison, so we should be careful when we use a comparison function that does not satisfy transitivity (this is possible in real Hadoop). [5]

[5] If we use the definition $a \equiv b \Longleftarrow |a - b| < 2$, then the list [3, 4, 5, 7] will be grouped as [3, 4, 5] and [7], not [3, 4] and [5] and [7]. In Haskell, there is a function *Data.List.groupBy* :: $(\alpha \to \alpha \to Bool) \to [\alpha] \to [[\alpha]]$ that has similar functionality, but it returns [[3, 4], [5], [7]] for this case.

4.3 Map and Reduce Phases

In Hadoop MapReduce programs, users develop the *mapper* function by inheriting the following `Mapper` class.

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  protected void setup(Context context) { ... }
  protected void map(KEYIN key, VALUEIN value,
                     Context context) { ... }
  protected void cleanup(Context context) { ... }
}
```

In the computation of `Mapper`, the three functions are called:

- `setup` is called once for each split before processing the key-value pairs.
- `map` is called for each key-value pair.
- `cleanup` is called once for each split after processing the key-value pairs.

Usually and in the simplest case, users only provide the `map` function (we denote it as f_{map} to avoid confusion). In this case, the computation of the *mapper* is specified by the following function *mkMapper1*.

$$
\begin{aligned}
&\mathit{mkMapper1} :: ((k_1, v_1) \to [(k_2, v_2)]) && --f_{\text{map}}\\
&\qquad \to [(k_1, v_1)] \to [(k_2, v_2)] && --\text{input/output}\\
&\mathit{mkMapper1}\ f_{\text{map}} = \mathit{flatten} \circ \mathit{map}\ f_{\text{map}}
\end{aligned}
$$

We can include attributes when we inherit the `Mapper` class, and here we may want to use the attributes for the computation. More concretely, we set the initial value in the `setup` function, then update the value of attributes at every call of the `map` function, and finally output the accumulated value in the `cleanup` function. Such a computation can be specified using the *foldl* function on lists. The following function *mkMapper2* takes three parameter functions corresponding to `setup`, `map`, and `cleanup` and performs the computation using the attributes.

$$
\begin{aligned}
&\mathit{mkMapper2} :: \mathit{att} && --f_{\text{setup}}\\
&\qquad \to (\mathit{att} \to (k_1, v_1) \to \mathit{att}) && --f_{\text{map}}\\
&\qquad \to (\mathit{att} \to [(k_2, v_2)]) && --f_{\text{cleanup}}\\
&\qquad \to [(k_1, v_1)] \to [(k_2, v_2)] && --\text{input/output}\\
&\mathit{mkMapper2}\ f_{\text{setup}}\ f_{\text{map}}\ f_{\text{cleanup}} = f_{\text{cleanup}} \circ \mathit{foldl}\ f_{\text{map}}\ f_{\text{setup}}
\end{aligned}
$$

We can also combine these two. We use attributes for accumulating some information through the list of key-value pairs, and at the same time output key-value pairs from the `map` function. The results of f_{map} are the updated attributes and intermediate results added to the result list[6]. Here, the types of intermediate results from f_{map} and

[6] Readers who know Haskell well would write the definition with a state-monadic function.

f_{cleanup} should be the same $[(k_2, v_2)]$.

