Implementing Generate-Test-and-Aggregate Algorithms on Hadoop

Yu Liu[1], Sebastian Fischer[2], Kento Emoto[3], Zhenjiang Hu[4]

[1] The Graduate University for Advanced Studies,Tokyo, Japan
yuliu@nii.ac.jp

[3] University of Tokyo, Japan
emoto@mist.i.u-tokyo.ac.jp

[2],[4] National Institute of Informatics,Tokyo, Japan
{fischer,hu}@nii.ac.jp

Abstract MapReduce is a useful and popular programming model for large-scale parallel processing. However, for many complex problems, it is usually not easy to develop the efficient parallel programs that match MapReduce paradigm well. To remedy this situation, the generate-test-and-aggregate algorithms have been proposed to simplify parallel programming for a large class of problems, where efficient parallel programs can be derived from users' naive but correct programs in terms of a generator, a test and an aggregate. The obtained efficient-parallel algorithms are in the form that fit for implementation with MapReduce.

In this paper we show that the generate-test-and aggregate algorithms, together with the optimization rules, can be efficiently implemented on Hadoop as a Java library. With this library, one can just concentrate on specifying naive generate-test-and aggregate algorithms in Java, and obtain efficient MapReduce programs for free. We demonstrate our library with some generate-test-and-aggregate examples, showing that they can be easily and efficiently implemented using our library over MapReduce.

1 Introduction

Parallel programming is notoriously difficult due to the complexity of concurrency, performance problems and diversity of parallel machine models. Skeletal parallel programming [2, 13] is an important and useful methodology for developing parallel programs, by which programmers can use a few basic structures of parallel algorithm as building blocks to express complex patterns of computation.

Google's MapReduce [4], the de facto standard for large scale data- intensive applications, can be seen as a skeletal parallel programming model with two skeletons: map and reduce. By making use of the skeletal parallelism, users can program their applications with ready-made skeletons without considering low level details of parallelization, data distribution, load balancing and fault tolerance.

Despite the popularity of MapReduce, developing efficient MapReduce parallel programs is not always easy in practice. As an example, consider the known statistics problem of inferring a sequence of hidden states of a probabilistic model that most likely causes a sequence of observations [9]. Suppose that we have a sequence of observed events

$$(x_1, ..., x_n)$$

and a set of hidden states S that cause events and transition into other states with certain probabilities. We want to calculate a most likely sequence of states $z_i \in S$

$$(z_0, z_1, ..., z_n)$$

that is, one that maximizes

$$\prod_{i=1}^{n} P(x_i \mid z_i) P(z_i \mid z_{i-1})$$

where $P(a||b)$ denotes the conditional probability of a, given b. This problem is important in natural language processing and error code correction, but it is far from easy for one to come up with an efficient MapReduce program to solve it. The problem becomes more difficult, if we would like to find the most likely sequence with additional requirements such that the sequence should contain a specific state exactly five times, or that the sequence should not contain a specific state anywhere after another. The main difficulty in programming with MapReduce is that nontrivial problems are usually not in a simple divide-and-conquer form that can be easily mapped to MapReduce without producing an exponential number of intermediate candidates.

To remedy this situation, a calculational theory has been proposed [5] to synthesize efficient MapReduce parallel program for a general class of problems which can be specified in the following generate-test-and- aggregate form:

$$\mathit{aggregate} \circ \mathit{test} \circ \mathit{generate}.$$

Problems that match this specification can be solved by first generating possible solution candidates, then keeping those candidates that pass a test of a certain condition, and finally selecting a valid solution or making a summary of valid solutions with an aggregating computation. For example, the above statistics problem may be informally specified by generating all possible sequences of state transitions, keeping those that satisfy a certain condition, and selecting one that maximizes the products of probabilities.

In this paper, show that the calculation theory for the generate-test-and aggregate algorithms can be efficiently implemented on Hadoop as a Java library. With this library, one can just concentrate on specifying naive generate-test-and aggregate algorithms in Java, and obtain efficient MapReduce programs for free. Our main technical contributions are two folds. First, we design a Generic Java Interface that makes it easy for users to specify various generate-test-and-aggregate algorithms, and show how the two important calculation theorems [5], the *semiring fusion theorem* and the *filter embedding theorem*, can be implemented in Java so that efficient MapReduce programs can be automatically generated and run on Hadoop. Second, we demonstrate the usefulness of the Java library with two interesting examples, the *Knapsack* problem and the *Viterbi* algorithm, showing that they can be easily and efficiently implemented using our library on Hadoop. All the source codes are available online[1].

