Towards Systematic Parallel Programming over MapReduce

Yu Liu[1], Zhenjiang Hu[2], and Kiminori Matsuzaki[3]

[1] The Graduate University for Advanced Studies, Japan
yuliu@nii.ac.jp
[2] National Institute of Informatics, Japan
hu@nii.ac.jp
[3] School of Information, Kochi University of Technology, Japan
matsuzaki.kiminori@kochi-tech.ac.jp

Abstract. MapReduce is a useful and popular programming model for data-intensive distributed parallel computing. But it is still a challenge to develop parallel programs with MapReduce systematically, since it is usually not easy to derive a proper divide-and-conquer algorithm that matches MapReduce. In this paper, we propose a homomorphism-based framework named Screwdriver for systematic parallel programming with MapReduce, making use of the program calculation theory of list homomorphisms. Screwdriver is implemented as a Java library on top of Hadoop. For any problem which can be resolved by two sequential functions that satisfy the requirements of the third homomorphism theorem, Screwdriver can automatically derive a parallel algorithm as a list homomorphism and transform the initial sequential programs to an efficient MapReduce program. Users need neither to care about parallelism nor to have deep knowledge of MapReduce. In addition to the simplicity of the programming model of our framework, such a calculational approach enables us to resolve many problems that it would be nontrivial to resolve directly with MapReduce.

1 Introduction

Google's MapReduce [5] is a programming model for data-intensive distributed parallel computing. It is the de facto standard for large scale data analysis, and has emerged as one of the most widely used platforms for data-intensive distributed parallel computing.

Despite the simplicity of MapReduce, it is still a challenge for a programmer to systematically solve his or her (nontrivial) problems. Consider the *maximum prefix sum* problem of a sequence. For instance, if the input sequence is

$$\underline{3, -1, 4, 1, -5, 9, 2}, -6, 5$$

the maximum of the prefix sums should be 13 to which the underlined prefix corresponds. It is not obvious how to solve this problem efficiently with MapReduce (and we encourage the reader to pause to think how to solve this). Such problems widely exist in the real world, e.g, financial time series analysis.

Our basic idea to resolve such problems is to wrap MapReduce with *list homomorphisms* (or homomorphisms for short) [2]. We propose a simpler programming model based on the theory of list homomorphisms, and implement an associated framework called *Screwdriver*[1] for systematic parallel programming over MapReduce. Screwdriver provides users with an easy-to-use programming interface, where users just need to write two sequential programs: one for solving the problem itself and the other for solving the inverse problem. By building an algorithmic parallelization layer upon MapReduce, Screwdriver automatically generates homomorphisms from user-specific programs and efficiently executes them with MapReduce. We implemented this homomorphism-based framework efficiently in Java on top of an open source MapReduce framework Hadoop[2].

The rest of the paper is organized as follows. In Section 2 we review the concept of MapReduce and the theory of homomorphisms. The design and implementation of the homomorphism-based algorithmic layer on top of MapReduce are illustrated in Section 3. Then, we demonstrate the usefulness of our system with the maximum prefix sum problem above in Section 4, and report some experiment results in Section 5. The conclusion and highlight of future work are summarized in Section 6.

2 MapReduce and List Homomorphisms

The notations are mainly based on the functional language Haskell [1]. 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 two operators $\circ$ and $\vartriangle$ over functions: by definition, $(f \circ g)\ x = f\ (g\ x)$ and $(f \vartriangle g)\ x = (f\ x, g\ x)$. 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.

We denote lists with square brackets. We use $[\,]$ to denote an empty list, and $+\!\!+$ to denote the list concatenation: $[3, 1, 4] +\!\!+ [1, 5] = [3, 1, 4, 1, 5]$. A list that has only one element is called a *singleton*. Operator $[\cdot]$ takes a value and returns a singleton list with it.

2.1 MapReduce and MapReduce Programming Model

Figure 1 depicts the MapReduce computation model. The input and output data of MapReduce are managed as a *set* of key-value pairs and stored in distributed file system over a cluster. MapReduce computation mainly consists of three phases: the MAP phase, SHUFFLE & SORT phase, and the REDUCE phase[3].