$$
\begin{aligned}
&\mathit{mkMapper3} :: \mathit{att} && -- f_{\text{setup}}\\
&\quad \to (\mathit{att} \to (k_1, v_1) \to (\mathit{att}, [(k_2, v_2)])) && -- f_{\text{map}}\\
&\quad \to (\mathit{att} \to [(k_2, v_2)]) && -- f_{\text{cleanup}}\\
&\quad \to [(k_1, v_1)] \to [(k_2, v_2)] && --\text{input/output}\\
&\mathit{mkMapper3}\ f_{\text{setup}}\ f_{\text{map}}\ f_{\text{cleanup}}\ \mathit{xs}\\
&\quad = \textbf{let}\ a = f_{\text{setup}}\\
&\qquad\quad (a', \mathit{ys}) = \mathit{aux}\ f_{\text{map}}\ a\ [\,]\ \mathit{xs}\\
&\qquad \textbf{in}\ \ \mathit{ys} +\!\!+ f_{\text{cleanup}}\ a'\\
&\qquad \textbf{where}\ \mathit{aux}\ f_{\text{map}}\ a\ \mathit{ys}\ [\,] = (a, \mathit{ys})\\
&\qquad\qquad \mathit{aux}\ f_{\text{map}}\ a\ \mathit{ys}\ (\mathit{kv} : \mathit{xs}) = \textbf{let}\ (a', \mathit{ys}') = f_{\text{map}}\ a\ \mathit{kv}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ \mathit{aux}\ f_{\text{map}}\ a'\ (\mathit{ys} +\!\!+ \mathit{ys}')\ \mathit{xs}
\end{aligned}
$$

We can develop the *reducer* function in the same manner as that for the *mapper* function, except for the type of f_{reduce} that takes a pair of a key and a list of values.

$$f_{\text{reduce}} :: (k_2, [v_2]) \to [(k_3, v_3)]$$

Here, we only show the most simple case *mkReducer1* that generates the *reducer* function from the parameter function f_{reduce}. Note that we can easily extend it to *mkReducer2* or *mkReducer3* in the same way as we did for *mapper*.

$$
\begin{aligned}
&\mathit{mkReducer1} :: ((k_2, [v_2]) \to [(k_3, v_3)]) && -- f_{\text{reduce}}\\
&\quad \to [(k_2, [v_2])] \to [(k_3, v_3)] && --\text{input/output}\\
&\mathit{mkReducer1}\ f_{\text{reduce}} = \mathit{flatten} \circ \mathit{map}\ f_{\text{reduce}}
\end{aligned}
$$

4.4 Including Combiner

In MapReduce programming, a combiner can be inserted between the Map phase and the Shuffle phase. It merges some key-value pairs from the preceding `Mapper` and improves the performance by reducing the amount of intermediate data communicated through the network.

We can embed the combiner into our model just by the function composition of *mapper* and *combiner*; $\mathit{mapper}' = \mathit{combiner} \circ \mathit{mapper}$. To support this, the type of *combiner* should be as follows.

$$\mathit{combiner} :: [(k_2, v_2)] \to [(k_2, v_2)]$$

Here is a simplified definition of *mkCombiner1* that shows the computation with *combiner*. To make the definition simple, we assume that we can check the equality of keys. In the real implementation, this grouping is executed with `Comparators` as in *shuffleMR*.

$$
\begin{aligned}
&mkCombiner1 :: \mathsf{Eq}\ k_2 \\
&\qquad \Rightarrow ((k_2, [v_2]) \rightarrow [(k_2, v_2)]) && --f_{\text{combine}} \\
&\qquad \rightarrow [(k_2, v_2)] \rightarrow [(k_2, v_2)] && --\text{input/output} \\
&mkCombiner1\ f_{\text{combine}}\ input = flatten\ \$\ map\ f_{\text{combine}}\ \$\ grpByID'\ input
\end{aligned}
$$

$$
\begin{aligned}
&grpByID' :: \mathsf{Eq}\ k \Rightarrow [(k, v)] \rightarrow [(k, [v])] \\
&grpByID'\ [\,] \qquad\qquad = [\,] \\
&grpByID'\ ((k, v) : x) \quad = \textbf{let}\ (xa, xo) = partition\ (\backslash(k', v') \rightarrow k == k')\ x \\
&\qquad\qquad\qquad\qquad\ \ \textbf{in}\ (k, v : map\ snd\ xa) : grpByID'\ xo
\end{aligned}
$$

4.5 Word Count Example on Low-Level Model

Here, we briefly show the program for a well-known word-count application. In this simple application, we can develop a MapReduce program using the simple *mkMapper1* and *mkReducer1*. For the partitioning, we gather the intermediate data on a single reducer task (*partitionWC* always returns 1). For the sorting and grouping, we need nothing special (we used a normal *compare* function for [*Char*]).