The rest of the paper is organized as follows. After explaining the background of MapReduce,

[1] http://screwdriver.googlecode.com

homomorphisms, and the calculational framework for generate-test-and-aggregate algorithms in Section 2, we show how we design and implementation of Generate-Test-and-Aggregate Library in Section 3. Then, we demonstrate the usefulness of our Java library using the Knapsack and the Viterbi problems, and report our experimental results in Section 4. Finally, we conclude the paper and highlight some future work in Section 5.

2 Background

In this section we introduce the concepts of list homomorphisms, MapReduce, and generate-test-and-aggregate algorithms. The notation we use to formally describe algorithms is based on the functional programming language Haskell [1]. Function application can be written without parentheses, i.e., $f\ a$ equals $f(a)$. Functions are curried and application associates to the left, thus, $f\ a\ b$ equals $(f\ a)\ b$. Function application has higher precedence than 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.

In addition to formal descriptions, we provide Java code to describe the concrete implementation of introduced computation patterns in Section 3.

2.1 List Homomorphism

List homomorphisms belong to a special class of recursive functions on lists.

Definition 1 (List Homomorphism) *Function h is said to be a list homomorphism, if and only if there is a function f, an associative operator $\odot$, and an identity element $id_\odot$ of $\odot$, such that the function h satisfies the following equations:*

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

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:

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ [a] &= a \\ sum\ (x +\!\!+ y) &= sum\ x + sum\ y. \end{aligned}$$

List homomorphisms have a close relationship with parallel computing and they have been studied intensively [2, 8, 10]. The following is a well-known lemma for list homomorphisms [7].

Lemma 1 (First Homomorphism Lemma) *Any list homomorphism can be written as the composition of a map and a reduce:*

$$(\![f, \odot]\!) = reduce\ (\odot) \circ map\ f.$$

where informally,

$$\begin{aligned} &reduce\ (\odot)\ [x_1, x_2, ..., x_n] \\ &\qquad = x_1 \odot x_2 \odot ... \odot x_n \\ &map\ f\ [x_1, x_2, ..., x_n] \\ &\qquad = [f\ x_1, f\ x_2, ..., f\ x_n] \end{aligned}$$

This lemma says that any list homomorphism can be decomposed using the map and reduce functions, which are both suitable for efficient parallel implementation.

2.2 MapReduce

Google's MapReduce [3] is a simple programming model, with associated efficient implementations like Hadoop, for processing very large data sets in a massively parallel manner. It is inspired by, but not the same as, using the map and reduce functions like in the First Homomorphism Theorem.

Figure 1 shows the data flow of MapReduce. To deal with a very large amount of data, it is split into blocks and distributed to different nodes in a cluster. In the MapReduce framework, the input and output data is managed as a set of key-value pairs. Computation of MapReduce mainly consists of three phases: a MAP phase, a SHUFFLE & SORT phase, and a REDUCE phase. In the MAP phase, each input key-value pair is processed independently and a set of key-value pairs will be produced. Then in the SHUFFLE & SORT phase, the key-value pairs are sorted and grouped based on the key. Finally, in the REDUCE phase, the key-value

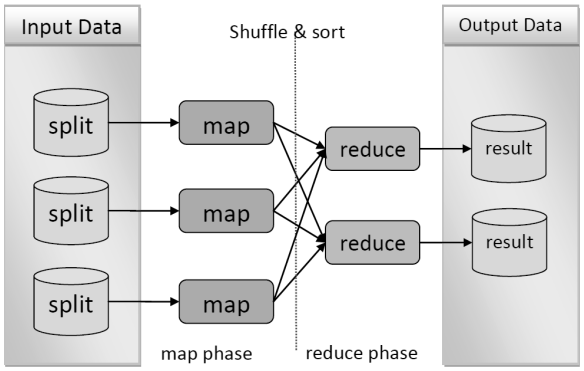

Figure 1. MapReduce data flow

pairs with the same key are processed to generate a combined result.

