A Calculational High-Level Parallel Programming Framework for Data-Intensive Computing

Yu Liu

Department of Informatics

The Graduate University for Advanced Studies

2014

Acknowledgements

I would like to express my deep-felt gratitude to my supervisor, Prof. Zhenjiang Hu of the Information Systems Architecture Research Division at National Institute of Informatics (also Department of Informatics at the Graduate University for Advanced Studies), for his advice, encouragement, enduring patience and constant support. He was never ceasing in his belief in me, always providing clear explanations when I was lost, and always giving me his time, in spite of anything else that was going on.

I also would like to express my deep gratitude to my advisers Professor Shin Nakjima and Assistant Professor Soichiro Hidaka of National Institute of Informatics, for their insightful comments and kind encouragements to carry out this research.

I also wish to thank Professor Hideya Iwasaki at the University of Electro-Communications, Associate Professor Kiminori Matsuzaki at the Kochi University of Technology, Assistant Professor Kento Emoto at Kyushu Institute of Technology and Assistant Professor Hiroyuki Kato at National Institute of Informatics as well as Doctoral student Shigeyuki Sato at the University of Electro-Communications and all other members of our IPL laboratory. Their suggestions, comments and additional guidance were invaluable to the completion of this work.

Additionally, I want to thank the Graduate University for Advanced Studies at Informatics Department professors and staffs for all their hard work and dedication, providing me the means to finish my five years doctoral study.

Abstract

Writing parallel programs has been proven over the years to be a difficult task, requiring various specialized knowledge and skills. Recent years, how to process large scale data efficiently becomes a crucial problem in practice. Traditional parallel programming models and frameworks like MPI, OpenMP, have been widely used to write efficient parallel programs for many years. However, for data-intensive processing, programming under these frameworks usually directly leads to overly complex, non-portable, and often unscalable code.

High level data-intensive parallel programming frameworks such as MapReduce become very popular in data intensive processing and analysis, which are actually based on a few specialized algorithmic skeletons, e.g., Map and Reduce functions. Many researchers and programmers implement various of algorithms on data mining, machine learning or nature language processing, using MapReduce-like frameworks. Despite the success of MapReduce-like frameworks, developing efficient parallel programs is still a challenge. The main difficulties are two folded.

The first difficulty is the programmability of these frameworks. The programming interface is restricted to a few higher order functions so that it is difficult to represent various algorithms in such a high abstraction level. There are many research papers explaining how to transform a classic parallel algorithm to MapReduce, which also confirms our observation that programming under such a programming model is still a non-trivial task.

The second one is that optimization for certain parallel programs developed by MapReduce is difficult. To measure and adjust the performance of MapReduce programs usually causes lots of efforts, not only because of the complex software/hardware environments (clusters containing thousands of computers) but also because of the large amount of input data. The static analysis and optimization is also not easy because of the in-black-box processes of MapReduce.

An ideal programming model is that users just need to write specifications of their problems in a natural (even naive) way, but the programming framework can produce efficient code for parallel executing.

The main contribution of this thesis is that we have developed a high-level programming framework for resolving above problems, based on program transformation. The objective is to introduce a simple, expressive, and flexible programming interface for representing algorithms in a structural way and the underlying optimizing mechanism of the framework can automatically transform the high-level structural programs to efficient and scalable parallel programs.

Under the programming model of our framework, users write clear specifications of their problems in terms of structural recursions in sequential manner, without any concern of efficiency or parallelism. Then the structural recursions will be transformed to parallel implementations of algorithmic skeletons by the techniques of program calculation. Moreover, this programming framework enforces deterministic semantics of programs and thus can simplify composing, optimizing, porting, reasoning about, debugging, and testing parallel programs.

Inside this framework, we provide several libraries that support writing efficient parallel programs in Java or other popular languages. We show that many problems which are difficult to be expressed by MapReduce can be easily implemented in our framework. Experimental results show that using our framework, users can easily write efficient parallel programs to deal with large input data.

Contents

Contents vii

List of Figures xi

List of Tables xiii

Nomenclature xiii

1 Introduction 1

1.1 Background 1

1.2 High-Level Parallel Programming Models 3

1.3 Automatic Parallelization and Optimization by Program Calculation 5

1.4 Contributions and Organization of the Thesis 7

2 Algebraic Data Structures and Program Calculation Laws 9

2.1 Basic Notations 9

2.1.1 Functions 10

2.1.2 Sets 10

2.1.3 Graphs 10

2.2 Theory of Lists 11

2.2.1 Algebraic Data Type of Lists 11

2.2.2 List homomorphism 12

2.2.3 Basic Parallel Skeletons On Lists 14

2.2.4 Basic Program Calculation Laws on Lists 15

2.3 Theory of Trees 16

2.3.1 Algebraic Data Type of Trees 16

2.3.2 Tree Homomorphisms 17

2.4 Related Work 17

3 Calculational Approach to Constructing Parallel Programs on Lists 19
3.1 Functional Formalization of MapReduce 19
3.2 Implementing List Homomorphisms on MapReduce 22
3.2.1 Data Models of Lists 23
3.2.2 A MapReduce Algorithm for Implementing Homomorphisms . . . 24
3.3 An Accumulative Computation Pattern on MapReduce 26
3.3.1 Accumulative Computations 27
3.3.2 The Programming Interface for Accumulation 30
3.4 A Generate-Test-Aggregate Parallel Programming Pattern on MapReduce . 32
3.4.1 The GTA Programming Interface 37
3.4.2 Solving Problems with GTA 40
3.5 Related Work 45

4 Calculational Approach to Constructing Parallel Programs on Graphs 47
4.1 Tree Decomposition of Graph 48
4.2 Parallel Computation on Tree Decompositions of Graphs 49
4.2.1 Transforming Trees to Zippers 50
4.2.2 Partition Binary Trees by Zippers 52
4.2.3 Programming Interface for Parallel Algorithms on Hierarchical Zippers 55
4.2.4 Representations of MapReduce Programs 56
4.3 On Resolving Maximum Weighted Independent Set Problem 58
4.3.1 A Generate-Test Algorithm for Combinatorial Problems on Graphs 58
4.3.2 A Bottom-up Algorithm on Tree Decompositions 59
4.3.3 The Parallel Algorithm on Zippers 61
4.3.4 Evaluation 62
4.4 Related Work 63

5 A Practical Approximation Approach to Optimization Problems 69
5.1 The Target Set Selection Problem 69
5.1.1 Formal Definition of TSS 71
5.1.2 An Extension of the TSS Problem 72
5.2 Using Treewidth-Bounded Partial Graphs to Approximate TSS 72
5.2.1 Graph Reduction for TSS 73

5.2.2 The Maximum Partial K-Tree Problem . . . 77
5.2.3 An Exact Algorithms for Computing TSS . . . 79
5.2.4 Implementation and Evaluation for the Target Set Selection Problem 82
5.3 Related Work . . . 84