$$
\begin{aligned}
&wc :: Bag\ [(Int, [Char])] \rightarrow Bag\ [([Char], Int)] \\
&wc = mapReduceL\ mapperWC\ reducerWC \\
&\qquad\qquad partitionWC\ compare\ compare \quad --\text{for Shuffle phase}
\end{aligned}
$$

$$
\begin{aligned}
&mapperWC = mkMapper1\ (\backslash(\_, w) \rightarrow [(w, 1)]) \\
&reducerWC = mkReducer1\ (\backslash(w, as) \rightarrow [(w, foldl\ (+)\ 0\ as)]) \\
&partitionWC\ (\_, \_) = 1
\end{aligned}
$$

5 High-Level Model of Hadoop MapReduce

The model in Sect. 4 was developed based on the implementation of Hadoop MapReduce. Its Shuffle phase, however, is not easy to handle for two reasons. The first reason is that we require one hash function and two comparison functions, *compS* and *compG*, for a single key, and they should be consistent:

$$
\begin{aligned}
&compS\ a\ b = EQ \Longrightarrow compG\ a\ b = EQ\ , \text{and} \\
&compG\ a\ b = EQ \Longrightarrow hashP\ (a, \_) = hashP\ (b, \_).
\end{aligned}
$$

The second reason is the order of subphases: sorting followed by grouping. It is more intuitive and easier to understand if the three subphases are executed as:

1. We partition the whole data for reducer tasks,
2. We then group the key-value pairs inside a single reducer task, and
3. Finally we sort the key-value pairs inside each group.

These steps of execution are often called *secondary sorting*. Note that in the implementation of Hadoop MapReduce, there is no sorting subphase after the grouping, and thus there is a gap between the above steps of execution and the real implementation.

In this section, we propose another model for the Shuffle phase (and for MapReduce) to bridge the gap. The key idea is to introduce three keys instead of using a single key k_2 for intermediate data: k_P for partition, k_G for grouping, and k_S for sorting. Therefore, now we assume that the intermediate data passed from *mapper* to *reducer* are in the form $((k_P, k_G, k_S), v_2)$. Here, we assume that k_P belongs to the Eq class and k_G and k_S belong to the Ord class. The following are three functions used to extract the keys.

$$\begin{aligned} getP\ ((k_P, k_G, k_S), v_2) &= k_P \\ getG\ ((k_P, k_G, k_S), v_2) &= k_G \\ getS\ ((k_P, k_G, k_S), v_2) &= k_S \end{aligned}$$

With the extended key-value pairs, we can develop a new definition for the Shuffle phase in Fig. 6. Note that we have the key-value pairs sorted after the grouping in this definition. There are two sortings in this definition: one is for sorting the groups and the other is for sorting inside the groups.

This *shuffleMR2* can work as a high-level model of the Shuffle phase because we can derive the functions required in the low-level model as follows.

$$\begin{aligned} &hashP2\ ((k_P, k_G, k_S), v2) = toInt\ k_P \\ &compS2\ (k_P, k_G, k_S)\ (kP', kG', kS') \mid k_G == k'_G = compare\ k_S\ k'_S \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \textbf{otherwise} = compare\ k_G\ k'_G \\ &compG2\ (k_P, k_G, k_S)\ (kP', kG', kS') = compare\ k_G\ k'_G \end{aligned}$$

Here, the function *toInt* converts a value k_P into an integer. Note that the comparison functions satisfy the consistency condition: $compS\ a\ b = EQ \Longrightarrow compG\ a\ b = EQ$ (the condition for *hashP* should be guaranteed by the user).