As data is represented as *sets* of key value pairs in the MapReduce model, the order of elements in the input of a list homomorphism needs to be accounted for explicitly. How to implement list homomorphisms on MapReduce is described in [11] and there is an implementation of this idea called Screwdriver.

2.3 Generate, Test, and Aggregate Algorithms

Recently, the generate-test-and-aggregate framework [5] has been proposed to specify MapReduce programs as search problems based on three components:

$$aggregate \circ test \circ generate$$

A generate-test-and-aggregate algorithm consists of

- a generator that can generate a bag (or multiset) of intermediate lists,
- a test that filters the intermediate lists, and
- an aggregator that computes a summary of valid intermediate lists.

A specification in this form can be transformed into a single list homomorphism if

- the generator is defined as a list homomorphism and parameterized by an arbitrary semiring,
- the test is defined as a list homomorphism, and
- the aggregator is defined as a semiring homomorphism from the semiring of bags of lists into an arbitrary semiring.

The following definitions and theorems, copied from [5], clarify what this means.

Definition 2 (Monoid) *Given a set M and a binary operator $\odot$, the pair $(M, \odot)$ is called a monoid, if $\odot$ is associative and has an identity element $id_\odot \in M$.*

$$\begin{aligned}(a \odot b) \odot c &= a \odot (b \odot c)\\ id_\odot \odot a &= a \odot id_\odot = a\end{aligned}$$

Definition 3 (Monoid homomorphism) *Given two monoids $(M, \odot)$ and $(M', \odot')$, a function*

$$hom : M \to M'$$

is called a monoid homomorphism from $(M, \odot)$ to $(M', \odot')$ if and only if

$$\begin{aligned}hom\ id_\odot &= id_{\odot'}\\ hom(x \odot y) &= hom\ x \odot' hom\ y\end{aligned}$$

For example function *sublists* that computes the bag of all sublists of a given input list, is a monoid homomorphism

$$\begin{aligned}sublists[\,] &= \wr[\,]\int\\ sublists[x] &= \wr[\,]\int \uplus \wr[x]\int\\ sublists(xs +\!\!+ ys) &= sublists\ xs \times_{+\!\!+} sublists\ ys\end{aligned}$$

Here $\wr[x]\int$ denotes a bag of lists, $\uplus$ denotes bag union, and $\times_{+\!\!+}$ is the lifting of list concatenation to bags. For example:

$$\begin{aligned}& sublists[1,2,3] =\\ =\ & sublists([1] +\!\!+ [2] +\!\!+ [3])\\ =\ & sublists[1] \times_{+\!\!+} sublists[2] \times_{+\!\!+} sublists[3]\\ =\ & (\wr[\,]\int \uplus \wr[1]\int) \times_{+\!\!+} (\wr[\,]\int \uplus \wr[2]\int)\\ & \quad \times_{+\!\!+} (\wr[\,]\int \uplus \wr[3]\int)\\ =\ & (\wr[\,], [1]\int) \times_{+\!\!+} (\wr[\,], [2]\int) \times_{+\!\!+} (\wr[\,], [3]\int)\\ =\ & (\wr[\,] +\!\!+ [\,], [\,] +\!\!+ [2], [1] +\!\!+ [\,], [1] +\!\!+ [2]\int)\\ & \quad \times_{+\!\!+} (\wr[\,], [3]\int)\\ =\ & (\wr[\,], [1], [1,2], [2]\int) \times_{+\!\!+} (\wr[\,], [3]\int)\\ =\ & (\wr[\,], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3]\int)\end{aligned}$$

Note that we can reorder bag elements (here lexicographically) because bags abstract from the order of their elements.

To implement generate-test-and-aggregate based problems efficiently it is crucial to generalize a generator function like *sublists* as follows:

$$\begin{aligned} &sublists_{\oplus,\otimes}\ f\ [\,] = id_\otimes \\ &sublists_{\oplus,\otimes}\ f\ [x] = id_\otimes \oplus f\ x \\ &sublists_{\oplus,\otimes}\ f\ (xs +\!\!+ ys) \\ &\qquad = sublists\ xs \otimes sublists\ ys \end{aligned}$$

This version is parameterized by operations $\oplus$ and $\otimes$, their units, and a function f. We get the original definition by instantiating it as follows

$$sublists = sublists_{\uplus,\times_{+\!\!+}}\ (\lambda x \to \wr[x]\int)$$

where the lambda abstraction denotes an anonymous function that creates a singleton bag with a singleton list.