6 Implementation and Evaluations 87
6.1 List Homomorphism Wrapper of MapReduce . . . 87
6.1.1 Manipulation of Ordered Data . . . 87
6.1.2 Implementing Homomorphism by Two Passes of MapReduce . . . 88
6.1.3 Implementation Issues . . . 89
6.1.4 Experiments and Evaluation . . . 89
6.2 Implementation of The Accumulation Library Based on MapReduce . . . 91
6.2.1 A Two-pass MapReduce Algorithm for Accumulation . . . 91
6.2.2 The MapReduce Implementation for General Accumulation . . . 93
6.2.3 Discussions on Efficiency . . . 95
6.2.4 Experiments and Evaluations . . . 97
6.3 Implementation of The GTA Framework . . . 100
6.3.1 System Architecture . . . 101
6.3.2 Algebraic Data Structures and Building Blocks of GTA . . . 102
6.3.3 Semiring Fusion and Filter Embedding . . . 108
6.3.4 Serialization . . . 110
6.3.5 Performance Evaluation of Our GTA Framework . . . 111

7 Conclusion 117
7.1 Summary of the Thesis . . . 117
7.2 Future Works . . . 118

References 119

List of Figures

1.1 Parallel Data-processing Flow of MapReduce 4
1.2 Overview of Our Calculational Framework for High-level Parallel Programming 7

3.1 System Overview of Screwdriver 21
3.2 The GTA Program Transformation 38

4.1 An example of a tree decomposition of width two: blue circles (big circles) denote the bags; red dashed lines connect the same vertices between adjacent bags. 49
4.2 A Graph G, A Tree Decomposition of G and A Nice Tree Decomposition of G. 50
4.3 A Replace Node and A Join Node in Nice Tree Decompositions. 51
4.4 The contractL operation 51
4.5 The rake and compress Operations 52
4.6 An example of a zipper on a bianry tree, which expresses a path from the root to the black leaf. The path is shown in the blue (thicker) line. 52
4.7 An example of a hierarchical zipper: the subtrees in red dotted rectangles are partitioned to new zippers pointed by the arrows; the id of a subtree is shown on its top. 53
4.8 A Weighted Graph and Its Tree Decomposition. The Weight of Each Node is Same as Its Id. 66
4.9 An Example to Show Computation on a Zipper. 66
4.10 Running time of the MWIS problem with cores. 66
4.11 Speedup of the MWIS problem with cores. 67
4.12 Memory usage of the MWIS problem with cores. 67

5.1 An example of TSS problem under the deterministic threshold model 71
5.2 Structure “a vertex connects to a $t(x)$=1 vertex" 74
5.3 Structure “two vertices has common $t(x)$=1 vertices in between" 74

5.4 Structure “twins that can be reduced" 76
5.5 The running time of the partial k-tree generating algorithm 79

6.1 Time Consuming of Applications with Different Numbers of Computing Nodes. 91
6.2 The 2-pass MapReduce Accumulation 92
6.3 Running Times of Each Accumulative Programs 98
6.4 Relative Speedup Calculated with Respect to The Result on 8 CPUs 99
6.5 Architecture of Our GTA Library 108
6.6 Automaton of GTA Fusion 109
6.7 Finite Predicate 110
6.8 Execution Time of GTA Programs on Single CPU 111
6.9 Execution Times of GTA-Knapsack Program for Different W (64 CPUs) . . 112
6.10 Speedup of GTA-Knapsack on Spark Clusters 113
6.11 Execution Times of GTA-MSS Programs on Spark Cluster (64 CPUs) . . . 114
6.12 Execution Times of GTA Vitirbi Programs on Spark Cluster ((64 CPUs)) . . 115

List of Tables

3.1 Some Predefined Generators, Testers, and Aggregators 40

5.1 Reduction of subgraphs of *facebook-sg* 76
5.2 Evaluation of *Grow* on subgraphs of *facebook-sg*. 79
5.3 Comparison of properties of generated 3-trees and the original input graphs. 80
5.4 Small data sets (subgraphs of *facebook-sg*) 83
5.5 Comparison of results of our algorithm and [119](on data sets of Table 5.4) 84
5.6 Evaluation on big data sets (subgraphs of *facebook-sg*) 84

6.1 Execution Time (second) and Relative Speedup w.r.t. 2 Nodes 91
6.2 An Running Example for Simulating the Accumulation Procedure 96
6.3 Data Sets for Evaluating Examples on Hadoop Clusters with Different Number of Working Nodes 97
6.4 Large Data Sets for Comparing Two Kinds of Implementation: with/without Using our Framework 98
6.5 In Comparison of Length of Code and Performance to Vanilla Hadoop Programs (Using the Data Sets Listed in Table 6.4) 100
6.6 Comparison of Naive and GTA Knapsack Programs 111

Chapter 1

Introduction

1.1 Background

Nowadays, efficiently processing and analyzing terabytes of data from the Internet highly relies on parallel computing. However, distributed/parallel programming is generally considered as a difficult task, especially when the targeting environment is a large cluster of computers. For example, programs written in conformation with many widely used parallel programming models like Message Passing Interface (MPI [105]), Threading Building Blocks (TBB [73]), or multi-threaded languages like Java and C#, can be extremely difficult to understand and debug.

Recently, high level parallel programming models and frameworks such as MapReduce [43] and Pregel [90], have become popular in data-intensive computing. They provide high-level programming models to simplify the parallel programming. For example, MapReduce is based on two specialized algorithm skeletons, i.e., the *Map* and *Reduce* functions. Programmers only need to implement their algorithms in terms of *Map* and *Reduce*; the MapReduce framework then automatically execute user-programs in parallel. MapReduce and MapReduce-like frameworks are widely used in both industry and academic research. In particular, Hadoop [5] is a famous open-source implementation of MapReduce. Spark [133] is a fast in-memory MapReduce-like framework, which is implemented in the JVM based functional language Scala [109].

Despite the success of MapReduce-like programming models, developing efficient MapReduce programs is still a challenge. The main difficulties are considered as follows.

Firstly, the programming model is limited to a few specified basic skeletons (two functions in the case of MapReduce). The restriction on the expressiveness of such skeletons affects the programmability a lot. For example, implementing many algorithms in MapReduce, even ones have already been developed in Pthreads or MPI, still need a lot of efforts because of the restrictions of MapReduce programming model.

Secondly, there is a big gap between the programming model and the algorithms of individual problems. Taking the MapReduce as an example again, using *Map* and *Reduce* is not a natural way to describe the problem's specification. MapReduce Programming model requires programmers to develop divide & conquer algorithms that must fit its massive parallel execution manner. Particularly, MapReduce is not easy to use for many problems on graphs, therefore several particular frameworks especially for graph analysis are developed [56, 90, 118]. Moreover, the programming interface of MapReduce is considered being low-level and rigid [50, 87, 111, 133]; programming with it is difficult for many users.

Thirdly, optimization is difficult for certain programs. Measuring and adjusting the performance of MapReduce programs usually needs lots of efforts, not only because of the complex software/hardware environments (clusters containing thousands of computers) but also because of the large amount of input data. The static analysis and optimization is also not easy because of the in-black-box processes of MapReduce.

