A Generate-Test-Aggregate Parallel Programming Library

Systematic Parallel Programming for MapReduce

Yu Liu
The Graduate University for Advanced Studies
Tokyo, Japan
yuliu@nii.ac.jp

Kento Emoto
University of Tokyo
Tokyo, Japan
emoto@mist.i.u-tokyo.ac.jp

Zhenjiang Hu
National Institute of Informatics
Tokyo, Japan
hu@nii.ac.jp

ABSTRACT

Generate-Test-Aggregate (GTA for short) is a novel programming model for MapReduce, dramatically simplifying the development of efficient parallel algorithms. Under the GTA model, a parallel computation is encoded into a simple pattern: *generate* all candidates, *test* them to filter out invalid ones, and *aggregate* valid ones to make the result. Once users specify their parallel computations in the GTA style, they get efficient MapReduce programs for free owing to an automatic optimization given by the GTA theory.

In this paper, we report our implementation of a GTA library to support programming in the GTA model. In this library, we provide a compact programming interface for hiding the complexity of GTA's internal transformation, so that many problems can be encoded in the GTA style easily and straightforwardly. The GTA transformation and optimization mechanism implemented inside is a black-box to the end users, while users can extend the library by modifying existing (or implementing new) generators, testers or aggregators through standard programming interfaces of the GTA library. This GTA programming library supports both sequential or parallel execution on single computer and on-cluster execution with MapReduce computing engines. We evaluate our library by giving the results of our experiments on large data to show the efficiency, scalability and usefulness of this GTA library.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Algorithms, Design

Keywords

High-level Parallel Programming, Generate Test Aggregate Programming Model, MapReduce, Optimization, Functional Programming, Scala

1. INTRODUCTION

Google's MapReduce is the de facto standard for large scale data-intensive applications. Despite the popularity of MapReduce, developing efficient MapReduce programs for some optimization problems is usually difficult in practice.

As an example, consider the well-known 0-1 Knapsack problem: fill a knapsack with items, each of certain value v_i and weight w_i, such that the total value of packed items is maximal while adhering to a weight restriction W of the knapsack. This problem can be formulated as:

$$\begin{array}{l} \text{maximize } \sum_{i=1}^{n} v_i x_i \\ \text{subject to } \sum_{i=1}^{n} w_i x_i \leq W \ , \ x_i \in \{0,1\} \end{array}$$

However, designing an efficient MapReduce algorithm for the Knapsack problem is difficult for many programmers because the above formular does not match MapReduce model directly. Moreover, designing one for the problem with additional conditions is more difficult.

The Generate-Test-Aggregate (GTA for short) theory has been proposed in the previous work [10, 11] to remedy this situation. It synthesizes efficient MapReduce programs (i.e., parallel and scalable programs) for a general class of problems that can be *specified* in terms of `generate`, `test` and `aggregate` in a naive way of first generating all possible solution candidates, then keeping those candidates that pass a test of certain conditions, and finally selecting the best solution or making a summary of valid solutions with an aggregating computation. For instance, the Knapsack problem could be specified by a GTA program like this: generating all possible selections of items, keeping those that satisfy the constraint of total weight, and then selecting the one which has the maximum sum of value. Note that directly implementing such an algorithm by MapReduce is possible but not practical. Given n items, this naive program will generate $O(2^n)$ possible selections. Suppose there are 100 items and each item just takes 1 byte space, then the generated data will be about 10^{18}TB that is beyond any storage system's capability.

The GTA theory [10, 11] introduces a way to synthesize from such a naive program an work efficiency $O(n)$ and fully parallelized MapReduce program of which run-time efficiency is guaranteed by elimination of exponential blow-up, which is so-called the GTA optimization.

Although the previous work [10, 11] theoretically gave the methodology, it has not mentioned the implementation: It is non-trivial to implement the GTA theory with both the powerful optimization and a nice programming interface because of the gap between mathematical concepts and a practical programming language. Moreover, more work on practical programming with GTA is necessary in order to extract and indicate the real capability of the GTA theory, which will guide new users to this new world.

PMAM'2013, February 23, 2013, ShenZhen [Guangdong, China]

In this paper we present our implementation of a lightweight GTA library which is a functional programming platform allowing users to write GTA programs and execute them on local machines or large computer-clusters. Our main technical contributions are two folds. First, we design a generic program interface for users who don't know details of the GTA theorem and parallel programming, to let them use GTA in a black-box. Our library also shows how the theoretical GTA fusion can be implemented and executed on practical MapReduce engines. Second, we demonstrate the usefulness of our GTA library with some interesting examples, showing that lots of application problems can be easily and efficiently resolved by using our library.

The rest of the paper is organized as follows. After explaining the background knowledge in Section 2, we show the design and implementation of our library in Section 3. In Section 4 we introduce more details of the implementation. Then, we demonstrate the usefulness of our library using the Knapsack problem and other examples, and report our experimental results in Section 5. The related work is discussed in Section 6. Finally, we conclude the paper and highlight some future work in Section 7. The source code used for our experiments is available online[1].

2. BACKGROUND

In this section we briefly review the concepts of Generate-Test-Aggregate (GTA for short) theory [10, 11], as well as its background knowledge, list homomorphism [4, 8, 16] and MapReduce [9].

The notation we use to formally describe algorithms is based on the functional programming language Haskell [4]. 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 the operator $\circ$ over functions: by definition, $(f \circ g)\ x = f\ (g\ x)$. The identity element of a binary operator $\odot$ is represented by $\iota_{\odot}$.

2.1 List Homomorphism

A list homomorphism is a special, useful recursive function on lists. Naturally, it is a simple divide-and-conquer parallel computation [8, 16]. List homomorphisms have a close relationship with parallel computing that have been studied intensively [8, 16, 18], and

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 the identity element $\iota_{\odot}$ of $\odot$ such that the following equations hold.*

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

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

For instance, summation function *sum* on a list of integers can be defined in this format as a list homomorphism, in which $\odot$, $\iota_{\odot}$ and f are replaced with $+$, 0 and the identity function $\lambda x.x$:

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

Another example is function *sublists* that given a list as input produces a bag (multi-set) of all sublists of the list:

$$\begin{array}{ll} \mathit{sublists}\ [\,] & = \Lbag [\,] \Rbag \\ \mathit{sublists}\ [x] & = \Lbag [\,] \Rbag \uplus \Lbag [x] \Rbag \\ \mathit{sublists}\ (xs +\!\!+ ys) & = \mathit{sublists}\ xs \times_{+\!+}\ \mathit{sublists}\ ys. \end{array}$$

Here, $\Lbag a_1, \ldots, a_k \Rbag$ denotes a bag of elements $a_1, \ldots, a_k$, so $\Lbag [x] \Rbag$ is a singleton bag of singleton list $[x]$, the operator $\uplus$ denotes bag union, and $\times_{+\!+}$ denotes the cross concatenation of lists in two bags. For example,

$$\begin{array}{ll} \Lbag [] \Rbag \uplus \Lbag [1] \Rbag & = \Lbag [\,], [1] \Rbag \\ \Lbag [\,], [1] \Rbag \times_{+\!+} \Lbag [\,], [1] \Rbag & = \Lbag [\,], [1], [1], [1,1] \Rbag \\ \mathit{sublists}\ [1,1,2,3] & = \Lbag [\,], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3], \\ & \quad [1], [1,1], [1,1,2], [1,1,2,3], [1,1,3], [1,2], \\ & \quad [1,2,3], [1,3] \Rbag. \end{array}$$

Note that we may have duplications in a bag, and the order of elements are ignored in a bag. We write $\Lbag A \Rbag$ to denote the type of bags whose elements have type A.