```
shuffleMR2 :: Eq kP ⇒ Ord kG ⇒ Ord kS                -- (requirements)
           ⇒ Bag [((kP, kG, kS), v2)]                -- input
           → Bag [((kP, kG, kS), [v2])]              -- output
shuffleMR2 input
  = let
      aftP = grpByID $ map_B (applyW getP) $ flatten_B input
      aftG = map_B (sortBy_B compGS ∘ grpByID ∘ map_B (applyW getG)) aftP
          where compGS = comparing (getG ∘ head_B)
    in map_B (map sortMerge) aftG
          where sortMerge xs = let ss = sortBy_B compSS xs
                               in (fst $ last ss, map snd ss)
                compSS = comparing getS
```

Fig. 6 High-level definition of Shuffle phase: *shuffleMR2*

```
mapReduceH :: Eq k_P ⇒ Ord k_G ⇒ Ord k_S                    -- (requirements)
            ⇒ ([(k_1, v_1)] → [((k_P, k_G, k_S), v_2)])     -- mapper
            → ([((k_P, k_G, k_S), [v_2])] → [(k_3, v_3)])   -- reducer
            → Bag [(k_1, v_1)] → Bag [(k_3, v_3)]           -- input / output
mapReduceH mapper reducer input
  = let aftMap = map_B mapper input
        bfrRed = shuffleMR2 aftMap
    in  map_B reducer bfrRed
```

Fig. 7 High-level model of MapReduce computation: *mapReduceH*

A high-level functional model of Hadoop MapReduce is given with this *shuffleMR2* as shown in Fig. 7. In this model, we have requirements of type classes but fewer parameters are required than in the low-level model.

Here, we briefly show the program for a well-known word-count application. The *mapper* and *reducer* are almost the same as those for the low-level model. Since we explicitly need three keys, the f_{map} function sets $(1, w, 1)$ for the key of intermediate results.

$$
\begin{aligned}
&wc2 :: Bag\ [(Int, [Char])] \to Bag\ [([Char], Int)] \\
&wc2 = mapReduceH\ mapperWC2\ reducerWC2 \\
&mapperWC2 = mkMapper1\ (\backslash(\_, w) \to [((1, w, 1), 1)]) \\
&reducerWC2 = mkReducer1\ (\backslash((\_, w, \_), as) \to [(w, foldl\ (+)\ 0\ as)])
\end{aligned}
$$

6 Implementing List Scan on MapReduce Model

Scan (prefix sums) is an accumulative computation on lists. It not only has a lot of direct applications [6] but also plays an important role in calculating programs [12]. It is thought to be hard to implement it on MapReduce due to the following reasons:

- The data to manipulate are in a list. We need not only to put an index on each element but also to know how data are manipulated, which is concealed in the conventional MapReduce model.
- The computation also depends on the order among the elements. Usual MapReduce computation takes an associative and *commutative* operation but the computation of scan is inherently noncommutative.

In this section, two algorithms are developed on the functional model proposed in Sect. 5.

6.1 Two Well-Known Algorithms of Scan

There are several parallel algorithms of scan [6]. Here, we introduce two well-known ones.

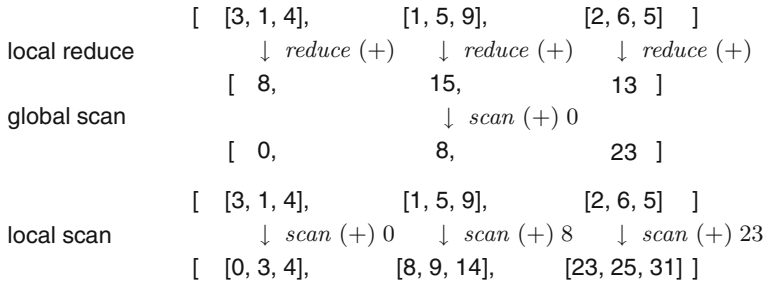

Fig. 8 Scan algorithm with three phases: *scanDist*

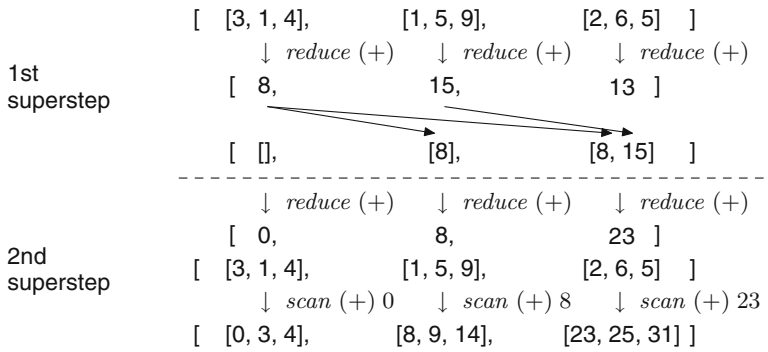

Fig. 9 Scan algorithm on BSP model: *scanBSP*

The first one is often used on distributed-memory environments. The computation consists of three phases: local reduction, global scan, and local scan (Fig. 8).