[1] It is available online http://code.google.com/p/screwdriver/

[2] http://hadoop.apache.org/

[3] For readability, we use MAP and REDUCE to denote the phases in MapReduce, and f_{MAP} and f_{REDUCE} for the parameter functions used in the MAP and REDUCE phases. When unqualified, *map* and *reduce* refer to the functions of Haskell.

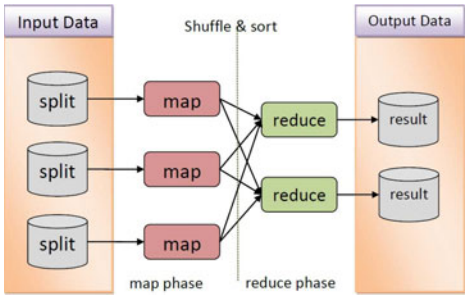

Fig. 1. MapReduce Computation

In the MAP phase, each input key-value pair is processed independently and a list of key-value pairs will be produced. Then in the SHUFFLE & SORT phase, the key-value pairs are grouped based on the key. Finally, in the REDUCE phase, the key-value pairs of the same key are processed to generate a result.

To make the discussion precise, we introduce a specification of the MapReduce programming model in a functional programming manner. Note that the specification in this paper is based on that in [11] but is more detailed. In this model, users need to provide four functions to develop a MapReduce application. Among them, the f_{MAP} and f_{REDUCE} functions performs main computation.

- Function f_{MAP}.
  $$f_{\text{MAP}} :: (k_1, v_1) \rightarrow [(k_2, v_2)]$$
  This function is invoked during the MAP phase, and it takes a key-value pair and returns a list of intermediate key-value pairs.
- Function f_{SHUFFLE}.
  $$f_{\text{SHUFFLE}} :: k_2 \rightarrow k_3$$
  This function is invoked during the SHUFFLE&SORT phase, takes a key of an intermediate key-value pair, and generate a key with which the key-value pairs are grouped.
- Function f_{SORT}.
  $$f_{\text{SORT}} :: k_2 \rightarrow k_2 \rightarrow \{-1, 0, 1\}$$
  This function is invoked during the SHUFFLE&SORT phase, and compares two keys in sorting the values.
- Function f_{REDUCE}.
  $$f_{\text{REDUCE}} :: (k_3, [v_2]) \rightarrow (k_3, v_3)$$
  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.

Now a functional specification of the MapReduce framework can be given as follows, which accepts four functions f_{MAP}, f_{SHUFFLE}, f_{SORT}, and f_{REDUCE} and transforms a set of key-value pairs to another set of key-value pairs.

$$\begin{aligned}
&\mathit{MapReduce} :: ((k_1, v_1) \to [(k_2, v_2)]) \to (k_2 \to k_3) \to (k_2 \to k_2 \to \{-1, 0, 1\}) \\
&\qquad\qquad\qquad \to ((k_3, [v_2]) \to (k_3, v_3)) \to \{(k_1, v_1)\} \to \{(k_3, v_3)\} \\
&\mathit{MapReduce}\ f_{\text{MAP}}\ f_{\text{SHUFFLE}}\ f_{\text{SORT}}\ f_{\text{REDUCE}}\ \mathit{input} \\
&\quad = \mathbf{let}\ \mathit{sub1} = \mathit{map}_{\mathrm{S}}\ f_{\text{MAP}}\ \mathit{input} \\
&\qquad\quad\ \ \mathit{sub2} = \mathit{map}_{\mathrm{S}}\ (\lambda(k', \mathit{kvs}).\ (k', \mathit{map}\ \mathit{snd}\ (\mathit{sortByKey}\ f_{\text{SORT}}\ \mathit{kvs}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{shuffleByKey}\ f_{\text{SHUFFLE}}\ \mathit{sub1}) \\
&\qquad \mathbf{in}\ \mathit{map}_{\mathrm{S}}\ f_{\text{REDUCE}}\ \mathit{sub2}
\end{aligned}$$

Function $\mathit{map}_{\mathrm{S}}$ is a set version of the *map* function: i.e., it applies the input function to each element in the set. Function *shuffleByKey* takes a function f_{SHUFFLE} and a set of a list of key-value pairs, flattens the set, and groups the key-value pairs based on the new keys computed by f_{SHUFFLE}. The result type after *shuffleByKey* is $\{(k_3, \{k_2, v_2\})\}$. Function *sortByKey* takes a function f_{SORT} and a set of key-value pairs, and sorts the set into a list based on the relation computed by f_{SORT}.