Given a set M and an associative binary operator $\odot$ on M with its identity $\iota_{\odot}$, the pair $(M, \odot)$ is called a Monoid. For example, the set of integers with the usual plus operator forms a monoid $(\mathbb{Z}, +)$. Given a monoid $(M, \odot)$ and a function $f :: A \to M$, we have a unique list homomorphism $(\![f, \odot]\!) :: [A] \to M$.

2.2 List Homomorphisms on MapReduce

Google's MapReduce [9] is a popular programming model for processing large data sets in a massively parallel manner. Nowadays, several free, realistic implementations of MapReduce are available. Hadoop [2] is a famous open-source implementations of MapReduce using Java as its primitive language. Spark [31] is a fast inmemory MapReduce cluster implementation on Scala [24].

List homomorphisms fit well with MapReduce, because their input can be freely divided to sub-lists which can be distributed among machines. Then on each machine the programs are computed independently, and the final result can be got by a merging procedure. In fact, it has been shown that list homomorphisms can be efficiently implemented using MapReduce [20]. Therefore, if we can derive an efficient list homomorphism to solve a problem, we can solve the problem efficiently with MapReduce, enjoying its advantages such as automatic load-balancing, fault-tolerance, and scalability.

2.3 Generate, Test, and Aggregate

The GTA programming style and powerful fusion optimization [10, 11] have been proposed to synthesize MapReduce programs from naive specifications (GTA programs) in the following form.

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

A GTA program consists of a *generate* that generates a bag of intermediate lists, a *test* that filters out invalid intermediate lists, and an *aggregate* that computes a summary of valid intermediate lists. A GTA program in this form can be transformed into a single list homomorphism , if these components meet the condition of GTA fusion optimization. To understand the meaning, we review several important concepts.

DEFINITION 2 (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*

- *$\oplus$ is an associative and commutative operator with identity element $\iota_{\oplus}$,*
- *$\otimes$ is associative with identity element $\iota_{\otimes}$ and distributes over $\oplus$, and*

---

[1] https://bitbucket.org/inii/gtalib

- *$\iota_\oplus$ is a zero of $\otimes$.*

For example, a set of bags of lists forms a semiring $(\lbag [A] \rbag, \uplus, \times_{++})$ with the bag union and the cross concatenation for any element type A. The distributivity plays an important role in the optimization in the GTA theory.

Similar to the connection between a monoid and a list homomorphism, a semiring is naturally connected to a special recursive function on bags of lists.

Definition 3 (Semiring homomorphism). *Given arbitrary semiring $(S,\oplus,\otimes)$ and function $f :: A \to S$, function $shom :: \lbag [A] \rbag \to S$ is a semiring homomorphism from $(\lbag [A] \rbag, \uplus, \times_{++})$ to $(S,\oplus,\otimes)$, iff the following hold.*

$$\begin{array}{ll} shom\ (x \uplus y) & = shom\ x \oplus\ shom\ y \\ shom\ (x \times_{++} y) & = shom\ x \otimes\ shom\ y \\ shom\ \lbag [a] \rbag & = f\ a \\ shom\ \lbag\rbag & = \iota_\oplus \\ shom\ \lbag [\,] \rbag & = \iota_\otimes \end{array}$$

Since shom is uniquely determined by f, $\oplus$ and $\otimes$, we write $shom = \{\!\!\{ f,\oplus,\otimes \}\!\!\}$.

Since a semiring homomorphism consumes a bag of lists, it can be used as an aggregator in the GTA program. Actually, semiring homomorphisms in combination with generators and testers of specific kinds have very powerful fusions.

An example of semiring homomorphisms is aggregator *maxsum f* using semiring the max-plus semiring $(\mathbb{Z},\uparrow,+)$ to find the maximum among f-weighted sums of lists in a given bag:

$$\begin{array}{ll} maxsum\ f\ (x \uplus y) & = maxsum\ x \uparrow maxsum\ y \\ maxsum\ f\ (x \times_{++} y) & = maxsum\ x + maxsum\ y \\ maxsum\ f\ \lbag [a] \rbag & = f\ a \\ maxsum\ f\ \lbag\rbag & = -\infty \\ maxsum\ f\ \lbag [\,] \rbag & = 0 \end{array}$$

Here, $\uparrow$ is an operator to take the maximum of two operands. Readers can check whether *maxsum f* actually computes the maximum f-weighted sum of a given bag of lists, using the facts that every bag can be decomposed into union of singleton bags, and that every singleton bag of a list can be decomposed into cross-concatenation of singleton bags of singleton lists. For example, $\lbag [1,2,3],[2,3] \rbag = \lbag [1,2,3] \rbag \uplus \lbag [2,3] \rbag = (\lbag [1] \rbag \times_{++} \lbag [2] \rbag \times_{++} \lbag [3] \rbag) \uplus (\lbag [2] \rbag \times_{++} \lbag [3] \rbag)$.

Now, we introduce a class of generators that have good fusability with semiring homomorphisms.

Definition 4 (Semiring Polymorphic Generator). *A function polymorphic over semirings $(S,\oplus,\otimes)$*

$$generator_{\oplus,\otimes} :: (A \to S) \to [A] \to S$$

is called a semiring polymorphic generator.

Parameterised with semirings, a semiring polymorphic generator does different computation with different semirings. Particularly, using the semiring $(\lbag [A] \rbag, \uplus, \times_{++})$ of bags of lists, function $generator_{\uplus,\times_{++}}\ (\lambda a.\lbag [a] \rbag) :: [A] \to \lbag [A] \rbag$ is a generator that can be used in the GTA program. For example, abstracting the semiring in generator *sublists*, we have $sublists = sublists'_{\uplus,\times_{++}}\ (\lambda a.\lbag [a] \rbag)$ where

$$\begin{array}{ll} sublists'_{\oplus,\otimes}\ f\ [\,] & = \iota_\otimes \\ sublists'_{\oplus,\otimes}\ f\ [x] & = \iota_\otimes \oplus f x \\ sublists'_{\oplus,\otimes}\ f\ (xs +\!\!+ ys) & = sublists'_{\oplus,\otimes}\ f\ xs \otimes sublists'_{\oplus,\otimes}\ f\ ys. \end{array}$$

Moreover, we have the following powerful result to fuse such a generator with an aggregator of semiring homomorphisms.