The result type of such a generalized generator is determined by the result types of the passed operators. We will use operators with specific properties as discussed in the following definition and call such generators *polymorphic semiring generators*.

Definition 4 (Semiring) *Given a set S and two binary operations $\oplus$ and $\otimes$, the triple $(S, \oplus, \otimes)$ is called a semiring if and only if*

- *$(S, \oplus)$ is a commutative monoid with identity element $id_\oplus$*
- *$(S, \otimes)$ is a monoid with identity element $id_\otimes$*
- *$\otimes$ is associative and distributes over $\oplus$*
- *$id_\oplus$ is a zero of $\otimes$*

Definition 5 (Semiring homomorphism) *Given two semirings $(S, \oplus, \otimes)$ and $(S', \oplus', \otimes')$, a function $hom : S \to S'$ is a semiring homomorphism from $(S, \oplus, \otimes)$ to $(S', \oplus', \otimes')$, if and only if it is a monoid homomorphism from $(S, \oplus)$ to $(S', \oplus')$ and also a monoid homomorphism from $(S, \otimes)$ to $(S', \otimes')$.*

If we compose a polymorphic semiring generator that produces a bag of lists with a semiring homomorphism that consumes this bag of lists, we can fuse this composition as explained by the following theorem.

Theorem 1 (Semiring Fusion) *Given a set A, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism aggregate from $(\wr[A]\int, \uplus, \times_{+\!\!+})$ to $(S, \oplus, \otimes)$, and a polymorphic semiring generator generate, the following equation holds:*

$$\begin{aligned} &aggregate \circ generate_{\uplus,x_{+\!\!+}}\ (\lambda x \to \wr[x]\int) \\ &\quad = generate_{\oplus,\otimes}(\lambda x \to aggregate \wr[x]\int) \end{aligned}$$

The resulting definition does not generate an intermediate bag of lists and is, therefore, more efficient that the original composition.

The next theorem additionally incorporates a test that is defined using a filter function for bags and a list homomorphism.

Theorem 2 (Filter Embedding) *Given a set A, a finite monoid $(M, \odot)$, a monoid homomorphism hom from $([A], +\!\!+)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism aggregate from $(\wr[A]\int, \times_{+\!\!+} \uplus)$ to $(S, \oplus, \otimes)$, a function $ok : M \to Bool$ and a polymorphic semiring generator generate, the following equation holds.*

$$\begin{aligned} &aggregate \circ filter(ok \circ hom) \\ &\qquad \circ generate_{\uplus,x_{+\!\!+}}\ (\lambda x \to \wr[x]\int) \\ &= postprocess_M\ ok \\ &\quad \circ generate_{\oplus_M,\otimes_M}(\lambda x \to aggregate_M \wr[x]\int) \end{aligned}$$

Here

$$postprocess_M\ ok\ f \quad = \bigoplus_{\substack{m \in M \\ ok\ m = True}} f_m$$

$$(aggregate_M\ \wr[x]\int)_m \quad = \begin{cases} aggregate\ \wr[x]\int & \text{if } m = hom[x] \\ id_\oplus & \text{otherwise} \end{cases}$$

The details of the definitions of the postprocessing function and the modified aggregator are not in the scope of this paper but the important observation here is the following: the composition of a generator, a test, and an aggregator as in Theorem 2 can be expressed as composition of the generator parameterized with a specific semiring and a postprocessing function. Again, it is crucial to use a polymorphic semiring generator with a semiring different from the semiring of bags of lists.

If the generator is a list homomorphism, then the resulting program can be executed on Hadoop using Screwdriver.