2.2 List Homomorphism and Homomorphism Theorems

List homomorphisms are a class of recursive functions on lists, which match very well the divide-and-conquer paradigm [2, 4, 5, 12, 15, 16]. They are attractive in parallel programming, not only because they are suitable for parallel implementation, but also because they enjoy many nice algebraic properties, among which, the three well-known homomorphism theorems form the basis of *systematic development of parallel programs* [4, 6, 7, 9, 10]. Recently, it has been shown [8, 14] that homomorphisms can be automatically developed for solving various kinds of problems. All these have indeed motivated us to see how to apply these results to parallel programming with MapReduce.

Definition 1 (List homomorphism). *Function h is said to be a list homomorphism, if and only if there is a function f and an associative operator $\odot$ such that the function h is defined as follows.*

$$\begin{aligned}
h\ [a] &= f\ a \\
h\ (x +\!\!+\, y) &= h\ x \odot h\ y
\end{aligned}$$

Since h is uniquely determined by f and $\odot$, we write $h = (\![f, \odot]\!)$.

For instance, the function that sums up the elements in a list can be described as a list homomorphism $(\![\mathit{id}, +]\!)$:

$$\begin{aligned}
\mathit{sum}\ [a] &= a \\
\mathit{sum}\ (x +\!\!+\, y) &= \mathit{sum}\ x + \mathit{sum}\ y.
\end{aligned}$$

Below is the well-known theorem for homomorphisms [6]. It provides a necessary and sufficient condition for the existence of a list homomorphism.

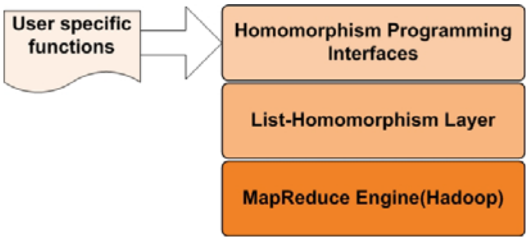

Fig. 2. System Overview

Theorem 1 (The third homomorphism theorem). *Let h be a given function and $\ominus$ and $\oslash$ be binary operators. If and only if the following two equations hold for any element a and list y*

$$\begin{aligned} h\ ([a] +\!\!+\ y) &= a \ominus h\ y \\ h\ (y +\!\!+\ [a]) &= h\ y \oslash a \end{aligned}$$

then the function h is a homomorphism.

In order to show how to automatically derive a list homomorphism, firstly, we introduce the concept of function's right inverse.

Definition 2 (Right inverse). *For a function h, its right inverse h° is a function that satisfies $h \circ h^\circ \circ h = h$.*

By taking use of right inverse, we can obtain the list-homomorphic definition as follows.

$$h = (\!| f, \odot |\!) \ \ \textbf{where}\ f\ a = h\ [a] \\ \qquad\qquad\qquad\qquad l \odot r = h\ (h^\circ\ l +\!\!+\ h^\circ\ r)$$

With this property, the third homomorphism theorem is also an important and useful theorem for automatic derivation of list homomorphisms [14]. Our parallelization algorithm is mainly based on the third homomorphism theorem.

3 A Homomorphism-Based Framework for Parallel Programming with MapReduce

The main contribution of our work is a novel programming model and its framework for systematic programming over MapReduce, based on theorems of list homomorphisms [8, 14]. Our framework *Screwdriver* is built on top of Hadoop, purely in Java.

As shown in Fig. 2, *Screwdriver* consists of three layers: the interface layer for easy parallel programming, the homomorphism layer for implementing homomorphism, and the base layer of the MapReduce engine (Hadoop).

Listing 1.1. Programming Interface

```
public abstract class ThirdHomomorphismTheorem<T1,T2> {
...
    public abstract T2 fold(ArrayList<T1> values);
    public abstract ArrayList<T1> unfold(T2 value);
...
}
```

3.1 Programming Interface and Homomorphism Derivation

The first layer of Screwdriver provides a simple programming interface and generates a homomorphism based on the third homomorphism theorem.