Theorem 1 (Semiring Fusion [10]). *Given a semiring polymorphic generator $generator_{\oplus,\otimes} :: (A \to S) \to [A] \to S$ and a semiring homomorphism $\{\!\!\{ f,\oplus,\otimes \}\!\!\}$ to $(S,\oplus,\otimes)$, the following holds.*

$$\{\!\!\{ f,\oplus,\otimes \}\!\!\} \circ generator_{\uplus,\times_{++}}\ (\lambda a.\lbag [a] \rbag) = generator_{\oplus,\otimes}\ f$$

The left-hand-side of the equation is a GTA program without testers, in which $generator_{\uplus,\times_{++}}\ (\lambda a.\lbag [a] \rbag)$ is the generator and $\{\!\!\{ f,\oplus,\otimes \}\!\!\}$ is the aggregator. In this program, possibly an exponential number of intermediate lists are generated by the generator and then consumed by the aggregator, so that the total cost would be exponential in the length of the input list. On the other hand, the right-hand-side is usually an efficient program without such an exponential blow-up, because it does not use the heavy operator $\times_{++}$ but uses (possibly) lightweight operator $\otimes$. For example, the theorem says that program $maxsum\ (\lambda a.a) \circ sublists$, which given a list computes the maximum of sums of its all sublists, is equivalent to program $sublists'_{\uparrow,+}\ (\lambda a.a)$. This is easily verified because the program computes a sum of all positive numbers and it is clearly the maximum sum of all sublists.

Another important concept in GTA is *filter embedding*, which fuses an aggregator of semiring homomorphisms and a tester of a specific filter form:

Definition 5 (Homomorphic Tester). *If a tester test is a filter with a predicate defined with a function ok and list homomorphism $(\![f,\odot]\!)$, namely, $test = filter\ (ok \circ (\![f,\odot]\!))$, we call it a homomorphic tester.*

For example, in a GTA program for the knapsack problem, the tester to filter out item selections with too heavy total weights is a homomorphic tester as follows.

$$removeInvalidSelection = filter\ ((\leq w) \circ (\![getWeight,+]\!))$$

Here, the homomorphism $(\![getWeight,+]\!)$ computes the total weight of the given list, and the judgment $(\leq w)$ compares it with the weight limit to find invalid ones.

Now, we are ready to introduce the filter embedding:

Theorem 2 (Filter Embedding). *Given a homomorphic tester filter $(ok \circ (\![f,\odot]\!))$ in which the list homomorphism is to $(M,\odot)$ and a semiring homomorphism $\{\!\!\{ f,\oplus,\otimes \}\!\!\}$ to $(S,\oplus,\otimes)$, there exists a lifted semiring $(S^M,\oplus^M,\otimes^M)$, a lifted function f^M and function $postprocess_{ok}$ such that the following holds.*

$$\{\!\!\{ f,\oplus,\otimes \}\!\!\} \circ filter\ (ok \circ (\![f,\odot]\!)) = postprocess_{ok} \circ \{\!\!\{ f^M,\oplus^M,\otimes^M \}\!\!\}$$

This filter embedding is useful because we can remove a tester between a generator and an aggregator so that we can fuse them by the semiring fusion. Readers who are interested with these can reader the paper [10, 11] for details.

Reasoned by the theory of GTA, i.e., the combination of the filter embedding and the semiring fusion, a GTA program consisting of those components is eventually transferred to an efficient program $postprocess \circ generator_{\oplus^M,\otimes^M}\ f^M$. For example, the *naive* solution of Knapsack problem will generate $O(2^n)$ intermediate candidates and thus costs $O(n2^n)$, but the efficient final program only costs $O(n)$.

Note that, currently the GTA theory is an approach to constructing list homomorphisms, so the input of *generate* (also *aggregate*, and *filter*) is limited to *lists* but not *trees* or other data structures. GTA for trees/graphs is on our schedule as a future work.

3. ARCHITECTURE AND PROGRAMMING INTERFACE

```
trait MapReduceable[A,M,+R] extends ListHomo[A,M]{
  override def f(a:A):M
  override def combine(l:M,r:M):M
  def postProcess(a:M):R
}
```

Listing 1: Almost List Homomorphism

In this section we introduce our GTA programming environment. The GTA theory provides an approach to systematic derivation of list homomorphisms, so that we can get efficient MapReduce computation automatically once problems are specified GTA programs. Our library provides GTA programming interface to form a GTA program which produces an instance of so-called *MapReduceable*. The instance of *MapReduceable* adapts the list homomorphism to the MapReduce; its definition is shown in Listing 1. The `MapReduceable` has three methods, `f` corresponding to the function `f` of the list homomorphism, `combine` corresponding to the binary operator $\odot$, and a newly introduced method `postProcess` which is applied on the out put of the *REDUCE* processing as a final processing of whole computation [2]. *MapReduceable* can be used in any MapReduce engine or parallel frameworks which provides MapReduce style APIs. By passing GTA objects (instances of *MapReduceable*) to such MapReduce engines' programming interface we can construct fully scalable MapReduce computation quite easily.

3.1 Target Environment

Our GTA library is targeted big scale distributed/parallel computations on clusters which may have lots of computing nodes, but it also works well on single machine in either sequential or multi-threads model. Our implementation of GTA fusion is modular and easy to be extends.

Currently, our library officially supports three execution models: native, Spark and Hadoop. In native model, GTA works with Scala collection framework, and in Spark or Hadoop model GTA works with Spark or Hadoop, respectively.

3.2 GTA Programming Interface

There is a top level class named GTA for wrapping the GTA programming environment. The user should extend this top class to write his GTA program.

Usually a GTA expression (in Scala) is written like:

`val gta = generate(...) filter(...) aggregate(...)`.

`val gta` is a GTA object (MapReducable) which can be executed in parallel. For "(...)"s, the user should choose proper parameters respectively to the generator, tester and aggregator.

Given proper parameters, the GTA expression produces an instance of MapReducable corresponding to the efficient program synthesized by the GTA optimization. Then, the instance can be used in the supported execution models. To grasp an image of the GTA programming, concrete example for computing the *maximum sum of all segments (contiguous sublists) of an integer list* is shown in Listing 2. We will explain the details of the components *allSegments* and *maxSum* used in the program later in Section 3.4.

To make the programming easier, the GTA library predefined common generators, testers and aggregators. Users can choose them to compose various of programs. We list some useful generators, testers, and aggregators in Table 1. There are four generators, which, given a list, can generate all sublists (sublists), all prefix lists (prefixes), all continuous segments (segments), and paint colors (attaching some informations) to each element (coloring), respectively. Each tester in the table tests whether the sum (length, or its mod of some `k`) of a list is equal to (or less / more than) a constant value `c`. The aggregaters are for aggregating the generated lists to compute the maximum summation (maxSum), minimum summation (minSum) , maximum probability or the list which is the solution of above aggregations (select), respectively.. Here we just list the generic names for them, and the details will be explained by concrete examples (using more appropriate names of them, according to the context) in the following sections.

```
package Examples
import GTAS._

object userApp extends GTA[Int] with App {
    /* Spark job configuration */
    def ctx(contex:SparkContext,input:spark.RDD[Int])
    = {
    /* GTA expression */
    val gta=generate(allSegments) aggregate (maxSum)
    /* compute result. alist is the input list */
    val rst=alist.map(gta.f).reduce(gta.combine(_, _))
    println("rst")
  }
}
```

Listing 2: GTA Example: Maximum Segments Sum (Spark)

```
class AllSegments[I] extends
                        GeneratorCreater[I,I,T4] {
 def makeGenerator[S](s: Aggregator[I, S])
                  = new MapReduceable[I,T4[S],S]
 {
 override def f(i: I) =
  new T4(s.f(i), s.plus(s.id, s.f(i)),
       s.plus(s.id,s.f(i)),s.plus(s.zero,s.f(i)))
 override def combine(l: T4[S], r: T4[S]): T4[S] =
 {
 val ss = s.plus(s.plus(l._1, r._1),
                        s.times(l._2, r._3))
 val tails = s.plus(r._2, s.times(l._2, r._4))
 val inits = s.plus(l._3, s.times(l._4, r._3))
 val all = s.times(l._4, r._4)
 /* T4 is a type of four-tuple: (T,T,T,T)*/
 new T4( ss , tails, inits, all)
 }
 override val id: T4[S] =
               new T4(s.zero,s.zero,s.zero, s.id)
 /*The 1st element of the four-tuple is segments*/
 override def postProcess(a: T4[S]): S = a._1
  }
}
```

Listing 3: Generator of "All Segments"

Defining aggregators, testers, and generators.