Skeletal parallel programming based on *algorithmic skeletons* [9, 44, 82, 112] gives a systematic way to construct or derive efficient parallel programs using generic and expressive program patterns or structures, a.k.a, *skeletons*. Algorithms can be represented as a sequence of skeletons or compositions of skeletons. Optimizations can be done by transforming an inefficient representation (program) to an efficient one by using *algorithmic skeletons* as the underlying building blocks to represent the programs [2, 33, 41, 96, 101]. However, the studies on skeletal parallel programming do not provide optimal solutions for the particular domain of data intensive parallel processing. Instead, they are more like interesting in theoretic discussions but do not have much practical use, as Murray Cole said, skeletal parallel programming still "*remains absent from mainstream practice*" [38].

This thesis is a study on constructing a high-level programming framework for resolving the above problems using a calculational approach. The objective is to introduce a simple, expressive, and flexible programming interface for representing algorithms in a structural way, and to enable the underlying optimizing mechanism of the framework to automatically transform the high-level structural programs into efficient and scalable parallel programs.

Our framework is suitable not only for dealing with problems on lists or sets (like MapReduce does), but also for dealing with many important optimization problems on graphs.

1.2 High-Level Parallel Programming Models

High-level parallel programming approaches, e.g., skeletal programming [34] and MapReduce [43], tend to rely on a *relatively* high level of abstraction of parallelism and implementation. High-level parallel programming frameworks can help users get rid of hard work on implementing miscellaneous details such as spawning a thread or maintaining a critical section between several threads.

Skeletal Parallel Programming. Skeletal parallel programming is a well-known high-level parallel programming approach that has been well-documented in the literatures [4, 7, 34, 38, 61, 112]. It observes that many parallel algorithms can be characterized and classified by their adherence to one or more generic patterns of computation and interaction. Such generic patterns are called *algorithmic skeletons* which can be abstracted and provided as a programmer's toolkit [27, 34, 95]. A program runs on machines with different architectures can be represented as a unique specification by skeletons that transcend architectural variations. The underlying implementations of these skeletons can be specialized to address performance issues in different hardware/software environments.

Algorithmic skeletons are polymorphic higher-order functions representing common parallelization patterns, and they are implemented in parallel [34]. Users can use such skeletons to compose parallel programs without considering low-level parallelism or implementation. Algorithmic skeletons can be used as building blocks of parallel and distributed applications by integrating them into a sequential language. The algorithmic skeletons are in two categories: task-parallelism skeletons and data-parallelism skeletons. Note that in this thesis, when we mention *algorithmic skeletons*, we are only talking about data-parallelism skeletons.

For data-intensive computing, usually, the parallel programs are firstly developed and tested with *small* data on a single machine or a *small* computer-cluster, and then deployed onto computer clusters which may contain tens of thousands of computers and process hundreds of thousands of terabytes data. Parallel programs are required to have good scalability and

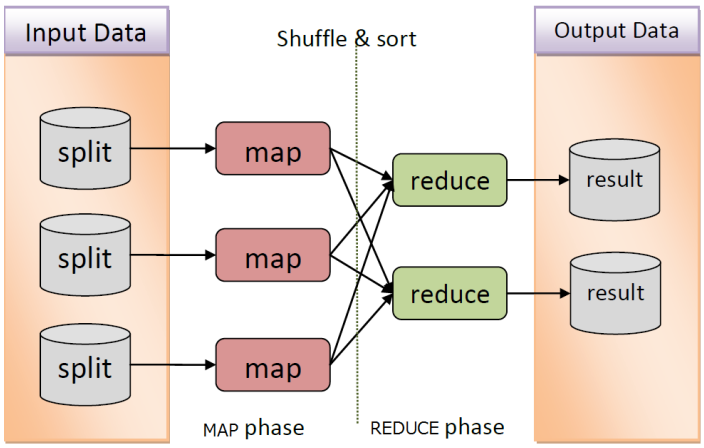


Fig. 1.1 Parallel Data-processing Flow of MapReduce

can be easily debugged, tested and tuned by using small instances of input.

MapReduce. MapReduce [43] is a programming model for data-intensive distributed parallel computing. It becomes now de facto standard for large scale data analysis and has emerged as one of the most widely used platforms for data-intensive distributed parallel computing.

Many algorithms can be represented by using two functions *Map* and *Reduce* in MapReduce programming model. However, MapReduce model is still considered being too low-level and rigid [50, 87, 111, 133], thus many higher level frameworks are built upon it. Sawzall [111], Pig Latin [50], DryadLINQ [132], and Hive [127] offer high-level languages with SQL-like syntax for Map-Reduce-like environments.

Automatic optimization of MapReduce programs is an important problem and studied by [63, 75]. These studies more or less using the traditional database query optimization approaches. For example, Pig [50] has query optimizers for generating logic plan, physic plan and also MapReduce-execution plan. Manimal [75] is a framework which can analyze MapReduce programs (Java code) and apply appropriate optimizations. Usually, these optimizations can achieve better performance but do not change the algorithmic complexity. However, [63, 75] are either ad hoc or domain-specific optimizations. They are quite difficult to be ported to another programming environment. Domain-independent fusion optimization is still necessary for different domain-problems. As an important part, this thesis studies algorithmic approaches to automatically constructing and optimizing MapReduce programs.

1.3 Automatic Parallelization and Optimization by Program Calculation

Program calculation [10, 15] (also called calculational programming) is a style for achieving transformational program development. In program calculation, we derive efficient programs by a set of program transformation laws. For example, the map function is a higher-order function that is defined as follows:

$$\begin{array}{lcl} \mathsf{map}\ f\ [\,] & = & [\,] \\ \mathsf{map}\ f\ ([a] +\!\!+ xs) & = & [f(a)] +\!\!+ \mathsf{map}\ f\ xs. \end{array}$$

In the above definition, $[\,]$ denotes an empty list, $[a]$ denotes a singleton list that consists of an element a, and $+\!\!+$ denotes the concatenation of two lists. The map function takes two arguments: a function f and an input list, and it applies the function f to each element in the list. We also use $(\mathsf{map}\ f)$ to denote a map function parametrized by a f.

Given two differently parametrized map functions, for example, $(\mathsf{map\ sqrt})$ and $(\mathsf{map\ sqr})$ and an integer list $[1,2,3,4]$. $(\mathsf{map\ sqrt} \circ \mathsf{map\ sqr})\ [1,2,3,4]$ means we first compute the square of each element in the input list, then let the result, i.e, a new list $[1,4,9,16]$ be the input of $(\mathsf{map\ sqrt})$. Then $(\mathsf{map\ sqrt})$ computes all square roots of elements in the list and we get result $[1,2,3,4]$. The total work of the computation contains two loops: $(\mathsf{map\ sqr})$ consumes a list and outputs a list; $(\mathsf{map\ sqrt})$ consumes the output of first one and produces a list as its output.