Users specify a pair of sequential functions instead of specifying a homomorphism directly: one for solving the problem itself and the other for a right inverse of the problem. Consider the summing-up example again. A right inverse sum° of the function *sum* takes a value (the result of *sum*) and yields a singleton list whose element is the input value itself. The functional definition of sum° is: $\mathit{sum}^\circ\ s = [s]$.

Listing 1.1 shows the programming interface provided in Screwdriver, where users should write a program by inheriting the `ThirdHomomorphismTheorem` class. The function `fold` corresponds to the sequential function that solves the problem, and the function `unfold` corresponds to the sequential function that computes a right inverse. In a functional specification, the types of the two functions are $\mathit{fold} :: [t_1] \to t_2$ and $\mathit{unfold} :: t_2 \to [t_1]$. The concrete Java source code with Screwdriver for the summing-up example can be found on our project's site.

To utilize the third homomorphism theorem, users are requested to confirm that the two functions satisfy the following conditions. Firstly, the function *unfold* should be a right inverse of the function *fold*. In other words, the equation $\mathit{fold} \circ \mathit{unfold} \circ \mathit{fold} = \mathit{fold}$ should hold. Secondly, for the *fold* function there should exist two operators $\ominus$ and $\oplus$ as stated in Theorem 1. A sufficient condition for this second requirement is that the following two equations hold respectively for any a and x.

$$\mathit{fold}([a] +\!\!+\, x) = \mathit{fold}([a] +\!\!+\, \mathit{unfold}(\mathit{fold}(x))) \tag{1}$$

$$\mathit{fold}(x +\!\!+\, [a]) = \mathit{fold}(\mathit{unfold}(\mathit{fold}\ x) +\!\!+\, [a]) \tag{2}$$

Note that we can use some tools (such as QuickCheck [3]) in practice to verify whether Equations (1) and (2) hold or not.

Under these conditions, Screwdriver automatically derives a list homomorphism from the pair of *fold* and *unfold* functions. A list homomorphism $([f, \oplus])$ that computes *fold* can be obtained by composing user's input programs, where the parameter functions f and $\oplus$ are defined as follows.

$$\begin{aligned} f\ a &= \mathit{fold}([a]) \\ x \oplus y &= \mathit{fold}(\mathit{unfold}\ x +\!\!+\, \mathit{unfold}\ y). \end{aligned}$$

3.2 Homomorphism Implementation on MapReduce

In the second layer, Screwdriver provides an efficient implementation of list homomorphisms over MapReduce. In particular, the implementation consists of two passes of MapReduce.

Manipulation of Ordered Data

The computation of a list homomorphism obeys the order of elements in the input list, while the input data of MapReduce is given as a set stored on the distributed file system. This means we need to represent a list as a set.

On Screwdriver, we represent each element of a list as an $(index, value)$ pair where *index* is an integer indicating the position of the element. For example, a list $[a, b, c, d, e]$ may be represented as a set $\{(3, d), (1, b), (2, c), (0, a), (4, e)\}$. Note that the list can be restored from this set representation by sorting the elements in terms of their indices. Such indexed pairs permit storing data in arbitrary order on the distributed file systems

Implementing Homomorphism by two Passes of MapReduce

For the input data stored as a set on the distributed file system, Screwdriver computes a list homomorphism in parallel by two passes of MapReduce computation. Here, the key idea of the implementation is that we group the elements consecutive in the list into some number of sublists and then apply the list homomorphism in parallel to those sublists.