For advanced users, we provide Scala trait/classes as programming interface to implement their own generators, testers and ag-

Table 1: Some Predefined Generators, Testers, and Aggregators

G	T	A
sublists	sum =, $\geq$, $\leq$ c	maxSum
prefixes	length =, $\geq$, $\leq$ c	minSum
segments	sum % k = c	maxProbablity
coloring	length % k =c	select

---

[2] Such an extended list homomorphism is called an *almost list homomorphism* [6, 17]

```
trait Aggregator[A,S] {
   def plus(l: S, r: S)
   def times(l: S, r: S)
   def f(a: A): S
   val id: S
   val zero: S
}
```

Listing 4: SemiringHomomorphism

```
abstract class MaxSum[T] extends Aggregator[T, Int] {
   def plus(l: Int, r: Int) = l max r
   def times(l: Int, r: Int) = l + r
   def f(a: T): Int
   val id: Int = 0
   //zero is -infty
   val zero: Int = Int.MinValue
 }
```

Listing 5: MaxSum

gregators.

In our library, an aggregator is a semiring homomorphism, and its base class `Aggregator`[`A`, `S`] is provided to implement user-defined aggregators by extending it. Here, the type parameter `A` is a type of elements in lists of the input bag, and `S` is a type of the carrier set of the semiring. Its methods `plus` and `times` correspond to the operators of a semiring, and `zero` and `id` are their identity elements, respectively. Listing 4 shows Scala code of the base class. For example, abstract class `MaxSum` (Listing 5) is an aggregator that finds the maximum among weighted[3] sums of lists in the given bag.

The library accepts homomorphic testers, which can be specified by the list homomorphism $(\![f,\oplus]\!)$ and the judgment function *ok*. Thus, a tester is represented by a specialized `MapReduceable` named `Predicate` of which `postProcess` method always returns Boolean value as shown in Listing 6 [4]. For example, `WeightLimit` shown in Listing 7 is an implementation of a tester to check whether a given list of items has total weight less than or equal to the weight limit w. We can check this condition by computing the sum of weights by a list homomorphism with the usual plus operator $+$ and then comparing the result with the limit w. Since we do not need an exact value of total weight greater than w, the implementation uses the cut-off by `min`(`w`+`1`). The reason why we use the cut-off will be explained later in Section 4.1.

In our library, a generator has to be a list homomorphism parameterized by a semiring as shown in Definition 4. How to implement it in the context of object-oriented language is a very interesting problem. We define a generic class named `GeneratorCreater` (shown in Listing 8) which has a generic (polymorphic) function to produce an instance of `MapReducable`:

$$\texttt{gen}[\texttt{S}](\texttt{s} : \texttt{Aggregator}[\texttt{A},\texttt{S}]) : \texttt{MapReduceable}[\texttt{I},\texttt{S},\texttt{P}[\texttt{S}]].$$

It takes an aggregator as its parameter, to produce a concrete instance of `MapReduceable`.

There are four type parameters in this `gen` function. The type parameter `I` is the type of elements of the input list, `A` and `S` are type parameters of aggregator, which we have explained. In addition to these input/output types, the function has the third type parameter `P` for its intermediate result, i.e., the result of its homomorphism part. This type can be parameterized by `S`, e.g, it can be `Id`[`S`] (equivalent to `S`), `Pair`[`S`] (equivalent to (`S`, `S`)), `Triple`[`S`] (equivalent to (`S`, `S`, `S`)), etc.

```
/*
 * Predicate is an almost-listhomomorphism
 * whose postProcess returns Boolean */
trait Predicate[M,T]
      extends MapReduceable[M,T,Boolean]

trait Countable[T] extends {def count:Int}

trait Finite[T] extends Iterable[T] with Countable[T]
/* FinitePredicate is Predicate on finite monoid*/
 trait FinitePredicate[M,T] extends Predicate[M,T]
                               with Finite[T]
```

Listing 6: Predicate and FinitePredicate

```
/*
 * tests if the total weight is <= the limit w
 */
object WeightLimit (w : Int)
    extends Predicate[KnapsackItem, Int]{
  def postProcess(t: Int) = t <= w
  def combine(l: Int, r : Int) = (l + r) min (w+1)
  def f(i: KnapsackItem) = (i.weight) min (w+1)
  val id = 0
}
```

Listing 7: Example of Predicate (WeightLimit)

Extending the `GeneratorCreater`, one can define a semiring polymorphic generator of the almost list homomorphism form by implementing the function `gen`. Notice that the function `gen` is a generic function such that the instance of `MapReduceable` can only be produced by methods of `s` whose type is `Aggregator`[`A`, `S`]. Therefore, this class is functionally equal to the class of functions $generator_{\oplus,\otimes} :: (A \to S) \to [A] \to S$. The Scala codes are showed in Listing 8. The generator `sublists` (actually, `sublists`$'$) can be implemented as the scala object `allSelects` shown in Listing 9, which simply implements the definition.

The function `gen` has slightly different type compared to the definition in Definition 4 [5], because we are focusing on generators whose computation patterns are suitable for the MapReduce model, while the original definition has no assumption on computation patterns. The generic function `gen` returns an instance of `MapReduceable`[`I`, `P`[`S`], `S`] of which computation consists of a list homomorphism from $[I]$ to $P[S]$ and a `postProcess` from $P[S]$ to S. We design such interface so that users can feel more comfortable to write generators like `Prefixes` shown in Listing 10, in which the homomorphism part computes pairs of type (S,S) while its final result is of type S.

Generator `Surfixes` to produce all suffixes can be implemented similarly. Combined with various of testers and aggregators, these generators can express a lot of problems. More examples can be found in the source code of our library.

3.3 GTA Expression

As we described, GTA programming is to choose or define generators, testers and aggregators to write GTA expressions. Here, we give the formal definition of GTA expression in the EBNF for better understanding.

---

[3] The weights are determined by method `f`.