$$
\begin{aligned}
&scanDist :: (a \to a \to a) \to a \to [[a]] \to [[a]] \\
&scanDist\ (\oplus)\ e\ xss \\
&\quad = \textbf{let}\ ys = map\ (foldl\ (\oplus)\ \iota_{\oplus})\ xss && \text{--local reduce} \\
&\qquad\ zs = scan\ (\oplus)\ e\ ys && \text{--global scan} \\
&\quad \textbf{in}\ map\ scanl'\ (zip\ xss\ zs) && \text{--local scan} \\
&\quad \textbf{where}\ scanl'\ (xs, z) = scan\ (\oplus)\ z\ xs
\end{aligned}
$$

The second one is a well-known algorithm on the BSP (bulk synchronous parallel) model [24]. It consists of two supersteps (Fig. 9). In the first superstep we apply local reduce and send the results to all the processors on the right; in the second superstep we reduce the received results and apply a local scan. In the following specification of *scanBSP*, p is the number of processors used. Note that the computation of *zss* corresponds to the communication in the first superstep.

$$
\begin{aligned}
&scanBSP :: (a \to a \to a) \to a \to Int \to [[a]] \to [[a]] \\
&scanBSP\ (\oplus)\ e\ p\ xss
\end{aligned}
$$

```
scanDistMR (⊕) e input
  = let Cons_B [(_, gs)] _ = mapReduceH mapper1 reducer1 input
    in  mapReduceH (mapper2 gs) reducer2 input
    where
      mapper1 = mkMapper2 (0, ι⊕)                              -- f_setup
                          (\(l, s)(k, v) → (k, s ⊕ v))         -- f_map
                          (\(l, s) → [((1, 1, l), s)])         -- f_cleanup
      reducer1 = mkReducer1 (\(_, ss) → [(1, scan (⊕) e ss)])
      mapper2 gs = mkMapper3 ι⊕                                -- f_setup
                             (\s(k, v) → let p = div k 3       -- f_map
                                             v' = (gs !! p) ⊕ s
                                         in  (s ⊕ v, [((p, k, 1), v')]))
                             (\s → [])                         -- f_cleanup
      reducer2 = mkReducer1 (\((_, k, _), vs) → [(k, head vs)])
```

Fig. 10 Two-Pass MapReduce implementation for *scanDist*

$$
\begin{aligned}
&= \textbf{let } ys = map\ (foldl\ (\oplus)\ \iota_\oplus)\ xss \\
&\qquad zss = map\ (\lambda p.\ take\ p\ ys)\ [0..(p-1)] && \text{--1st superstep} \\
&\quad \textbf{in } map\ scan'\ (zip\ xss\ zss) && \text{--2nd superstep} \\
&\quad \textbf{where } scan'\ (xs, zs) = scan\ (\oplus)\ (foldl\ (\oplus)\ e\ zs)\ xs
\end{aligned}
$$

6.2 Two-Pass MapReduce Implementation for *scanDist*

To represent a list with a bag, it is a common technique to pair a value with its index. We assume that the input is a bag of lists of key-value pairs, whose key is a (global) index and the value is an element. We also assume that consecutive elements are put in a list in order. For example, the nested list in Fig. 8 may be given as follows. Note that the splits can be out of order. In this section, we denote bags with { } for better readability.