In the following, we summarize our two-pass implementation of homomorphism $([f, \oplus])$. Here, *hom f* $(\oplus)$ denotes a sequential version of $([f, \oplus])$, *comp* is a comparing function defined over the *Int* type, and *const* is a constant value defined by the framework.

```
$hom_{\mathbf{MR}} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \{(Int, \alpha)\} \to \beta$
$hom_{\mathbf{MR}}\ f\ (\oplus) = getValue \circ MapReduce\ ([\cdot])\ g_{\text{SHUFFLE}}\ comp\ g_{\text{REDUCE}}$
                                   $\circ\ MapReduce\ ([\cdot])\ f_{\text{SHUFFLE}}\ comp\ f_{\text{REDUCE}}$
  where
        $f_{\text{SHUFFLE}} :: Int \to Int$
        $f_{\text{SHUFFLE}}\ k = k / const$

        $f_{\text{REDUCE}} :: (Int, [\alpha]) \to (Int, \beta)$
        $f_{\text{REDUCE}}\ (k, as) = (k, hom\ f\ (\oplus)\ as)$

        $g_{\text{SHUFFLE}} :: Int \to Int$
        $g_{\text{SHUFFLE}}\ k = 1$

        $g_{\text{REDUCE}} :: (Int, [\beta]) \to (Int, \beta)$
        $g_{\text{REDUCE}}\ (1, bs) = (1, hom\ id\ (\oplus)\ bs)$

        $getValue :: \{(Int, \beta)\} \to \beta$
        $getValue\ \{(1, b)\} = b$
```

First pass of MapReduce: The first pass of MapReduce divides the list into some sublists, and computes the result of the homomorphism for each sublist. Firstly in the MAP phase, we do no computation (except for wrapping the key-value pair into a singleton list). Then in the SHUFFLE&SORT phase, we group the pairs so that the set-represented list is partitioned into some number of sublists and sort each grouped elements by their indices. Finally, we apply the homomorphism to each sublist in the REDUCE phase.

Second pass of MapReduce: The second pass of MapReduce computes the result of the whole list from the results of sublists given by the first pass of MapReduce. Firstly in the MAP phase, we do no computation as in the first pass. Then in the SHUFFLE&SORT phase, we collect the subresults into a single set and sort them by the their indices. Finally, we reduce the subresults using the associative operator of the homomorphism.

Finally, by the *getValue* function, we picked the result of the homomorphism out from the set (of single value).

Implementation Issues

In terms of the parallelism, the number of the MAP tasks in the first pass is decided by the data splitting mechanism of Hadoop. For one split data of the input, Hadoop spawns one MAP task which applies f_{MAP} to each record. The number of the REDUCE tasks in the first pass of MapReduce should be chosen properly with respect to the total number of the task-trackers inside the cluster. By this number of REDUCE task, the parameter *const* in the program above is decided. In the REDUCE phase in the second pass of MapReduce, only one REDUCE task is invoked because all the subresults are grouped into a single set.

4 A Programming Example

In this section we demonstrate how to develop parallel programs with our framework, by using the maximum prefix sum problem in the introduction as our example. As discussed in Section 3, users need to define a Java class that inherits the Java class shown in Listing 1.1 and implement the two abstract methods *fold* and *unfold*.

Recall the maximum prefix sum problem in the introduction. It is not difficult to develop a sequential program for computing the maximum prefix sum:

$$mps\ [1, -2, 3, ...] = 0 \uparrow 1 \uparrow (1 + (-2)) \uparrow (1 + (-2) + 3) \uparrow (1 + (-2) + 3 + ...)$$

where $a \uparrow b$ returns a if $a > b$ otherwise returns b.

Although the *mps* function cannot be represented by a homomorphism in the sense that it cannot be described at the same time, it is not difficult to see, as discussed in [14], that the tupled function $mps \vartriangle sum$ can be described leftwards and rightwards.

Listing 1.2. Our Parallel Program for Solving MPS Problem

```
import ...

public class Mps extends ThirdHomomorphismTheorem<LongWritable, LongPair> {

    // Computing mps and sum at a time.
    public LongPair fold(ArrayList<LongWritable> values) {
        long mps = 0;
        long sum = 0;
        for (LongWritable v : values) {
            sum += v.get();
            if (sum > mps) mps = sum;
        }

        return new LongPair(mps, sum);
    }

    // A right inverse of fold.
    public ArrayList<LongWritable> unfold(LongPair value) {
        long m = value.getFirst();
        long s = value.getSecond();

        ArrayList<LongWritable> rst = new ArrayList<LongWritable>();
        rst.add(new LongWritable(m));
        rst.add(new LongWritable(s-m));
        return rst;
    }
}
```