On the other hand, we have the following theorem on map, named $\mathsf{map} - \mathsf{map}$ fusion theorem [15]. For any two functions f and g, the following equation holds [15]].

$$\mathsf{map}\ f \circ \mathsf{map}\ g = \mathsf{map}\ (f \circ g)$$

According to this theorem, from $(\mathsf{map\ sqrt} \circ \mathsf{map\ sqr})$, we can derive a new program: $\mathsf{map\ sqrt} \circ \mathsf{sqr}$. The total work of $\mathsf{map\ sqrt} \circ \mathsf{sqr}$ function is 4 times of invocation of the composed function $\mathsf{sqrt} \circ \mathsf{sqr}$ in one iteration, which is much more efficient than $(\mathsf{map\ sqrt} \circ \mathsf{map\ sqr})$ for large input data because the load time of large data is a crucial factor for performance. This theorem identifies a useful idiom in transformational program development and it provides a high-level program transformation rule for a high-level programming language

containing map as a primitive construct.

Program calculation is a special form of program transformation, which is based on high-level program transformation rules, namely calculational laws. It reduces the difficulty of writing efficient algorithms. Users just need to represent computations in a direct (even naive) way, and we can apply calculational laws to generate efficient programs, under certain necessary conditions.

To be more specific, in our framework, programmers can represent their algorithm in a natural way simply by writing structural recursions on particular data structures. Structural recursion is a fundamental part of the definition of functions in type theory, and also in functional programming languages. Structural recursions on lists and trees are used to construct parallel programs which are studied in [13, 36, 58, 68, 92]. Structural recursions act as computation specifications of users' problems. Then our framework transforms specifications to efficient parallel programs by the calculation rules deterministically. For example, the following transformation indicates how we transform a user-program (sqr_sum function in pure functional style) to a MapReduce version parallel-sqr_sum.

The square_sum function is defined as following structural recursion form:

$$\begin{array}{lcl} \mathsf{sqr\_sum}\ [] & = & 0 \\ \mathsf{sqr\_sum}\ [a] & = & \mathsf{sqr}\ a \\ \mathsf{sqr\_sum}\ (x +\!\!+ y) & = & \mathsf{sqr\_sum}\ x + \mathsf{sqr\_sum}\ y. \end{array}$$

Here, users give a function sqr and an associative binary operator $+$ for sqr_sum. Our framework then generates a MapReduce program $\mathsf{sqr\_sum}_{\mathsf{MapRed}}$ which is a parallel version of sqr_sum.

$$\mathsf{sqr\_sum}_{\mathsf{MapRed}} = \mathsf{reduce}\ (+) \circ (\mathsf{map}\ \mathsf{sqr})$$

There have been many studies on formalizing effective calculational laws [11, 12, 39, 40, 53, 102, 116], deriving nontrivial algorithms [14, 15, 17, 18, 26, 121], and automating calculations [42]. Unlike the traditional *fold-unfold* approach [107] to program transformation on arbitrary programs, the calculational approach imposes restrictions on program structures, resulting in some suitable calculational forms such as *homomorphisms* and *mutumorphisms* that enjoy a collection of generic algebraic laws for program manipulation [110].

In such transformational computations, parallelism is not a part of the problem specifica-

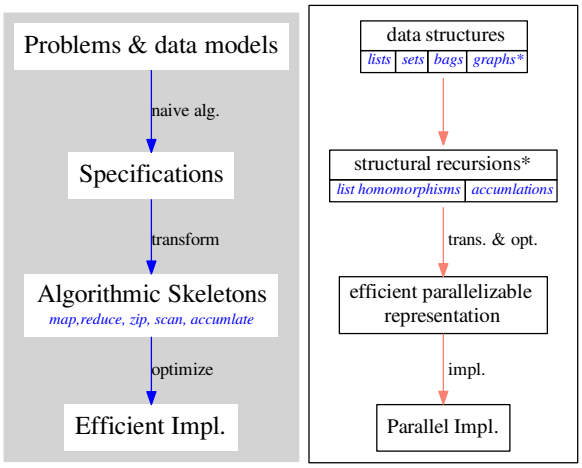


Fig. 1.2 Overview of Our Calculational Framework for High-level Parallel Programming

tion. Users just need to consider how to represent their problems/algorithms in the form of *structural recursions*. The whole picture of the calculational approach to parallel programming for data intensive processing is summarized in Figure1.2. In our framework, the programming interface is defined as structural recursion functions on lists. The framework will transform the specifications to intermediate representations (by using algorithmic skeletons) which represent efficient parallel algorithms. Finally, intermediate representations are implemented for different hardware environments.

For optimization problems on graphs, we transform graphs to a special form of trees (tree decompositions of graphs) and then trees are represented by lists, so that we can apply a similar parallelization approach to such lists.

1.4 Contributions and Organization of the Thesis

The main contribution of this thesis is a high-level parallel programming framework for data-intensive computing. Our framework can generate efficient parallel programs for processing large data sets. In other words, the optimization mechanisms of this framework are particularly suitable for the operations performed on large data sets, and only work with data-parallelism skeletons like map, reduce, or scan.

We have developed several libraries [85–87, 129] that support efficient parallel processing over various data structures such as lists, sets, trees and graphs. Generally, the framework provides a simple programming interface that parallelism is not user-concerned. It can perform light-weight program transformations to produce efficient and scalable parallel programs. Moreover, this programming framework enforces deterministic semantics that simplifies composing, optimizing, porting, reasoning about, debugging, and testing parallel programs. Finally, for a class of NP-hard optimization problems on graphs, we provide programming methodologies that allow users programming in an algorithmic and easy way to obtain exact or approximate solutions.

The organization of this thesis is as follows. Chapter 1 is the introduction of this thesis. The other chapters are grouped into three parts. In the first part (Chapters 2 and 3), an approach to resolving problems of the data model list is introduced. In Chapter 2, algebras for various data structures and homomorphisms on the algebras are introduced. Our high-level parallel programming principle is based on the structures of data and algebras for such data structures. There are deep relationships between the algebraic data structures and the computations (i.e., the problem specifications as well as the algorithms) defined on them. We make use of the mathematical properties (e.g., associativity, commutativity, distributivity and fusibility of the operators inside the definitions of algebras) of these algebras to construct programs and also we can transform/optimize programs thanks to these properties. In Chapter 3, we introduce our calculational approaches to resolving programming problems that are based on the data model of lists (also for sets).

In the second part (Chapters 4 and 5), our contributions on resolving programming problems for some combinatorial optimization problems on graphs are introduced. In Chapter 4, we introduce our approach to parallel programming on graphs that can be represented as a special form of trees by which we can represent a large graph as a tree and applying efficient dynamic programming algorithms on such tree to resolve many NP-hard optimization problems [25, 114]. Tree decompositions act as a bridge that brings the unstructured graph data into our framework. In Chapter 5, we introduce our approximation approach to resolving optimization problems on graphs, which makes our framework practically useful.

Implementations, evaluations and conclusions are covered in the last part (Chapters 6 and 7). In Chapter 6, we explain the implementation of several libraries inside our framework. Finally, a summary of this thesis and an outline of future work are given in Chapter 7.

Chapter 2

Algebraic Data Structures and Program Calculation Laws

In this chapter, firstly, we will introduce notations, operators, and their properties used throughout the thesis. Then, we will introduce algebras, structural recursions on lists [13, 15, 37, 66, 68, 104], and trees [52, 54, 101, 103, 123], a.k.a, their homomorphisms and the calculational laws of them.