[4] In Listing 6 there are some other traits for extending the `Predicate` and they will be explained in Section 4.1

[5] The equivalent of this `gen` is actually $generator'_{\oplus,\otimes} :: (A \to S) \to [A] \to P[S]$

```
trait GeneratorCreater[I,A,P[_]] extends BaseType{
 def gen[S](s:Aggregator[A,S]):MapReduceable[I,P[S],Any]
}
```

Listing 8: Polymorphic Generator

```
object allSelects extends GeneratorCreater[Int,Int,Id]{
 def makeGenerator[S](s: Aggregator[Int, S])
                = new MapReduceable[Int,Id[S],S] {
 override def f(i: Int):Id[S] =
    Id(s.plus(s.f(i), s.id))
 override def combine(l: Id[S], r: Id[S]): Id[S] =
    Id(s.times(l, r))
 override def postProcess(a: Id[S]): S = a
 override val id: Id[S] = s.id
   }
  }
```

Listing 9: Generator of "All Selects"

$$\begin{array}{lll}
\mathit{expr} & ::= & \mathit{GenTerm}\ \mathit{FiltTerms}\ \mathit{AggTerm} \\
\mathit{GenTerm} & ::= & \texttt{generate}\,'('\ \texttt{GeneratorCreater}\ ')' \\
\mathit{FiltTerms} & ::= & \varepsilon \mid \texttt{filter}\,'('\ \texttt{Predicate}\ ')'\ \mathit{FiltTerms} \\
\mathit{AggTerm} & ::= & \texttt{aggregate}\,'('\ \texttt{Aggregator}\ ')'.
\end{array}$$

Where the `generate`, `filter` and `aggregate` are the three keywords (actually, functions). Each of them has an argument which is an instance of `GeneratorCreater`, `Predicate` or `Aggregator`, appropriately. Note that one GTA expression may have multiple testers in this library. (Because we do implementation in Scala, ans Scala allows a single-line GTA expression being written in multiple lines that each line is one or more terms and with a '.' between any two terms.)

3.4 Solving Problems with GTA

We use three examples to show how to use the GTA library.

Knapsack problem and its variants.

First, recall the 0-1 Knapsack problem in the introduction. Similarly, we can firstly generate all possible candidates, then filter them using the predicate of weight limitation, finally, compute the total value on every remained candidate and choose the one which has the maximum total value. This problem can be programmed by using the `allSelects`, `maxTotalValue` extending the `maxSum` and the `WeightLimit`, as shown in Listing 11. If we want get the solution of knapsack items but not the maximum summation, a `select` aggregator can be performed here instead of `maxSum` (the details of `select` aggregator can be found in the library).

At a glance, the cost of the algorithms is exponential in the number of items. However, the GTA library optimizes it by using the GTA Fusion Theorem [10, 11] so that we can get an efficient algorithm whose cost is linear in the number of items (and quadratic with respect to the capacity of the knapsack).

A more complex example of *multi-constraints Knapsack problem* is shown in Listing 15. A new constraint on *the maximum number of items in a knapsack* is given in this case: the predicate `LengthLimit` checks the length of the given list in a similar way to `WeightLimit`. Not only check the exact length but also we can check whether the length (or summation) is less/more than a constant value c, or the length mod k is equal/less/more than c. Conceptually, arbitrary numbers of testers can be used. For example, we can add another constraint on *the minimum number of items in a knapsack* to extend the problem more. In these examples, we only find the best solutions (the maximum/minimum one), but also we can extend them to $\texttt{k}_{\texttt{th}}-\texttt{best}$ solutions.

```
class Prefixes[K] extends GeneratorCreater[K,K,Pair] {
 def makeGenerator[S](s: Aggregator[K,S]) =
      new MapReduceable[K, Pair[S], S] {
      def f(i: K) =
          Pair[S](s.plus(s.id, s.f(i)),
                   s.plus(s.zero, s.f(i)))
      def combine(l: Pair[S], r: Pair[S]):Pair[S]=
          Pair[S](s.plus(l.l, s.times(l.r, r.l)),
                   s.times(l.r, r.r))
      val id: Pair[S] = Pair[S](s.zero, s.id)
      def postProcess(a: Pair[S]): S = a.l
   }
  }
```

Listing 10: Generator of "All Prefixes"

```
...
 val allSelects = new AllSelects[KnapsackItem]
 val withLimit_100 = new WeightLimit(100)
 object maxTotalVal extends MaxSum[KnapsackItem] {
   def f(a: KnapsackItem): Int = a.value
 }
/* define a GTA */
 val gta = generate(allSelects) .
            filter (withLimit_100) .
              aggregate (maxTotalValue)
 println(  /* x is the input list*/
  gta.postProcess(x.map(gta.f).reduce(gta.combine))
 )
```

Listing 11: 0-1 Knapsack Problem

Maximum segments sum problem.

Next, let us consider the famous Maximum segments sum (`mss` for short) problem [3, 7, 17, 21, 23, 26]: Given a list of integers, find the maximum of sums of its all segments (contiguous sublists). This is a simplified problem of finding an optimal period in a history of changing values.

Under GTA programming model, the approach is simple: First, choose the `allSegments` generator that generates all the segments [17] of input list. Then choose the `maxSum` as the aggregator, that means to compute the maximum sum among all sums of segments. Listing 2 shows the GTA solution. This problem only need to write a few lines of Scala code. More additional predicates can be added to extending the `mss`, e.g., the segment should only contain at most one negative number, or maximum sum has to be divisible by 3. The `allSegments` as shown in Listing 3 is similar to `allSelects` and `allPrefixes` but more complex on data structures. We need a four-tuple $\texttt{type T4[T]} = (\texttt{T},\texttt{T},\texttt{T},\texttt{T})$ as the type of intermediate data structure. The details of how to construct such a list homomorphism can be found in [17].

Viterbi algorithm.

More complex problems can also be encoded by GTA. Hidden markov model (HMM) is known for its applications in temporal pattern recognition. The Viterbi algorithm [30]] is to find the most likely sequence of hidden states, i.e.,the Viterbi path , from the given sequence of observed events. In detail, given a sequence of observed events $(x_1,x_2,...,x_n)$, a set of states in a HMM model $S = (z_1,z_2,...,z_k)$, probabilities $P_{yield}(x_i \mid z_j)$ of events x_i being caused by states $z_,$, and probabilities $P_{trans}(z_i \mid z_j)$ of states z_i appearing immediately after states z_j, to compute the most likely sequence of $(z_1,z_2,...,z_n)$ is formalized as:

```
class MarkingGen[EV,ST](val states:Set[ST])extends
        GeneratorCreater[E,Tuple2[EV,(ST, ST)],Id]{
  val marks=for(x <- states ; y <-states) yield (x,y)
  type Marked=Tuple2[EV, (ST, ST)]
  def makeGenerator[S](s:Aggregator[Marked,S])=
    new MapReduceable[E,Id[S],S] {
      def f(i:E): Id[S]=(s.zero /: marks)
            {(z:S,mk:Mark)=>s.plus(z, s.f((i, mk)))}
      def combine(l:Id[S],r:Id[S]):Id[S]=s.times(l,r)
      val id:Id[S]=s.id
      def postProcess(a:Id[S]):S=a
    }
}
```

Listing 12: MarkingGenerator

$$\mathop{\mathbf{arg\,max}}_{Z\in S^{n+1}}\Big(\prod_{i=1}^{n} P_{yield}(x_i \mid z_i) P_{trans}(z_i \mid z_{i-1})\Big)$$