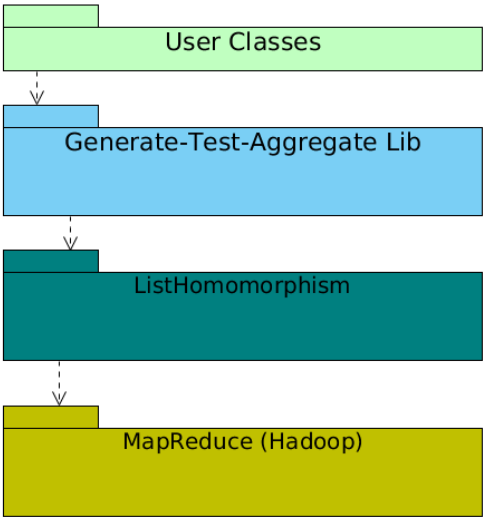

Figure 2. System Overview

Listing 1. Aggregator.java

```
1 public interface Aggregator<Single,Val> {
2     public Val zero();
3     public Val one();
4     public Val singleton(Single x);
5     public Val plus(Val left, Val right);
6     public Val times(Val left, Val right);
7 }
```

3 Implementing Generate, Test, and Aggregate Algorithms

In this section we present a Java library that implements generate-test-and-aggregate algorithms as list homomorphisms. Figure 2 shows an overview of our implementation. The Generate-Test-and-Aggregate Library allows users to implement algorithms following the generate-test-and-aggregate based pattern and implements them as list homomorphisms on top of Screwdriver which itself uses Hadoop under the hood.

3.1 Semiring Homomorphisms

The aggregator in a generate-test-and-aggregate algorithm must be expressed as a semiring homomorphism from the bag of lists semiring into an arbitrary semiring. We provide an interface *Aggregator* shown in Listing 1 to represent such homomorphisms. In implementations of this interface $(Val, plus, times)$ must be a semiring and *zero* and *one* must be the identity elements of *plus* and *times* respectively. Then the implementation represents the semiring homomorphism from bags of lists with elements of type *Single* that maps a singleton bag with a singleton list according to the *singleton* method.

Users can implement their aggragators by providing an implementation of this interface. We show examples in Section 4.

3.2 List Homomorphisms

List homomorphisms are used in two places when defining generate-test-and-aggregate algorithms: for the generator and for the test. In both cases the list homomorphisms are combined with a postprocessing function so we abstract this structure as interface *MapReducer* shown in Listing 2. In implementations of this interface, $(Val, combine)$ must be a monoid and *identity* must return the identity element of *combine* such that the implementation represents the function: $postprocess \circ (\![element, combine]\!)$. All type parameters of this interface must implement the *Writable* interface provided by Hadoop such that corresponding data can be distributed in a cloud. We can pass an arbitrary implementation of the *MapReducer* interface to Screwdriver to execute it on Hadoop.

The *Test* interface shown in Listing 3 is a specialization of *MapReducer* restricting the result type of the postprocessing function. Users can implement their tests as composition of a list homomorphism with a postprocessing function by providing an implementation of this interface.

In order to define a polymorphic semiring generator as a list homomorphism users need to implement a subclass of the abstract class *Generator* shown in Listing 4. This class implements the three methods corresponding to the definition of a monoid homomorphism based on three abstract methods that take an additional aggregator as argument. Users need to implement these parameterized methods as well as a postprocessing function when defining a subclass of *Generator*. We give an example in Section 4.

The three abstract methods are more restrictive than the corresponding methods without an aggregator as parameter because users cannot choose the result type. Instead, they can only use the passed aggregator to construct a result which is crucial to implement Theorem 2. When implementing a generator, users can think of the passed aggregator to construct

Listing 2. MapReducer.java interface

```
public interface MapReducer
    <Elem extends Writable, Val extends Writable, Res extends Writable> {

    public Val identity();
    public Val element(Elem elem);
    public Val combine(Val left, Val right);
    public Res postprocess(Val val);
}
```

Listing 3. Test.java

```
public interface Test<Elem extends Writable,Key extends Writable>
    extends MapReducer<Elem,Key,BooleanWritable> {}
```

Listing 4. Generator.java

```
public abstract class Generator
    <Elem   extends Writable,
     Single extends Writable,
     Val    extends Writable,
     Res    extends Writable>
    implements MapReducer<Elem,Val,Res> {

    protected final Aggregator<Single,Val> aggregator;

    public Generator(Aggregator<Single,Val> aggregator) {
           this.aggregator = aggregator;
    }

    public abstract <V> V identity(Aggregator<Single,V> aggregator);
    public abstract <V> V element(Aggregator<Single,V> aggregator, Elem e);
    public abstract <V> V combine(Aggregator<Single,V> aggregator, V l, V r);

    public Val identity() {
        return identity(aggregator);
    }

    public Val element(Elem elem) {
        return element(aggregator, elem);
    }

    public Val combine(Val left, Val right) {
        return combine(aggregator, left, right);
    }

    public <Key> Generator<Elem,Single,WritableMap<Key,Val>,Res>
        embed(final Test<Single,Key> test) {
            // implementation omitted
    }
}
```

a bag of lists but in the implementation no intermediate bags will be created.