Structural recursion is a fundamental part of the definition of functions in type theory, as well as in functional programming languages. Structural recursion is algorithmic, which indicates the way of computing on algebraic data types. They can be understood and analyzed locally and easy to be mapped to parallel computations. There are several algebraic laws for optimizing the compositions of structural recursions, which has been shown in the theory of Constructive Algorithmics [10, 14, 120, 122].

Structural recursions discussed in this thesis are also called *homomorphisms* because their computation structures and their processing data structures are closely related to each other. We do not distinguish the phrase *structural recursion* and *homomorphism* in this thesis.

2.1 Basic Notations

In this section, we introduce necessary preliminaries and notations used in this thesis.

2.1.1 Functions

The notations we use to formally describe algorithms are based on the functional programming language Haskell [16].

Function application can be written without parentheses, i.e., $f\ a$ equals $f(a)$. Functions are curried and function application is left associative, thus, $f\ a\ b$ equals $(f\ a)\ b$. Function application has higher precedence than operators, so $f\ a \oplus b = (f\ a) \oplus b$.

Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\ \oplus)\ b = (\oplus\ b)\ a = (\oplus)\ a\ b$. The identity element of a binary operator $\oplus$ is represented by $\iota_{\oplus}$. We use the operators $\circ$ and $\vartriangle$ over functions: by definition, $(f \circ g)\ x = f\ (g\ x)$ and $(f \vartriangle g)\ x = (f\ x, g\ x)$. The maximum and minimum operators are denoted by arrows $\uparrow$ and $\downarrow$.

$$\begin{aligned} x \uparrow y &= x \geq y\ ?\ x\ :\ y \\ x \downarrow y &= x \leq y\ ?\ x\ :\ y \end{aligned}$$

Two binary operators $\ll$ and $\gg$ are defined by $a \ll b = a$, and $a \gg b = b$, respectively.

Definition 2.1 (Right Inverse). *A function $f^{\circ} : B \to A$ is said to be a right inverse of another function $f : A \to B$ if $f \circ f^{\circ} \circ f = f$ holds.*

2.1.2 Sets

A set is denoted by a pair of curly brackets, for example, a denotes a set that consists an element a, and $1,2,3$ denotes a set that consists of 1, 2, and 3. The empty set is denoted by $\emptyset$. All subset of A is denoted by 2^A that itself is a set. Basic operators on sets: $\cap$, $\cup$ and $\times$, are defined as: $A \cap B := \{a \mid a \in A \wedge a \in B\}$, $A \cup B := \{a \mid a \in A \vee a \in B\}$, and $A \times B := \{(a,b) \mid a \in A \wedge b \in B\}$. The size of a set S is denoted by $|S|$.

2.1.3 Graphs

Formally, a graph $G = (V,E)$ is a set of vertices V and a set of edges E formed by unordered pairs of vertices. Usually, we use n denoting $|V|$ and m for $|E|$. Unless especially mentioned, all graphs in this paper are assumed to be finite, simple and undirected. We also assume the graphs under consideration are connected, since otherwise, the techniques being discussed

here can be applied to find solutions for each connected component, which can then be easily combined into a solution for the entire graph. For a vertex $v \in V$,let $N_G(v) = \{u|(u,v) \in E\}$ be the (open) neighbors of v. The *closed neighborhood* of a vertex $v \in V$ in G is $N_G[v] := N_G(v) \cup \{v\}$. When there is no ambiguity, we use $N(v)$ instead of $N_G(v)$. We say $H = (W,F)$ is a subgraph of $G = (V,E)$, denoted $H \subseteq G$, if both $W \subseteq V$ and $F \subseteq E$. An induced subgraph is one that satisfies $(x,y) \in F$ for every pair $x,y \in W$ such that $(x,y) \in E$. We denote the induced subgraph of G with vertices $X \subseteq V$ as $G[X]$.

In graph theory, a tree $T(I,F)$ is an acyclic graph that $I \subset \mathbb{N} \cup \{0\}$, and any two vertices are connected by exactly one simple path. In a tree, the number of vertices are one plus the number of edges, i.e., $|I| = |F| + 1$.

2.2 Theory of Lists

Firstly, we will introduce an algebra of lists and then we explain a special form of structural recursion on list. At last we will introduce basic program calculation theorems on lists.

2.2.1 Algebraic Data Type of Lists

Lists are sequences of elements, we use the following algebra of [14] to represent them.

Definition 2.2 (Algebraic Data Type of List [14]). *The algebraic data type of lists is defined with two constructors: $++$ (concatenation) and $[\cdot]$ (singleton).*

$$\begin{array}{lcl} \textbf{data}\ \mathsf{List}\ \alpha & = & [\,] \\ & | & [\cdot]\ \alpha \\ & | & (\mathsf{List}\ \alpha) ++ (\mathsf{List}\ \alpha) \end{array}$$

Here, $[\cdot]\ \alpha$, or abbreviated as $[\alpha]$, represents a singleton list of element α. For two lists x and y, $x ++ y$ represents the concatenated list consisting of elements of x followed by those of y. For example, a list of three elements n_1, n_2, and n_3 is represented by $[n_1] ++ [n_2] ++ [n_3]$, and can also be abbreviated as $[n_1,n_2,n_3]$. The notation $\alpha : x = [\alpha] ++ x$ shows the head of a list is an element α and rest part is a list $[x]$. The operator $++$ has the associativity, a.k.a, for any lists x, y, and z, two concatenations $x ++ (y ++ z)$ and $(x ++ y) ++ z$ form the same list.

As an abstract data type, a list is a finite ordered collection of values, where the same value may occur more than once. A set is a finite unordered collection of values, where every element does not have duplication. A multiset or a bag is a special set that may contain multiple copies of a value.

2.2.2 List homomorphism

List homomorphism is a special form of structural recursion, which has a close relationship with parallel computing and they have been studied intensively [37, 60, 68].

Given a list, a list homomorphism on it is defined as follows.

Definition 2.3 (List homomorphism). *Function h is said to be a list homomorphism, if and only if there is a function f and an associative binary operator $\odot$ such that the function h is defined as follows.*

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

Note that $\iota_{\odot}$ is the unit of $\odot$. 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 $(\!|id, +|\!)$:

$$\begin{array}{lcl} \mathsf{sum}\ [\,] & = & 0 \\ \mathsf{sum}\ [x] & = & x \\ \mathsf{sum}\ (xs +\!\!+ ys) & = & \mathsf{sum}\ xs + \mathsf{sum}\ ys. \end{array}$$

When function h is a homomorphism, the computation of h on a list, which is a concatenation of two shorter ones, can be carried out by computing h on each piece in parallel and then combining the results.

Before introducing the homomorphism theorems, we should provide the definitions of leftwards function and rightwards function.