$$\{\ [(3, 1), (4, 5), (5, 9)],\ \ [(6, 2), (7, 6), (8, 5)],\ \ [(0, 3), (1, 1), (2, 4)]\ \}$$

For these inputs, we can compute the list scan by two-pass MapReduce computation (Fig. 10).

The Map phase in the first MapReduce corresponds to the local reduce, which generates (after flattening)

$$\{\ ((1, 1, 5), 15)),\ \ ((1, 1, 8), 13)),\ \ ((1, 1, 2), 8)\ \}\ .$$

These key-value pairs are grouped together by the first two keys and sorted by the third key, generating ((1, 1, 8), [8, 15, 13]). The Reduce phase in the first MapReduce corresponds to the global scan, which results in {(_, [0, 8, 23])}. Let the value part be bound by *gs*.

```
scanBSPMR (⊕) e pp xb
  = mapReduceH mapper reducer xb
    where
      mapper = mkMapper3 f_setup f_map f_cleanup
      f_setup = (0, ι_⊕)
      f_map (_, s) (k, v) = let p = div k 3
                            in ((p, s ⊕ v), [((p, 1, k), v)])
      f_cleanup (p, s) = map (\p' → ((p', 1, −1000 + p), s)) [p + 1..pp − 1]
      reducer = mkReducer1 f_reduce
      f_reduce ((p, _, _), vs) = let zs = take p vs
                                     xs = drop p vs
                                 in zip [(3 ∗ p)..(3 ∗ p + 2)]
                                        (scan (⊕) (foldl (⊕) e zs) xs)
```

Fig. 11 One-Pass MapReduce implementation for *scanBSP*

The Map phase in the second MapReduce corresponds to the local scan. Here, each map task selects the value in *gs* based on the index of the input. The result of this Map phase is

$$\{\ [((1,3,1),8),((1,4,1),9),((1,5,1),14)],\\ [((2,6,1),23),((2,7,1),25,((2,8,1),31)],\\ [((0,0,1),0),((0,1,1),3),((0,2,1),4)]\ \}$$

The Reduce phase in the second MapReduce just retrieves the index and the result value.

6.3 One-Pass MapReduce Implementation for *scanBSP*

We show another MapReduce algorithm for the scan algorithm on the BSP model. We assume that the input is in the same form as in the case of MapReduce implementation for *scanDist*.

The implementation of the BSP-inspired scan algorithm is shown in Fig. 11 where the parameter *pp* is the number of BSP processes. We hard-coded the number of key-value pairs to be 3 for simplicity, but we can resolve it by using a hash table etc. for indices. The implementation consists of a single MapReduce. The Map phase corresponds to the local computation in the first superstep, the Shuffle phase corresponds to the communication in the first superstep, and the Reduce phase corresponds to the second superstep.

In the Map phase, intermediate values are generated from both f_{map} and f_{cleanup}. Those from f_{map} correspond to the data of the original array and are needed since we cannot read the input data in the Reduce phase. Those from f_{cleanup} correspond to the result of local reduction and will be sent to all the reducer tasks with a larger ID. For example, the *mapper* processing [(0, 3), (1, 1), (2, 4)] generates

- [((0, 1, 0), 3), ((0, 1, 1), 1), ((0, 1, 2), 4)] for the reducer task 0 and
- [((1, 1, −1000), 8), ((2, 1, −1000), 8)] for the reducer tasks 1 and 2.

After the Shuffle phase, we will have the following input for the reduce phase.

$$\{\ [((1,1,5),[8,1,5,9])],\ \ [((2,1,8),[8,15,2,6,5])],\ \ [((0,1,2),[3,1,4])]\ \}$$

Each reducer task takes a singleton list of a key-value pair whose value includes all the information needed. By applying the computation corresponding to the second superstep in the BSP algorithm, we obtain the result. The final *zip* function is for assigning the global index.