The method *embed* of the *Generator* class implements the filter embedding. We omit it's implementation because we do not describe the semiring $WritableMap < Key, Val >$ used internally. The method *embed* returns a new instance of *Generator* that is constructed using a new instance of *Aggregator*. The abstract methods of the new generator simply call the corresponding abstract methods of the old generator which is possible because the result type is not fixed. In the new generator, the corresponding methods that are not parameterized by an aggregator also call their parameterized counterparts but have a different result type as those of the old generator.

3.3 Putting it all Together

We can execute an arbitrary implementation of the *MapReducer* interface on Hadoop using Screwdriver. To execute a generate-test-and-aggregate algorithm on Hadoop users need to compose its parts to get such an instance.

For example, if users have defined classes

- `MyGen extends Generator<MyElem,MyElem,MyRes,MyRes>`,
- `MyTest implements Test<MyElem,MyKey>`, and
- `MyAgg implements Aggregator<MyElem,MyRes>`

they can compose them as shown in Listing 5.

The aggregator is passed to the constructor of the generator which must call the corresponding constructor of its superclass. The test is passed to the *embed* method which returns a new generator that implements the generate-test-and-aggregate algorithm. Users can also use multiple calls to *embed* to embed multiple filters incrementally and only need to adjust the Val component of the $MapReducerCreator$ accordingly.

The interface $MapReducerCreator$ shown in Listing 6 is used by Screwdriver to create a list homomorphism and execute in on Hadoop.

4 Examples

4.1 Knapsack Problem

As a NP-hard problem, the *Knapsack* problem is how to fill a knapsack with items, each with certain value and weight, and to maximize total value of packed items without exceeding weight restriction of knapsack. Here, we have an example of knapsack which has a weight limitation of 100. We use $KnapsackItem$ denotes the items which in form of (w, v) pairs, and $IntWritable$ is the Hadoop wapper of Java type $Integer$.

As the generate-test-and-aggregate programming pattern, we should mainly implement the *aggregator*, *test*, and *generator* by implementing the corresponding classes provided by Generate-Test-and-Aggregate Library .

Firstly, an *Aggregator* class is defined as Listing 7. It defined a polymorphic semiring homomorphism form $(\wr KnapsackItem\int, \sqcup, \sqcup)$ to $(IntWritable, \max, +)$. Then a $Test$ is defined as Listing 8 showed. Its method *element* sets the upper bound of maximum sum of knapsack items. *postprocess* set the filter condition (less than 100)

Then, we define the monoid homomorphism $(M, \odot)$ by implementing the $Test$ interface.

At last, we use a *Generator* as Listing 9 (which already provided by the library). The efficiency of such algorithm is exponential with the number of items. But by theorem 2, we can get a pseudo polynomial algorithm.

As we have already known, the final efficient implementation is in the form of

$$generate_{\oplus,\otimes}(\lambda x \rightarrow aggregate\wr[x]\int)$$

With p processor, this algorithm will take $O((p+n/p)w^2)$ time.

4.2 Viterbi Algorithm

The *Viterbi* algorithm [15] is to find the most likely sequence of hidden states (the *Viterbi path*). Assume that S is the set of states, the observed events sequence is $X = x_1, x_2, ..., x_n$, and the hidden events sequence to be found is $Z = z_0, z_1, ..., z_n$. $P_{yield}(x_i \mid z_j)$ is probability of event x_i being caused by $z_j \in S$, and $P_{trans}(z_i \mid z_j)$ is probability of state of z_i appearing immediately after z_j The problem to find the maximum probable hidden events sequence can be formalized like:

$$\arg\max(\prod_{i=1}^{n} P_{yield}(x_i \mid z_i) P_{trans}(z_i \mid z_{i-1}))$$

Similarly, users have to implement necessary generator, test and aggregator. Listing 10 is