Definition 2.4 (Leftwards function). *Function h is leftwards if it is defined in the following form with function f and operator $\oplus$,*

$$\begin{array}{lcl} h\ [\,] & = & \iota_{\oplus} \\ h\ [x] & = & f\ x \\ h\ ([x] +\!\!+ xs) & = & x \oplus h\ xs. \end{array}$$

Definition 2.5 (Rightwards function). *Function h is rightwards if it is defined in the following form with function f and operator $\otimes$,*

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

Here, the binary operators $\oplus$ and $\otimes$ need not to be associative. Below are two well-known theorems for homomorphisms [53].

Theorem 2.1 (The First Homomorphism Theorem). *Any homomorphism can be written as the composition of a map and a reduce:*

$$(\!|f, \odot|\!) = \mathsf{reduce}\ (\odot) \circ \mathsf{map}\ f.$$

This theorem says that any list homomorphism can be decomposed to a *map* and a *reduce*, each of which is suitable for efficient parallel implementation.

Theorem 2.2 (The Third Homomorphism Theorem). *Function h can be described as a list homomorphism, if and only if it can be defined by both leftwards and rightwards functions. Formally, there exists an associative operator $\odot$ and a function f such that*

$$h = (\!|f, \odot|\!).$$

if and only if there exist f, $\oplus$, and $\otimes$ such that

$$\begin{array}{lcl} h\ ([a] +\!\!+ x) & = & a \oplus h\ x \\ h\ (x +\!\!+ [b]) & = & h\ x \otimes b. \end{array}$$

The third homomorphism theorem is an important and useful theorem for automatic derivation of list homomorphisms [53, 103], because it gives a necessary and sufficient condition

for the existence of a list homomorphism. As will be seen later, our parallelization algorithm is mainly based on the third homomorphism theorem.

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. 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.

2.2.3 Basic Parallel Skeletons On Lists

Map and Reduce. The skeletons map and reduce are two most commonly used algorithmic skeletons.

Definition 2.6 (Map). *For a given function f, the function of the form $(\!|[\cdot] \circ f, +\!\!+|\!)$ is a* map *function, and is written as* map f.

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

Definition 2.7 (Reduce). *The function of the form $(\!|id, \odot|\!)$ for some $\odot$ is a* reduce *function, and is written as* reduce $(\odot)$.

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

Zipwith and Scan. The skeleton zipwith is an extension of map, and scan is an extension of reduce.

Definition 2.8 (Zipwith). *For a given function f, the function* zipwith *takes two lists of the same length, and applies f to every pair of corresponding elements of the two lists to*

produce a new list.

$$\begin{array}{lcl} \mathsf{zipwith}\ f\ [\,]\ [\,] & = & [\,] \\ \mathsf{zipwith}\ f\ [a]\ [b] & = & [f\ a\ b] \\ \mathsf{zipwith}\ f\ (x +\!\!+ y)\ (u +\!\!+ v) & = & (\mathsf{zipwith}\ f\ x\ u) +\!\!+ (\mathsf{zipwith}\ f\ y\ v) \end{array}$$

Note that the length of x and u (also y and v) must be the same.

The skeleton zip is a specialization of zipwith to make a list of pairs.

$$\mathsf{zip}\ =\ \mathsf{zipwith}(\lambda\ x\ y.(x,y))$$

The skeleton scan holds all values generated in reducing a list by reduce (we assume that input is a non empty list).

Definition 2.9 (Scan).

$$\begin{array}{lcl} \mathsf{scan}\ (\oplus)\ [a] & = & [a] \\ \mathsf{scan}\ (\oplus)\ (x +\!\!+ y) & = & (\mathsf{scan}\ (\oplus)\ x) \oplus' (\mathsf{scan}\ (\oplus)\ y) \\ & & \textbf{where} \\ & & \quad sx \oplus' sy = sx +\!\!+ \mathsf{map}((\mathsf{reduce}(\gg)sx)\oplus)sy \end{array}$$

The function scan is very useful in algorithm design and is also a primitive operator in lots of parallel computations [19, 20, 59]. For example, lexical analysis, quick sort, and regular-expression matching can be implemented by using scan.

2.2.4 Basic Program Calculation Laws on Lists

Lemma 2.3 (Fusion law of list homomorphism[14]). *Let g and $(\!|f,\oplus|\!)$ be given. If there exists $\odot$ such that for any x and y the equation*

$$g(x \oplus y) = g\ x \odot g\ y$$

holds, then

$$g \circ (\!|f,\oplus|\!) = (\!|g \circ f,\odot|\!).$$

Proof. The lemma is proved by induction on the structure of lists. □

Lemma 2.4 (Map-map fusion [15]). *The following equation holds for any functions f and g.*

$$\mathsf{map}\ g \circ\ \mathsf{map}\ f = \mathsf{map}\ (g \circ f).$$

Proof. We can prove the lemma by substituting map g and map $f = (\!|[\cdot] \circ f, +\!\!+ |\!)$ to Lemma 2.3, for the function g and the homomorphism$(\!|f, \oplus|\!)$ in the lemma, respectively. □

This lemma is useful because we can fuse a sequence of map into only one map; we can write a program step by step with many maps, and we can get efficient program by fusing them by Lemma 2.4.

2.3 Theory of Trees

In this thesis, we mainly focus on two kinds of algebraic data structures: lists and trees. In this section, the theory of trees is briefly reviewed.

2.3.1 Algebraic Data Type of Trees

The algebraic data type of trees is given as follows.

Definition 2.10 (Algebra Data Type of Tree [14]). *The algebraic data type of trees is defined with two constructors* Leaf *for leaves and* Node *for internal nodes as follows:*

$$\begin{array}{lcl} \textbf{data}\ \mathsf{Tree}\ \alpha & = & \mathsf{Leaf}\,\alpha \\ & | & \mathsf{Node}\ \alpha\ [\mathsf{Tree}\ \alpha]. \end{array}$$

The Leaf is the constructor for leaves, and holds an element of type α. The Node is the constructor for internal nodes and takes two parameters: the value of the node (in type of α), a list of subtrees (in type of Tree α) In this thesis, we mainly deal with binary trees, in which internal nodes have at most two children.

Definition 2.11 (Algebra Data Type of Binary Tree [14]).

$$\begin{array}{lcl} \textbf{data}\ \mathsf{BTree}\ \alpha\ \beta & = & \mathsf{Leaf}\ \alpha \\ & | & \mathsf{BNode}(\mathsf{BTree}\ \alpha\ \beta)\ \beta\ (\mathsf{BTree}\ \alpha\ \beta) \end{array}$$

The BNode is the constructor for internal nodes and takes three parameters in this order: the left subtree, the value of the node, and the right subtree. The value of the node has type β.

2.3.2 Tree Homomorphisms

For binary trees, the following binary-tree homomorphism (or tree homomorphism for short) is given by [122, 124].