In [11], the approach of using GTA to compute above specification is introduced as: Firstly, we need to remove the index $i-1$ in the specification. To this end, we let the expression range over pairs of hidden states in $S \times S$ and introduce a predicate *trans* to restrict the considered lists of state pairs. Intuitively, $trans(p)$ is true if and only if the given sequence p of state pairs describes consecutive transitions

$$((z_0,z_1),(z_1,z_2),\ldots,(z_{n-2},z_{n-1}),(z_{n-1},z_n))$$

and false otherwise. Introducing the function

$$prob(x,(s,t)) = P_{yield}(x \mid t) P_{trans}(t \mid s)$$

the expression above can be transformed into the following equivalent expression.

$$\mathop{\mathbf{arg\,max}}_{\substack{p\in(S\times S)^n \\ trans(p)=True}} \Big(\prod_{i=1}^{n} prob(x_i, p_i)\Big)$$

Now, we are ready for building a Generate-Test-Aggregate algorithm. Given a set of marks, the generator `MarkingGenerator` in Listing 12 associates all possible mark to each element of the given list. For the Viterbi algorithm, the mark is the product set $S \times S$. For example, given $S = \{s_1, s_2\}$ and $x = [x_1, x_2]$, it can generate

$$\begin{array}{ll} \{ & [(x_1,(s_1,s_1)),(x_2,(s_1,s_1))], \\ & [(x_1,(s_2,s_1)),(x_2,(s_1,s_1))], \\ & [(x_1,(s_1,s_2)),(x_2,(s_1,s_1))], \\ & \vdots \\ & [(x_1,(s_2,s_2)),(x_2,(s_2,s_2))] \quad \}. \end{array}$$

The implementation of `MarkingGenerator` is almost the same as the `SublistGenerator`. The difference is that the method `f` sums up all possible associations of the marks, in which the type `Marked[Elem,Mark]` is the pair of the list element and the mark (Like painting colors on the input, so that the `MarkingGenerator` can be seen as a *coloring* generator.).

Among those associations of pairs of states to the input, we want to take only ones with valid transitions. To this end, *trans* is implemented as `ViterbiTest` shown in Listing 13. The method `f` extracts the mark, i.e., the associated pair of states, in which the pair has the type `Trans[State]` (a pair of states corresponds to a transition between states). The method `combine` appends two valid transitions (s,t) and (u,v) to make a new valid transition (s,v) if $t = u$. It returns a special value for invalid transitions otherwise.

```
abstract class ViterbiTest[E, Mark]
           extends Predicate[(E,Mark), Mark] {

  type MarkedTs = (E,Mark)
  def isTrans(a: Mark): Boolean
  def postProcess(a: Mark) = isTrans(a)
  def combine(l: Mark, r: Mark): Mark
  def f(a: MarkedTs): Option[Mark] = Some(a._2)
  val id: Mark
  ...//omit others
}
```

Listing 13: MarkingGenerator

```
abstract class MaxProdAggregator[T]
           extends Aggregator[T, Double] {
  def plus(l: Double, r: Double) = l max r
  def times(l: Double, r: Double) = l * r
  def f(a: T): Double
  val id: Double = 1.0
  val zero: Double = 0.0
}
```

Listing 14: ViterbiMaxProdAggregator

The method `postprocess` finally checks whether the input list had a valid transition or not.

The aggregator `ViterbiMaxProdAggregator` shown in Listing 14 implements the aggregator for computing the maximum probability, using the semiring $([0,1], max, \times)$ about the real numbers between 0 and 1. For simplicity, it computes not the Viterbi path but the Viterbi score (the maximum probability). The method `f` extracts the value from the marked element. Other parts are straightforward implementation of the semiring.

Finally, the composition of above components gives a GTA algorithm as

$$val\ vbGta\ =\ generate\ (vbGen)\ filter\ (vbTest)\ aggregate\ (vbAgg),$$

where `vbGen`, `vbTest` and `vbAgg` are instances of MarkingGenerator, ViterbiTest, and MaxProdAggregator respectively.

Similar to the knapsack problem, the program is optimized into a linear-cost parallel algorithm, although it looks an exponential algorithm at a glance. It is worth noting that we can compute the Viterbi path by replacing the aggregator with another aggregator based on a semiring to compute the path [14].

4. IMPLEMENTING THE GENERATE-TEST-AGGREGATE LIBRARY

We choose Scala to implement our library not only because it is a functional language with flexible syntax and strong type system, but also because of its performance and portability (Scala is JVM based so it is compatible with most of popular Java systems). We use Spark [31] and Hadoop [2] as MapReduce engines without any modification on them. To run a GTA program on a new MapReduce engine, the only thing we need to do is writing a Scala adapter for the engine, so that the MapReduce API can be invoked from the user's GTA program.

Design philosophy.

On implementation, we mainly concern the following three key points. The first is how to make an expressive, easy-to-use programming interface. We want to hide the complexity of intermediate computation and data structures, so that users only need to focus on how to convert their problems to the GTA style. Once

```
...
 val allSelects =new AllSelects[KnapsackItem]
 val withLimit_100 =new WeightLimit(100)
 val lessThan_10_items =LengthLimit[KnapsackItem](9)
 object maxTotalVal extends MaxSum[KnapsackItem] {
   def f(a: KnapsackItem): Int = a.value
 }
 /* define a GTA */
 val gta = generate(allSelects) .
            filter (withLimit_100) .
            /* add a new filter */
             filter (lessThan_10_items) .
              aggregate (maxTotalValue)
 println(
  /* x is input, postProcess returns the result*/
  gta.postProcess(x.map(gta.f).reduce(gta.combine))
 )
```

Listing 15: Extended 0-1 Knapsack Problem

users simply map their problems to navie GTA programs, they get scalable and efficient MapReduce programs.

The second is correctness and efficiency of the GTA fusion. As a practical problem, user's input and output data may have complex structures, and also generators, filters and aggregators chosen by the user may have a variety of combinations. Our library has to consider about such situations, and set constraints in the programming interface to prevent as many errors as possible. In addition, we make the fusion mechanism being type-safe (at library level, thanks to the type system of Scala), so that users can find semantic errors at compile time. The details of how to implement the fusion will be discussed in Section 4.1

The last one is compatibility and portability. It is important to clarify how our GTA can work together with other libraries/frameworks. Conceptually, our GTA can be used together with any MapReduce or other parallel frameworks which provide the standard MapReduce API , in Scala or Java. We also need to define a proper scope of our system so that adapters between GTA and execution engines can be easily made. Currently, we only provide unidirectional invocation: APIs of MapReduce engines can be used from GTA programs, but not the converse. This lets GTA expressions be easily defined, although it also requires the user's main MapReduce program being written in Scala. Since Scala is highly compatible with Java, this model works most of the time (most popular MapReduce engines provide Java or Scala API).

4.1 Semiring Fusion and Filter Embedding

The GTA fusion process can be described as a deterministic automaton shown in Figure 1. When `generate` function is invoked (by given a polymorphic generator as the parameter), an instance of `GEN` is created. `GEN` has two methods: `filter` and `aggregate` and keeps the polymorphic generator. When `filter` is invoked, it just composes new `Predicate` with previous one. When `aggregate` method is invoked, it embeds the `Predicate` into `Aggregator` to form another `Aggregator` with the lifted semiring, which is the filter embedding, and substitutes it to the polymorphic generator to produce a final `MapReduceable` instance, which is the semiring fusion.