What we need to do now is to develop an efficient sequential program for computing $mps \vartriangle sum$ and an efficient sequential program for computing a right inverse of $mps \vartriangle sum$. These two sequential programs are not difficult to obtain. A simple sequential program for computing the tupled function $(mps \vartriangle sum)$ can be defined by

$$\begin{aligned}
(mps \vartriangle sum)\ [a] &= (a \uparrow 0, a) \\
(mps \vartriangle sum)\ (x +\!\!+ [a]) &= \mathbf{let}\ (m, s) = (mps \vartriangle sum)\ x\ \mathbf{in}\ (m \uparrow (s + a), s + a)
\end{aligned}$$

and a right inverse of $(mps \vartriangle sum)$ can be defined as follows.

$$(mps \vartriangle sum)^{\circ}\ (m, s) = [m, s - m].$$

That is all for our development. We can now use $(mps \vartriangle sum)$ as the *fold* function and $(mps \vartriangle sum)^{\circ}$ as *unfold* function. Listing 1.2 gives the concrete Java program for solving the maximum prefix sum problem using Screwdriver.

5 Experiments

In this section, we report experiment results that evaluate the performance of our framework on PC clusters. We evaluated the scalability of programs on our framework, the overhead of our framework compared with the direct Hadoop program, and the overhead of the non-trivial parallel program compared with sequential program.

We configured clusters with 2, 4, 8, 16, and 32 virtual machines (VM) inside the *EdubaseCloud* system in National Institute of Informatics. Each VM has one CPU (Xeon E5530@2.4GHz, 1 core), 3 GB memory, and 5 GB disk space. We installed Hadoop (version 0.20.2.203) on each VM. Three sets of programs are used for the evaluation: *SUM* computes the sum of 64-bit integers; *VAR* computes the variance of 32-bit floating-point numbers; *MPS* solves the maximum-prefix-sum problem for a list of 64bit-integers. We both implemented the programs with the Hadoop APIs directly (*SUM-MR*, *VAR-MR*, *MPS-MR*), and with our Screwdriver (*SUM-LH*, *VAR-LH*, *MPS-LH*). Also a sequential program is implemented (*MPS-Seq*). The input for SUM and MPS was a list of 10^8 64bit-integer elements (593 MB), and the input for VAR is a list of 10^8 32bit-floating-point numbers (800 MB). Note that the elements of lists are indexed as in Section 3.2 with the type information (each element has a 64bit-integer index), stored in the Avro data format, and put in the HDFS.

The experiment results are summarized in Fig. 3 and Table 1. Note that the relative speedup is calculated with respect to the result of 2 nodes. The execution of the parallel programs on our framework and on Hadoop failed on 1 node, due to the limitation of disk space for the intermediate data.

All the programs achieved good scalability with respect to the number of nodes: the speedup ratios for 32 nodes against 2 nodes are more than 10 times. This shows that our framework does not spoil the strong advantage of MapReduce framework, namely *scalable data processing*. For the summation problem, the SUM-LH program on our framework cannot use combiner due to the limitation of Hadoop's implementation, so SUM-MR which uses combiner doing local reductionism can run almost twice faster. for almost all MapReduce programs combiners usually can increase performance very much. So we will work on to let our framework taking full use of data-locality. And we think it will bring notable performance improvement. Besides this, two-passes MapReduce processing and sorting with respect to the keys, which are unnecessary for the summation problem. In other words, with these overheads we can extend the MapReduce framework to support computations on *ordered* lists.

Finally we discuss the execution times for the maximum-prefix-sum problem. Although the parallel program on our framework MPS-LH shows good scalability (as well as MPS-MR), it ran slower on 32 nodes than the sequential program MPS-Seq. We consider this is a reasonable result: first, in this case of the maximum-prefix-sum problem, the parallel algorithm becomes more complex than that of sequential one, and in particular we produced (large) intermediate data when doing parallel processing on our framework but it is not the case for sequential processing. Second, the test data is not big enough, so the sequential program can still handle it. Because the limitation of our cloud, we cannot test big enough data. An important work to improve the performance in future is to make use of data locality to optimize the parallel execution.