$$\{\ [(3,8),(4,9),(5,14)],\ \ [(6,23),(7,25),(8,31)],\ \ [(0,0),(1,3),(2,4)]\ \}$$

7 Related Work

7.1 Modeling MapReduce

As far as the author knows, the first functional specification of MapReduce was formulated by Lämmel [15], and it has been referred to the most. It provides a good abstraction of MapReduce, where discussion of types of parameter functions and input/output models are appropriate. Berthold et al. [3] developed a Haskell implementation (model) of MapReduce where the basic model followed Lämmel's model and lacked the detailed Shuffle phase with sorting. Since the paper was published before the open-source implementation of Hadoop became widely available, there are some points included in Hadoop but excluded in the model. This was the first motivation with which the author started formalizing the models in this paper.

Some researchers discussed the models of the MapReduce computation from an algorithmic point of view [11,14]. They focused on specifying the class of algorithms that can be dealt with efficiently by MapReduce and had a deep interest in the space/time complexity. Their models were very abstracted from the MapReduce implementations, so it is not straightforward to apply them directly to program development. Recently, a more realistic model [20] was discussed in relation to the BSP (bulk synchronous parallel) model [24]. The BSP model and MapReduce computation have some common points. In Sect. 5, we saw the relation between BSP and MapReduce with an example of implementing scan.

One important application of the functional models is proof of correctness or some other properties. For example, Dörre et al. [9] developed a type-checking system that finds type errors that have not been caught at the compilation of Hadoop MapReduce programs. Ono et al. [19] used the Coq proof assistant for the proof of correctness. Using the code-generator mechanism in the Coq proof assistant, certified MapReduce programs are given from proved specifications. There is also a study on the proof system for the correctness of the use of the `Combiner` function [23].

Another important application of the functional models includes cost models. With cost models, we can find performance bugs and optimize programs before execution or after little profiling. Such a cost model was also given by Dörre et al. [10].

There have been other studies that specify the MapReduce computation on more formal models. Two such studies were done on communicating sequential processes (CSP) [26] and with the Event-B method [21].

7.2 Differences from Previous Work

Among several studies, two studies by Lämmel [15] and Dörre et al. [10] gave detailed functional models of MapReduce computation.

The former [15] first pointed out the difference between map/reduce (tasks) in MapReduce and *map*/*reduce* (functions) in functional programming and introduced the idea of map (dictionary) data structure. Although the idea behind MapReduce was clearly stated, the model assumed that the data are divided in a single stage. As we modeled in this paper, the MapReduce framework combines unordered parts with ordered parts for parallelism and performance.

The latter [10] was the most detailed model of MapReduce and included the partitioner, comparator, and grouping mechanisms in Hadoop. They used nested lists for input/output data to follow the data model in Hadoop, and the unordered property, which is important for parallelism, was not addressed well. It would be no problem for the cost model, but for proving the correctness of programs, this property is important.

In this paper, we started from the implementation in Hadoop MapReduce, and we also discussed the relationship between the secondary-sorting technique and the sorting in the implementation. As far as the author knows, there is no formal model that discusses this secondary-sorting technique.

8 Conclusion

In this paper, we have proposed two functional models of Hadoop MapReduce. The low-level model was developed based on the implementation of Hadoop MapReduce, and the high-level model provides a better understanding of the Shuffle phase. As shown in the implementation of scan algorithms, our model can be used to develop a nontrivial algorithm on Hadoop MapReduce. As far as the author knows, the last MapReduce implementation of BSP-inspired scan algorithm was not published.

As we discussed in the introduction, functional models are important for several applications. Among them, we would like to develop a formal model using the Coq proof assistant based on these models in this paper and prove the correctness of several nontrivial applications on Hadoop.

Acknowledgements The author thanks Yu Liu, Kento Emoto, and Le-Duc Tung for helpful discussion on the implementation of scans. In particular, the BSP-inspired algorithm on MapReduce was first suggested by Le-Duc Tung. Part of this work was conducted as part of the PaPDAS Project supported by ANR (ANR-2010-INTB-0205-02) and JST (10102704).