The semiring fusion is quite clearly introduced in the previous work [10, 11]: It is just to substitute an efficient semiring to the polymorphic generator. However, the filter embedding needs more work to do in practice. Here, we discuss how the filter embedding is implemented.

Filters tupling.

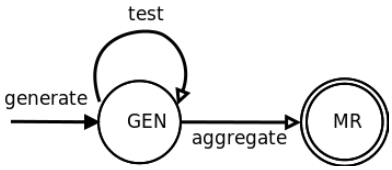

Figure 1: Automaton of GTA Fusion

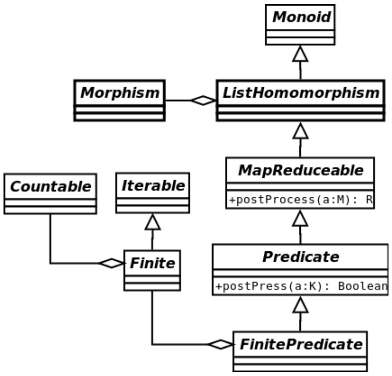

Figure 2: Finite Predicate

Every `filter` clause in a GTA expression introduce a `Predicate` that is to be involved in the fusion. We can combine all the list homomorphisms together and create a new one that computes all at once.

Usually, tupling two list homomorphisms into one is simple. Let $hh = (\ (\![f_1,\odot_1]\!),(\![f_2,\odot_2]\!)\ )$. We have:

$$
\begin{aligned}
&hh\ [\,] && = (\iota_{\odot_1}\ ,\ \iota_{\odot_2})\\
&hh\ [a] && = (f_1\ a\ ,\ f_2\ a)\\
&hh\ (x +\!\!+ y) && = hh\ x \odot hh\ y\\
&\quad \textbf{where}\ (hx_1,hx_2) \odot (hy_1,hy_2) = (hx_1 \odot_1\ hy_1\ ,\ hx_2 \odot_2\ hy_2)\ .
\end{aligned}
$$

By the *tupling*, multiple `filter` clauses can be merged to one. We fuse this composed filter (a `Predicate`) together with generator and aggregator, to form the final GTA `MapReduceable` object.

Lifted semiring.

In the filter embedding, the carrier set M of monoid $(M,\odot)$ should be finite, to guarantee the efficiency of the final program that uses the semiring lifted by M. The key point of implement the lifted semiring is to define a Scala class that can wrap the finite monoid and semiring. In order to define the operators $\oplus_M, \otimes_M$, the elements of the set M must be enumerated in constant time. Thus, we defined `FinitePredicate` to resolve this problem. In order to guarantee that the final GTA program is efficiently computable, the composed filters must be a (or subtype of) `FinitePredicate`.

Finite monoid and finite predicate.

In our implementation, a finite monoid is a monoid whose domain is a finite set. Using Scala to define such a set we can use the `Countable` and `Iterable` traits. An object that inherits from `Countable` must implement a count method. And `Iterable` requires all its concrete subclasses to implement an iterator. We defined a Scala class named `FinitePredicate` (Listing 6). It is a `Predicate` with a finite domain. Figure 2 shows the class inheritance.

To guarantee the linear cost of the final program, filter clauses in a GTA expression have to be under the constraint: all the `filters`

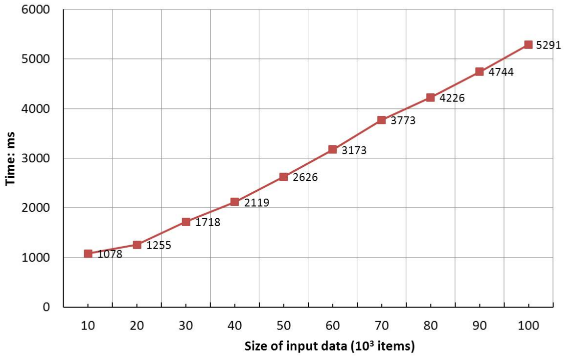

Figure 3: Execution Time of GTA Programs on Single CPU

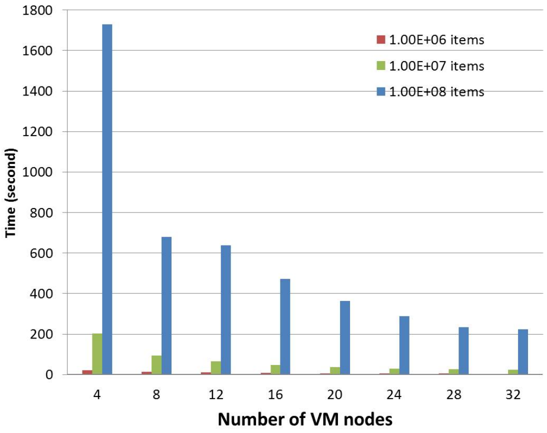

Figure 4: Execution Time of GTA Programs on Spark Clusters

take `FinitePredicate` as the parameters. Otherwise the computational cost of the final program is not guaranteed.

The rest of implementing filter embedding, is to use the tupled `tester` and together with the `aggregator` to construct the lifted semiring (Definition 2) which is introduced in the previous work [10, 11]. We use a `Map` data structure to denote the domain of monoid $(S^M, \odot^M)$ where the keys (index) of the `Map` are elements in set M and thus $\odot^M$ can be defined. Readers who are interested with this could find details in our source code.

4.2 Serialization

For MapReduce frameworks like Spark and Hadoop, Java objects used as output of *MAP* and *REDUCE* need to be serialized for saving in file system or being transfered through networks. The intermediate data produced by GTA also need to be serialized in some way. Currently, our GTA library uses different serializations for Spark and Hadoop. For Spark, the serialization is done by using Java serialization, and for Hadoop we use the Hadoop Writable interface.

There are some universal data serialization frameworks, such as Avro [1] and Protocol Buffers [29], which provide common protocols and supporting multiple languages. We are considering apply such approaches in future.

5. EXPERIMENTS

We evaluate our GTA library on both sequential and parallel (distributed) ways and show the efficiency and scalability of it.

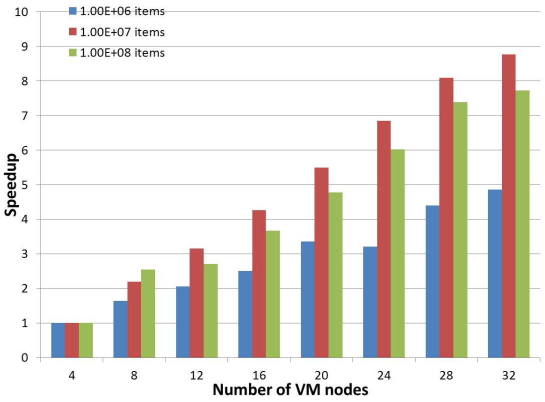

Figure 5: Speedup of GTA-Knapsack on Spark Clusters

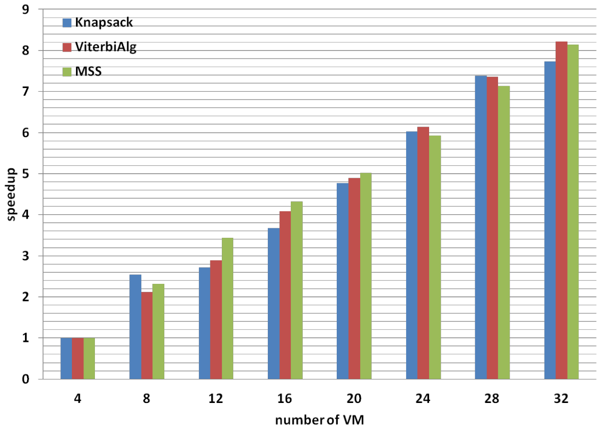

Figure 6: Speedup of GTA Programs on Spark Clusters (size of input=10^8)

5.1 Algorithmic Efficiency and Scalability

Algorithmic efficiency.