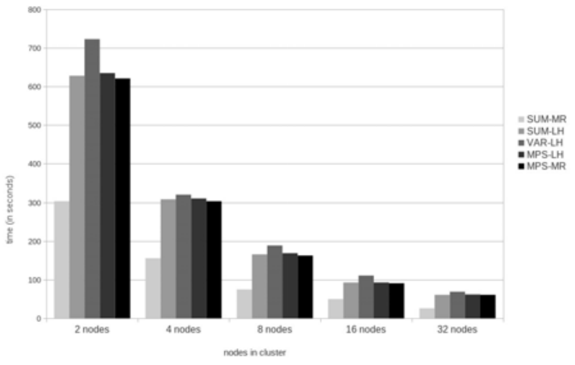

Fig. 3. Time Consuming

Table 1. Execution Time (second) and Relative Speedup w.r.t. 2 Nodes

Program	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
SUM-MR	NA (NA)	304 (1.00)	156 (1.95)	75 (4.05)	50 (6.08)	26 (11.69)
SUM-LH	NA (NA)	628 (1.00)	309 (2.03)	166 (3.78)	93 (6.75)	61 (10.30)
VAR-LH	NA (NA)	723 (1.00)	321 (2.25)	189 (3.82)	111 (6.50)	69 (10.45)
MPS-LH	NA (NA)	635 (1.00)	311 (2.04)	169 (3.76)	93 (6.78)	62 (10.24)
MPS-MR	NA (NA)	621 (1.00)	304 (2.04)	163 (3.81)	91 (6.82)	61 (10.22)
MPS-Seq	37	NA (NA)	NA (NA)	NA (NA)	NA (NA)	NA (NA)

6 Concluding Remarks

The research on list homomorphisms and the third homomorphism theorem indicate a systematic and constructive way to parallelization, by which this work was inspired. List homomorphisms and the theory related to them are very suitable for providing a method of developing MapReduce programs systematically.

In this paper, we presented a new approach to systematic parallel programming over MapReduce based on program calculation, and gave a concrete implementation to verify the approach. We believe that the calculation theorems for list homomorphisms can provide a higher abstraction over MapReduce, and such abstraction brings good algebraic properties of list homomorphisms to parallel programming with MapReduce.

We introduced a novel parallel programming approach based on list homomorphism to wrapping MapReduce. And we believe such approach is not limited to wrapping MapReduce, but also can be adopted to other parallel programming environments to provide higher-level programming interfaces.

As a future work, we plan to extend this framework for resolving parallel programming problems on trees and graphs. It will enlarge the computation capability of Screwdriver.

References

1. Bird, R.S.: Introduction to Functional Programming using Haskell. Prentice-Hall, Englewood Cliffs (1998)
2. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. NATO ASI Series F, vol. 36, pp. 5–42. Springer, Heidelberg (1987)
3. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) ICFP 2000: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pp. 268–279. ACM Press, New York (2000)
4. Cole, M.: Parallel programming with list homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: 6th Symposium on Operating System Design and Implementation (OSDI 2004), pp. 137–150 (2004)
6. Gibbons, J.: The third homomorphism theorem. Journal of Functional Programming 6(4), 657–665 (1996)
7. Gorlatch, S.: Systematic extraction and implementation of divide-and-conquer parallelism. In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 274–288. Springer, Heidelberg (1996)
8. Hu, Z.: Calculational parallel programming. In: HLPP 2010: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications, p. 1. ACM Press, New York (2010)
9. Hu, Z., Iwasaki, H., Takeichi, M.: Formal derivation of efficient parallel programs by construction of list homomorphisms. ACM Transactions on Programming Languages and Systems 19(3), 444–461 (1997)
10. Hu, Z., Takeichi, M., Chin, W.N.: Parallelization in calculational forms. In: POPL 1998, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 316–328. ACM Press, New York (1998)
11. Lämmel, R.: Google's MapReduce programming model — Revisited. Science of Computer Programming 70(1), 1–30 (2008)
12. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: InfoScale 2006: Proceedings of the 1st International Conference on Scalable Information Systems. ACM International Conference Proceeding Series, vol. 152. ACM Press, New York (2006)
13. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In: Shao, Z., Pierce, B.C. (eds.) POPL 2009: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 177–185. ACM Press, New York (2009)
14. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), pp. 146–155. ACM Press, New York (2007)
15. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2002)
16. Steele Jr. G.L.: Parallel programming and parallel abstractions in fortress. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 1–1. Springer, Heidelberg (2006)