Listing 5. MyCreator.java

```
public class MyCreator
  implements MapReducerCreator<MyElem,WritableMap<MyKey,MyRes>,MyRes> {
    MapReducer<MyElem,WritableMap<MyKey,MyRes>,MyRes> createMapReducer() {
      return new MyGen(new MyAgg()).embed(new MyTest());
    }
}
```

Listing 6. MapReducerCreator.java

```
public interface MapReducerCreator<Elem,Val,Res> {
    MapReducer<Elem,Val,Res> createMapReducer();
}
```

Listing 7. MaxSumAggregator.java

```
public abstract class MaxSumAggregator<Elem le
    implements Aggregator<Elem,IntWritable> {

    public IntWritable zero() {
        return null;
    }

    public IntWritable one() {
        return new IntWritable(0);
    }

    public IntWritable plus(IntWritable left,
        if (left==null) {
            return right;
        } else if (right==null) {
            return left;
        } else {
            return new IntWritable (Math.max(
        }
    }

    public IntWritable times(IntWritable left
        if (left==null) {
            return null;
        } else if (right==null) {
            return null;
        } else {
            return new IntWritable(left.get() + right.get());
        }
    }
}
```

Listing 8. LimitedSumTest.java

```
public class LimitedSumTest
    implements Test<KnapsackItem,IntWritable> {
    private Integer maxSum;
    public LimitedSumTest(IntWritable maxWeight) {
        this.maxSum = maxWeight.get();
    }
    @Override
    public IntWritable element(KnapsackItem itm) {
        return new IntWritable(Math.min(maxSum,itm.getWei
    }
    @Override
    public Boolean postprocess(IntWritable res) {
        return res.get() < maxSum;
    }
    @Override
    public IntWritable identity() {
        return new IntWritable(0);
    }
    @Override
    public IntWritable combine(IntWritable left, IntWrita
        return new IntWritable( Math.min(maxSum, left.get
    }
}
```

the Java codes of generator, Listing 11 is the Java codes of test, and Listing 12 shows the Java codes of aggregator. As mentioned in Section 2, we can get a linear cost algorithm.

4.3 Experimental Results

The experiment results that evaluate the performance of our framework on Edu-Cloud which can provide upto 112 virtual-machine(VM) nodes. Each VM has 3 GB memory and 8-9GB available hard disk space. The VMs are connected by virtualized networks.

We tested an example of *Viterbi* algorithm and an example of *Knapsack* and evaluate the performance. For the example of *Viterbi*, with $10^2 \times 2^20$ (100 mega) long observed events sequence (?GB) as input data. The size of hidden states is 2, the For *Knapsack*, using $10^2 \times 2^{20}$ (100 mega) and $10^3 \times 2^{20}$ (1000 mega) items (?GB) as input data. Each item's weight is between 0 to 10, the maximum weight of knapsack is 100.

The Hadoop clusters are configured in different configurations (2 nodes, 4 nodes, 8 nodes, 16 nodes, 32 nodes).

The results are shown in Table 1. Knapsack$_s$ is tested with a sequential implementation of

Listing 9. SublistGenerator.java

```
public class SublistGenerator<Elem,Res> exten
                Generator<Elem,Elem,Res,Res> {
    public SublistGenerator(Aggregator<Elem,R
        super(aggregator);
    }
    @Override
    public <V> V identity(Aggregator<Elem,V>
        return aggregator.one();
    }
    @Override
    public <V> V element(Aggregator<Elem,V>
        return aggregator.plus(aggregator.one
    }
    @Override
    public <V> V combine(Aggregator<Elem,V>
        return aggregator.times(left,right);
    }
    @Override
    public Res postprocess(Res res) {
        return res;
    }
}
```

Listing 10. MarkingGenerator

```
public class MarkingGenerator
    <Elem extends Writable,Mark extends Writable,Val exte
    extends Generator<Elem,Marked<Elem,Mark>,Val,Val> {

    private List<Mark> marks;

    public MarkingGenerator(Aggregator<Marked<Elem,Mark>,
                            List<Mark> marks) {
        super(aggregator);

        this.marks = marks;
    }

    public <V> V identity(Aggregator<Marked<Elem,Mark>,V>
        return aggregator.one();
    }

    public <V> V element(Aggregator<Marked<Elem,Mark>,V>
        V markings = aggregator.zero();
        for (Mark mark : marks) {
            Marked<Elem,Mark> marked = new Marked<Elem,Ma
            V single = aggregator.singleton(marked);
            markings = aggregator.plus(markings,single);
        }

        return markings;
    }

    public <V> V combine(Aggregator<Marked<Elem,Mark>,V>
        return aggregator.times(l,r);
    }

    public Val postprocess(Val val) {
        return val;
    }
}
```