Definition 2.12 (Tree Homomorphism). *Given two functions k_l and k_n. A function h is called* tree homomorphism *(or simply homomorphism), if it is defined in the following form.*

$$\begin{array}{lcl} h\ (\mathsf{Leaf}\ a) & = & k_l\ a \\ h\ (\mathsf{BNode}\ l\ b\ r) & = & k_n\ (h\ l)\ b\ (h\ r) \end{array}$$

Many computations on trees can be specified in the form of tree homomorphism. For example, function $\mathsf{hight}_{\mathsf{btree}}$ that computes the hight of a binary tree can be defined as follows:

$$\begin{array}{lcl} \mathsf{hight}_{\mathsf{btree}}\ (\mathsf{Leaf}\ a) & = & 1 \\ \mathsf{hight}_{\mathsf{btree}}\ (\mathsf{BNode}\ l\ b\ r) & = & 1\ +\ (\mathsf{hight}_{\mathsf{btree}}\ l)\ \uparrow\ (\mathsf{hight}_{\mathsf{btree}}\ r). \end{array}$$

2.4 Related Work

Using list homomorphisms to develop divide-and-conquer paradigm are studied by [13, 37, 95, 125]. Using the three well-known homomorphism theorems for *systematic development of parallel programs* are studied by [37, 53, 60, 68, 71]. It has been shown in [65, 104] that homomorphisms can be automatically developed for solving various kinds of problems.

Matsuzaki et al. [93] has proposed the ternary-tree representation for binary trees, in which three special nodes are introduced for the flexibility of structures. Morihata et al. [103] developed automatic parallelization approach based on third homomorphism theorem and applied it on trees.

Chapter 3

Calculational Approach to Constructing Parallel Programs on Lists

In this chapter, the calculational approach to resolving programming problems that based on the data model of lists (also for sets) is introduced. The main contribution of this work is a novel programming model and its framework for systematic programming over MapReduce, based on theorems of list homomorphisms [13, 37, 65].

3.1 Functional Formalization of MapReduce

MapReduce can be seen as a special case of skeletal programming model that only use two skeletons: *Map* and *Reduce*. *Map* and *Reduce* skeletons are related to data-parallelism. This thesis uses the capital MAP and REDUCE to denote the Map and Reduce, in order to distinguish them from the higher order functions map and reduce of functional programming languages.

In the MapReduce programming model, parallel computations are represented in the paradigm of a parallel MAP processing followed by a REDUCE processing [1]. Figure 1.1 shows the typical data-processing flow of MapReduce.

MapReduce computation mainly consists of three phases: MAP, SHUFFLE & SORT, and

[1] The REDUCE processing can be done in parallel or sequentially.

REDUCE[2]. In the MAP phase, each input key-value pair is processed independently and a list of key-value pairs will be produced. Then in the SHUFFLE & SORT phase, the key-value pairs are grouped based on the key. Finally, in the REDUCE phase, the key-value pairs of the same key are processed to generate a result.

To make the discussion precise, we introduce a specification of the MapReduce programming model in a functional programming manner. Note that the specification in this thesis is based on that in [83] but is more detailed. In this model, users need to provide four functions to develop a MapReduce application. Among them, the f_{MAP} and f_{REDUCE} functions performs main computation.

- Function f_{MAP}.

  $$f_{\text{MAP}} :: (k_1, v_1) \to [(k_2, v_2)]$$

  This function is invoked during the MAP phase, and it takes a key-value pair and returns a list of intermediate key-value pairs.

- Function f_{SHUFFLE}.

  $$f_{\text{SHUFFLE}} :: k_2 \to k_3$$

  This function is invoked during the SHUFFLE&SORT phase, takes a key of an intermediate key-value pair, and generate a key with which the key-value pairs are grouped.

- Function f_{COMP}.

  $$f_{\text{COMP}} :: k_2 \to k_2 \to \{-1, 0, 1\}$$

  This function is invoked during the SHUFFLE&SORT phase, and compares two keys in sorting the values.

- Function f_{REDUCE}.

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

  This function is invoked during the REDUCE phase, and it takes a key and a list of values associated to the key and merges the values.

A functional specification of the MapReduce framework can be given as follows, which

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

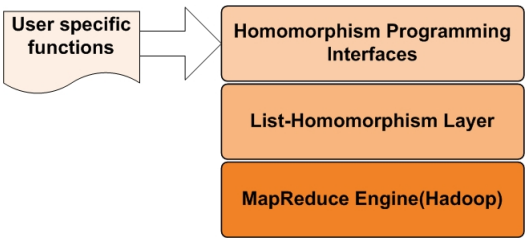


Fig. 3.1 System Overview of Screwdriver

Listing 3.1 Programming Interface for List Homomorphism

```
1 public abstract class ThirdHomomorphismTheorem<T,S> {
2     public abstract S fold(List<T> values);
3     public abstract List<T> unfold(S value);
4     ...
5 }
```

accepts four functions f_{MAP}, f_{SHUFFLE}, f_{COMP}, and f_{REDUCE} and transforms a set of key-value pairs to another set of key-value pairs.

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

Function $\mathsf{map_S}$ is a set version of the map function: i.e., it applies the input function to each element in the set. Function groupByKey takes a function f_{SHUFFLE} and a set of a list of key-value pairs, flattens the set, and groups the key-value pairs based on the new keys computed by f_{SHUFFLE}. The result type after groupByKey is $\{(k_3,\{k_2,v_2\})\}$. Function sortByKey takes a function f_{COMP} and a set of key-value pairs, and sorts the set into a list based on the order decided by f_{COMP}.

3.2 Implementing List Homomorphisms on MapReduce

In this section, how to use list homomorphism wrapping MapReduce is discussed. A library named *Screwdriver* have been implemented in our work [87], which is built on top of Hadoop, purely in Java. As shown in Figure 3.1, *Screwdriver* consists of three layers: the interface layer for easy parallel programming, the homomorphism layer for implementing homomorphism, and the base layer of the MapReduce engine (Hadoop).

The highest layer of Screwdriver provides a simple programming interface and generates a homomorphism based on the third homomorphism theorem. Users specify a pair of sequential functions instead of specifying a homomorphism directly: one for solving the problem itself and the other for an inverse of the problem. Consider the summing-up example again. A right inverse sum° of the function sum takes a value (the result of sum) and yields a singleton list whose element is the input value itself. The functional definition of sum° is: $\mathsf{sum}^\circ\ s = [s]$.

Listing 3.1 shows the Java programming interface provided in our framework, where users should write a program by inheriting the ThirdHomomorphismTheorem class. The function fold corresponds to the sequential function that solves the problem, and the function unfold corresponds to the sequential function that computes a right inverse. In a functional specification, the types of the two functions are $\mathsf{fold} :: [t_1] \to t_2$ and $\mathsf{unfold} :: t_2 \to [t_1]$.

To utilize the third homomorphism theorem, users are requested to confirm that the two functions satisfy the following conditions. Firstly, the function unfold should be a right inverse of the function fold. In other words, the equation $\mathsf{fold} \circ \mathsf{unfold} \circ \mathsf{fold} = \mathsf{fold}$ should hold. Secondly, for the fold function there should exist two operators $\ominus$ and $\oplus$ as stated in Theorem 2.2. A sufficient condition for this second requirement is that the following two equations hold respectively for any a and x.

$$\mathsf{fold}([a] +\!\!+ x) = \mathsf{fold}([a] +\!\!+ \mathsf{unfold}(\mathsf{fold}(x))) \tag{3.1}$$

$$\mathsf{fold}(x +\!\!+ [a]) = \mathsf{fold}(\mathsf{unfold}(\mathsf{fold}\ x) +\!\!+ [a]) \tag{3.2}$$

Note that we can use some tools (such as QuickCheck [32]) in practice to verify whether Equations **3.1** and **3.2** hold or not.