References

1. Apache Software Foundation: Hadoop. http://hadoop.apache.org/ (April 2015)

2. Apache Software Foundation: Hadoop Wiki: PoweredBy. http://wiki.apache.org/hadoop/PoweredBy (April 2015)
3. Berthold, J., Dieterle, M., Loogen, R.: Implementing parallel Google Map-Reduce in Eden. In: Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25–28, 2009. Proceedings, Lecture Notes in Computer Science. Springer, Berlin, vol. **5704**, pp. 990–1002 (2009)
4. Bird, R.: Introduction to Functional Programming using Haskell. Prentice-Hall, New York (1998)
5. Bird, R.S.: An introduction to the theory of lists. In: Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design, pp. 5–42. Springer, New York, Inc. (1987)
6. Blelloch, G.E.: Scans as primitive parallel operations. IEEE Trans. Comput. **38**(11), 1526–1538 (1989)
7. Breshears, C.: The Art of Concurrency. Oreilly & Associates Inc, Sebastopol (2009)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: 6th Symposium on Operating System Design and Implementation (OSDI2004), December 6–8, 2004, San Francisco, California, USA, pp. 137–150 (2004)
9. Dörre, J., Apel, S., Lengauer, C.: Static type checking of Hadoop MapReduce programs. In: Proceedings of the Second International Workshop on MapReduce and Its Applications (MapReduce '11), ACM, New York, pp. 17–24 (2011)
10. Dörre, J., Apel, S., Lengauer, C.: Modeling and optimizing MapReduce programs. Concurr. Comput. Pract. Exp. **27**(7), 1734–1766 (2014)
11. Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., Svitkina, Z.: On distributing symmetric streaming computations. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08), Society for Industrial and Applied Mathematics, pp. 710–719 (2008)
12. Hu, Z., Iwasaki, H., Takeichi, M.: Calculating accumulations. New Gener. Comput. **17**, 153–173 (1999)
13. Jiang, F., Tanabe, Y., Honiden, S.: Verification of Hadoop MapReduce application and Scala program extraction using Coq. IEICE Jpn. J. D **J97–D**(3), 625–634 (2014)
14. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10). Society for Industrial and Applied Mathematics, pp. 938–948 (2010)
15. Lämmel, R.: Google's MapReduce programming model—revisited. Sci. Comput. Program. **70**(1), 1–30 (2008)
16. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. SIGMOD Record **40**(4), 11–20 (2012)
17. Loulergue, F., Gava, F., Kosmatov, N., Lemerre, M.: Towards verified cloud computing environments. In: Smari, W.W., Zeljkovic, V. (eds.) 2012 International Conference on High Performance Computing & Simulation, HPCS 2012, pp. 91–97. IEEE, Silver Spring, MD (2012)
18. Ogawa, H., Nakada, H., Takano, R., Kudoh, T.: SSS: An implementation of key-value store based MapReduce framework. In: Proceedings of 2nd International Conference on Cloud Computing Technology and Science, pp. 745–761 (2010)
19. Ono, K., Hirai, Y., Tanabe, Y., Noda, N., Hagiya, M.: Using Coq in specification and program extraction of Hadoop MapReduce applications. In: Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM'11), pp. 350–365. Springer, Berlin (2011)
20. Pace, M.F.: BSP vs MapReduce. Procedia Comput. Sci. **9**, 246–255 (2012)
21. Pereverzeva, I., Butler, M., Fathabadi, A.S., Laibinis, L., Troubitsyna, E.: Formal derivation of distributed MapReduce. Tech. Rep. 1099, TUCS (2014)
22. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07), pp. 13–24. IEEE Computer Society, Silver Spring, MD (2007)
23. Suenaga, K.: Personal communication
24. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
25. White, T.: Hadoop: The Definitive Guide. O'Reilly Media/Yahoo Press, Sebastopol (2012)
26. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing MapReduce with CSP. In: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '10). IEEE Computer Society, Silver Spring, MD pp. 358–367 (2010)
27. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10), pp. 10. USENIX Association, Berkeley (2010)