Parallel Computing 40 (2014) 116–135

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A Generate-Test-Aggregate parallel programming library for systematic parallel programming

Yu Liu [a,c,*], Kento Emoto [b], Zhenjiang Hu [c,a]

[a] The Graduate University for Advanced Studies, Tokyo, Japan
[b] Kyushu Institute of Technology, Iizuka, Japan
[c] National Institute of Informatics, Tokyo, Japan

ARTICLE INFO

Article history:
Available online 10 December 2013

Keywords:
High-level parallel programming
Generate-Test-Aggregate algorithm
Program transformation
Program calculation
MapReduce
Functional programming

ABSTRACT

The *Generate-Test-Aggregate* (GTA for short) algorithm is modeled following a simple and straightforward programming pattern, for combinatorial problems. First, *generate* all candidates; second, *test* and filter out invalid ones; finally, *aggregate* valid ones to make the final result. These three processing steps can be specified by three building blocks namely, *generator*, *tester*, and *aggregator*. Despite the simplicity of algorithm design, implementing the GTA algorithm naively following the three processing steps, i.e., brute-force, will result in an exponential-cost computation, and thus it is impractical for processing large data. The theory of GTA illustrates that if the definitions of *generator*, *tester*, and *aggregator* satisfy certain conditions, an efficient (usually near-linear cost) MapReduce program can be automatically derived from the GTA algorithm.

The principle of GTA is attractive but how to make it being practically useful, remains as an important and challenge problem due to the complexity of GTA program transformations. In this paper, we report on our studying and implementation of a practical GTA library (written in the functional language Scala) which provides a systematic parallel programming approach for big-data analysis with MapReduce. The library provides a simple functional style programming interface and hides all the internal transformations. With this library, users can write parallel programs in a sequential manner in terms of the GTA algorithm, and the efficiency of the generated MapReduce programs is guaranteed systematically. Therefore, parallel programming for many problems could become no more a tough job. We demonstrate the usefulness of our GTA library on some interesting problems involving large data and show that lots of applications can be easily and efficiently solved by using our library.

1. Introduction

Google's MapReduce [1] is a famous parallel programing model that simplifies the parallel and distributed processing of large scale data. Despite the simplicity of MapReduce, developing efficient MapReduce programs is still a challenge for certain optimization problems, because users are required to make particular divide and conquer algorithms that must fit the execution model of MapReduce.

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 the weight restriction W of the knapsack. This problem can be formulated as:

* Corresponding author at: National Institute of Informatics, Tokyo, Japan. Tel.: +81 03 4212 2611.
E-mail addresses: yuliu@nii.ac.jp (Y. Liu), emoto@ai.kyutech.ac.jp (K. Emoto), hu@nii.ac.jp (Z. Hu).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.11.002

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leqslant W, \quad x_i \in \{0, 1\}$$

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

The theory of GTA has been proposed [2,3] 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 by first generating all possible solution candidates, keeping those candidates that have passed 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: generate all possible selections of items, keep those that satisfy the constraint of total weight, and then select the one which has the maximum sum of values. Note that directly implementing such an algorithm by MapReduce is not practical, because given n items, the naive program will generate $O(2^n)$ possible selections. The theory of GTA gives an algorithmic way to synthesize from such a naive program to a fully parallelized MapReduce program that has $O(n)$ work efficiency.[1]

The previous work [2,3] described the GTA programming style and the GTA fusion theorems theoretically, but it did not mention any about the implementation: because of the gap between mathematical concepts and practical programming languages, it is non-trivial to implement the GTA theory in such a way that it yields both a powerful optimization and a nice programming interface. Moreover, there has to be more work on GTA in order to identify its real capabilities for practical parallel programming and to make a sufficient guide for new users.

In this paper we present our implementation of a lightweight GTA library (in Scala [4]) that is a functional programming platform allowing users to write GTA programs and execute them on local machines or large computer clusters. Our main technical contribution is two fold. First, we design a generic program interface that allows users to write their programs in a sequential manner following the GTA programming style, without special requirements for knowing theoretical details of GTA or parallel programming. The GTA library takes the responsibility of transforming user-specified programs to efficient MapReduce programs, and executing them on practical MapReduce engines. Second, we demonstrate the usefulness of our GTA library with many interesting examples and show that lots of problems can be easily and efficiently solved by using our library.

The rest of the paper is organized as follows. After explaining the background in Section 2, we introduce the programming interface of our GTA library in Section 3. More examples and details about GTA programming are introduced in Section 5. Section 4 describes the implementation of the library in detail. Then, we describe the experimental results in Section 6. The related work is discussed in Section 7. Finally, we conclude the paper and highlight the future work in Section 8. The source code used for our experiments is available online.[2]

2. Background

In this section we briefly review the concepts of GTA [2,3] as well as its background knowledge, list homomorphism [5–7] and MapReduce [1]. The notation we use to formally describe algorithms is based on the functional programming language Haskell [5]. Function application can be written without parentheses, i.e., fa equals $f(a)$. Functions are curried [5], and function application is left associative, thus, fab equals $(fa)b$. Function application has higher precedence than operators, so $fa \oplus b = (fa) \oplus b$. We use the operator $\circ$ over functions: by definition, $(f \circ g)x = f(gx)$ and $(f \vartriangle g)x = (fx, gx)$. The identity element of a binary operator $\odot$ is represented by $\iota_\odot$.

This GTA library is implemented in Scala [4] that is an object-oriented functional language. Basic knowledge of Scala and generic programming is needed to understand the source code in this paper.

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 [6,7]. List homomorphisms are closely related to parallel computing and have been intensively studied [6–8].

Definition 1 (*List homomorphism*). A 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.

$$h[\,] = \iota_\odot$$
$$h[a] = f\ a$$
$$h(x + + y) = h\ x \odot h\ y.$$

[1] The efficient MapReduce program only produces $O(n)$ intermediate data.
[2] https://bitbucket.org/inii/gtalib.