We test the algorithmic efficiency of our GTA library by running a GTA-Knapsack program in local environment. In all the following test cases, the knapsack item's weight is in range (0, 10] and the capacity of the knapsack is 100. The machine we used has a 2 GHz Intel Core Duo CPU (two cores) with 2 GB RAM and the Java VM heap size was set as: "JAVA_OPTS=-Xmx1024m -Xms256m". The Scala version is 2.9.2 final.

The first test case is comparing the GTA-Knapsack program to a Knapsack program in a naive algorithm (brute-force) which generates all sublists, then filters them and make max-sum on the rest ones. Table 2 shows the comparison of running time. Obviously,

Table 2: Comparison of Naive and GTA Knapsack Programs

length	naive (ms)	GTA (ms)
8	47	24
12	106	27
16	271	53
20	6838	65
24	OutOfMemoryError	52

the GTA-Knapsack is much faster than the naive one. Optimized dynamic programming solution for 0-1 Knapsack problem can runs in $O(nW)$ time, which is theoretically faster than our GTA-Knapsack ($O(nW^2)$ time). But without carefully optimizing the program, a dynamic programming solution could be even slower. An notable superior of our GTA library is that optimization is transparent to programmers.

The second experiment tests the GTA-Knapsack program with different size of input data. The running time of GTA-Knapsack is linear with the increasing of input data size (from 10×10^3 to 10×10^4). Figure 3 shows the linear algorithmic efficiency of our GTA-Knapsack.

Evaluation on MapReduce clusters.

Our MapReduce clusters are built on the Edubase-Cloud (at National Institute of Informatics, Japan). It is a cloud computing environment like Amazon EC2. We have authority to use up to 32 virtual-machine (VM) nodes. We configured Mapreduce (Spark and Hadoop) clusters with 4, 8, 12, 16, 20, 24, 28, and 32 nodes. Each VM has one single-core CPU, 3 GB RAM and 8-9 GB available hard disk space. We prepared three sets of randomly generated items for the *Knapsack* programs: 1×10^6, 1×10^7 and 1×10^8 items. Experiments on Vitirbi Algorithm and Maximum segments sum are evaluated in similar manner (we use programs randomly generated all input data). The results are given in Figure 4 , 5 and 6. When input data size is too small (in case of using data set of size 1×10^6), the system overhead takes heavier ratio on the timing-results. For data with appropriate size, the results show that all the GTA programs gained near-liner speedup when increase the computing nodes. The evaluation of GTA programs on Hadoop clusters also shows the similar speedup though absolute performance is slower on same dataset, because of higher system overhead.

6. RELATED WORK

The research on parallelization via derivation of list homomorphisms has gained great interest since [8, 28, 32]. The main approaches include the function composition based method [5, 12, 19], the third homomorphism theorem based method [15, 22], and the matrix multiplication based method [27]. It has been shown that homomorphism-based approaches can be used on systematic programming of MapReduce [20].

GTA [10, 11] is a new approach to systematic development of efficient parallel programs and/or list homomorphisms, in which feaures of semirings are maximally exploited to connect naive design and efficient impementation, so that it dramatically simplifies the development of efficient parallel algorithms. However, there lacks of implementations which can support practical MapReduce programming. Our work is a continuation of previous research on GTA and making it work for common MapReduce frameworks.

There are also several high-level domain specific languages build upon MapReduce (Hadoop), such as Google's Sawzall [25], Apache Pig Latin [13], and so on. They wrap MapReduce (Hadoop) and provide optimization functionalities to optimize users' programs. Currently, they do not have optimizations similar to the GTA fusion. We believe that GTA can also be imported into the design of these languages as a primitive optimization choice.

7. CONCLUSIONS

In this paper, we show that the Generate-Test-Aggregate theory for systematic derivation of efficient parallel programs can be implemented on MapReduce in a concise and effective way. Our framework on Scala provides a convenient GTA programming interface for users to specify their problems in the GTA pattern easily, and to run them efficiently on multi-thread, Spark and Hadoop environments. Our initial experimental results on several interesting examples indicate its usefulness in solving practical problems.

Currently, the GTA theory is based on the list homomorphisms, and our GTA library is concentrated on covering problems which are based on the list data structure. A very attractive future work is to apply the GTA style programming to problems based on trees/-graphs, which is not supported by our GTA library yet.

We are now investigating interesting applications in the GTA framework, and challenging to extend our library from lists to trees and graphs so that GTA style algorithms can be efficiently implemented for processing large trees/graphs such as data from social networks.

References

[1] Apache Software Foundation. Avro. http://avro.apache.org/.

[2] Apache Software Foundation. Hadoop. http://hadoop.apache.org/.

[3] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer, 1987.

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

[5] W. N. Chin, S. C. Khoo, Z. Hu, and M. Takeichi. Deriving parallel codes via invariants. In *International Static Analysis Symposium (SAS2000)*, volume Lecture Notes in Computer Science 1824, pages 75–94. Springer Verlag, 2000.

[6] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical report, Department of Computer Science, University of Edinburgh, 1993.

[7] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, 1993.

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

[9] J. Dean and S. 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.

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

[11] K. Emoto, S. Fischer, and Z. Hu. Generate, test, and aggregate - a calculation-based framework for systematic parallel programming with mapreduce. In *Programming Languages and Systems, 21st European Symposium on Programming (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2012.

[12] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 135–146, 1994.

[13] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.

[14] J. Goodman. Semiring parsing. *Computational Linguistics*, 25:573–605, 1999.

[15] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 401–408. Springer, 1996.

[16] 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.

[17] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 553–562. Springer, 1996.

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

[19] Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in calculational forms. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 316–328. ACM Press, 1998.

[20] Y. Liu, Z. Hu, and K. 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.

[21] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 146–155. ACM, 2007.

[22] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 146–155. ACM Press, June 2007.

[23] S.-C. Mu. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 31–39, New York, NY, USA, 2008. ACM.

[24] M. Odersky and al. The scala language specification, version 2.9. Technical report, EPFL Lausanne, Switzerland, 2011.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, Oct. 2005.

[26] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear time algorithm for solving maximum-weightsum problems. In *The 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 137–149. ACM Press, 2000.

[27] S. Sato and H. Iwasaki. Automatic parallelization via matrix multiplication. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*, pages 470–479. ACM, 2011.

[28] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *NATO ASI Workshop on Software for Parallel Computation, NATO ARW "Software for Parallel Computation"*, volume 106 of *F*. Springer-Verlag NATO ASI, 1992.

[29] K. Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008.

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

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[32] P. G. H. Zully N. Grant-Duff. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.