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

Program	1 node	2 nodes	4 node
Knapsack(m)	NA (NA)	1602 (1.00)	882 (1.
Knapsack(g)	NA (NA)	00 (0.00)	00 (0.
Knapsack$_s$	1391 (NA)	NA (NA)	NA (N

same algorithm. Knapsack(m) and Knapsack$_s$'s input data items number is 100 mega, Input items Knapsack(g) number is 1000 mega. Because of the limitation of HDD size, the Knapsack(g) only can be tested on clusters with more than 16 nodes. The Knapsack(m) also can only be tested on clusters with more than 4 nodes.

analysis here ...

5 Conclusions

In this paper, we show that the calculational theory for derivation of efficient parallel programs for generate-test-and-aggregate algorithms can be implemented on Hadoop in a concise and effective way upon the homomorphism-based framework for MapReduce in [11]. The new Java library we proposed for generate-test-and-aggregate on Hadoop provides better programming interfaces for users to specify their problems and run them efficiently on Hadoop. Our initial experimental results on two interesting examples indicate its usefulness in solving practical problems.

It is worth remarking that the algebra concepts such as monoid and semiring in generate-test-and-aggregate might seem to be difficult to understand at the first sight, but it is indeed very useful and make it easier for parallel program development.

We are now investigating how to extend our Java library from lists to trees so that many generate-test-and-aggregate algorithms on trees can be efficiently implemented on Hadoop.

References

[1] R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[2] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[3] Jeffrey Dean and Sanjay Ghemawat. 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*, pages 137–150, 2004.

Listing 11. MaxProdAggregator

```
public abstract class MaxProdAggregator<Elem>
    implements Aggregator<Elem,DoubleWritable> {

    public DoubleWritable zero() {
        return new DoubleWritable(0.0);
    }

    public DoubleWritable one() {
        return new DoubleWritable(1.0);
    }

    public DoubleWritable plus(DoubleWritable left, DoubleWritable right) {
        return new DoubleWritable(Math.max(left.get(), right.get()));
    }

    public DoubleWritable times(DoubleWritable left, DoubleWritable right) {
        return new DoubleWritable(left.get() * right.get());
    }
}
```

Listing 12. ViterbiTest

```
    private class ViterbiTest
        implements Test<Marked<Event,Transition<State>>,Trans<State>>{

        public BooleanWritable postprocess(Trans<State> trans) {
            return new BooleanWritable(trans.doesSomething());
        }

        public Trans<State> identity() {
            return new Trans<State>();
        }

        public Trans<State>
            element(Marked<Event,Transition<State>> marked) {

            return marked.getMark();
        }

        public Trans<State> combine(Trans<State> left, Trans<State> right) {
            return left.append(right);
        }
    }
```


[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. Generate, test, and aggregate: A calculational framework for systematic parallel programming with mapreduce. submitted for publication, July 2011.

[6] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generators-of-generators library with optimization capabilities in fortress. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 26–37, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[8] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming languages: Implementation, Logics and Programs. PLILP'96*, Lecture Notes in Computer Science 1140, pages 274–288. Springer-Verlag, 1996.

[9] Tan-Jan Ho and Bor-Sen Chen. Novel extended viterbi-based multiple-model algorithms for state estimation of discrete-time systems with markov jump parameters. *IEEE Transactions on Signal Processing*, 54(2):393–404, 2006.

[10] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[11] Yu Liu, Zhenjiang Hu, and Kiminori Matsuzaki. Towards systematic parallel programming over mapreduce. In *Proceedings of the 17th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, volume 152 of *ACM International Conference Proceeding Series*. ACM, 2006.

[13] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[14] Guy L. Steele Jr. Parallel programming and parallel abstractions in Fortress. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, page 1. Springer, 2006.

[15] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260 – 269, apr 1967.