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

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

In the second layer, Screwdriver provides an efficient implementation of list homomorphisms over MapReduce. In particular, the implementation consists of two passes of MapReduce. Before we go to the details of implementation, we firstly discuss the parallel data structure for long lists or sets that we are dealing with.

3.2.1 Data Models of Lists

In shared memory environments, lists, sets or multisets are quite often used and have many kinds of implementations. Such as LinkedList, ArrayList and HashSet in Java collection library.

In the context of data-intensive computing, typically in MapReduce, the data model for lists and sets are file-based. Lists and sets are stored in the plain files and each element is a record in the file. Files are distributed in a distributed file system like Google File System (GFS) [51]. In the distributed file system, a large file will be split to smaller chunks. Sets can be easily implemented as a set of records because we do not need to consider the order of elements. However, the elements of a list has a fixed order that need to be carefully manipulated during processing. In a distributed computing environment, manipulating a global list among multiple computing nodes are quite different with that is under a shared memory environment.

We have two ways to manipulate a long list that distributed over multi-chunks. The first way is that we represent each element of a list as an $(index, value)$ pair where *index* is an integer indicating the position of the element. For example, a list $[a,b,c,d,e]$ may be represented as a set $\{(3,d),(1,b),(2,c),(0,a),(4,e)\}$. Note that the list can be restored from this set representation by sorting the elements by their indices. Such indexed pairs permit storing data in arbitrary order on the distributed file systems. In Screwdriver, we represent each element of a list as such $(index, value)$ pair.

In the second way, particularly in the environment of MapReduce, for simplicity of discussion, we suppose the input data are stored as a list of records in one binary file (could be very huge) in the distributed file system (DFS). Each record of the file represents an element of input list. If the size is larger than a threshold, the input file will be split to several splits by DFS (each split is called a chunk in the DFS) and distributed to several DataNodes, and the DFS knows the offsets of each splits [43, 51], which can be seen as the indices of the splits. When data are loaded to multiple mappers, users of MapReduce cannot control which mapper to load which splits. In order to keep the total order of computation, the output of each mapper must be associated with the offset of its input, so that when merging the results from mappers, the order can be carefully manipulated by making use of such offsets and the sorting function.

3.2.2 A MapReduce Algorithm for Implementing Homomorphisms

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

In the following, we summarize our two-pass implementation of homomorphism $(\!|f, \oplus|\!)$. Here, hom f $(\oplus)$ denotes a sequential version of $(\!|f, \oplus|\!)$, comp is a comparing function

defined over the *Int* type, and *const* is a constant value defined by the framework.

$$
\begin{aligned}
&\mathsf{hom}_{\mathrm{MR}} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \{(\mathit{Int}, \alpha)\} \to \beta \\
&\mathsf{hom}_{\mathrm{MR}}\ f\ (\oplus) = \mathsf{getValue} \circ \mathsf{MapReduce}\ ([\cdot])\ g_{\mathrm{SHUFFLE}}\ \mathit{comp}\ g_{\mathrm{REDUCE}} \\
&\qquad\qquad \circ \mathsf{MapReduce}\ ([\cdot])\ f_{\mathrm{SHUFFLE}}\ \mathit{comp}\ f_{\mathrm{REDUCE}}
\end{aligned}
$$

where

$$
\begin{aligned}
&f_{\mathrm{SHUFFLE}} :: \mathit{Int} \to \mathit{Int} \\
&f_{\mathrm{SHUFFLE}}\ k = k / \mathit{const} \\
&f_{\mathrm{REDUCE}} :: (\mathit{Int}, [\alpha]) \to (\mathit{Int}, \beta) \\
&f_{\mathrm{REDUCE}}\ (k, as) = (k, \mathsf{hom}\ f\ (\oplus)\ as) \\
&g_{\mathrm{SHUFFLE}} :: \mathit{Int} \to \mathit{Int} \\
&g_{\mathrm{SHUFFLE}}\ k = 1 \\
&g_{\mathrm{REDUCE}} :: (\mathit{Int}, [\beta]) \to (\mathit{Int}, \beta) \\
&g_{\mathrm{REDUCE}}\ (1, bs) = (1, \mathsf{hom}\ \mathsf{id}\ (\oplus)\ bs) \\
&\mathsf{getValue} :: \{(\mathit{Int}, \beta)\} \to \beta \\
&\mathsf{getValue}\ \{(1, b)\} = b
\end{aligned}
$$

First pass of MapReduce. The first pass of MapReduce divides the list into some sublists, and computes the result of the homomorphism for each sublist. In the MAP phase, we just wrap each key-value pair into a singleton list. Then in the SHUFFLE&SORT phase, we group the pairs so that the set-represented list is partitioned into some number of sublists and sort each grouped elements by their indices. Finally, we apply the homomorphism to each sublist in the REDUCE phase.

Second pass of MapReduce. The second pass of MapReduce computes the result of the whole list from the results of sublists given by the first pass of MapReduce. Firstly in the MAP phase, we do exactly as in the first pass. Then in the SHUFFLE&SORT phase, we collect the intermediate results into a single set and sort them by the their indices. Finally, we reduce the intermediate results using the associative operator of the homomorphism. By the getValue function, we picked the result of the homomorphism out from the set (of single value).

Implementation Issues. In terms of the parallelism, the number of the MAP tasks in the first pass is decided by the data splitting mechanism of Hadoop. For one split data of the

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

3.3 An Accumulative Computation Pattern on MapReduce

There are many problems that are difficult to be expressed in MapReduce model. As an example, consider the *elimSmallers* problem of eliminating all the *smaller* elements of a list to produce an ascending list (if an element is less than someone in the previous, it is *smaller*). For instance, given a list $[11,15,8,9,20,25,12,23]$, then 8, 9, 12 and 23 are smaller ones, and thus the result is $[11,15,20,25]$. A recursive function that solves this problem can be defined as follows, in Haskell [16].

$$
\begin{array}{lcl}
\mathit{elimSmallers}\ [\,]\ c & = & [\,] \\
\mathit{elimSmallers}\ (x:xs)\ c & = & (\textbf{if}\ x < c\ \textbf{then}\ [\,]\ \textbf{else}\ [x]) +\!\!+ \\
 & & \mathit{elimSmallers}\ xs\ (\textbf{if}\ x < c\ \textbf{then}\ c\ \textbf{else}\ x)
\end{array}
$$

In this function, recursively, numbers of input are compared with an accumulative parameter c (with initial value $-\infty$). The accumulative parameter c always holds the current maximum value and is used in the next recursion. If the head element is larger than c then it is appended to the tail of the result, and otherwise dropped. In functional programming, such kind of computation pattern with accumulative parameters is called accumulative computation [69, 72].

The recursive function *elimSmallers* clearly describes the computation (in $O(n)$ work, n is the length of input), but it cannot be easily mapped to MapReduce, because in the recursive function *elimSmallers*, every inner step of the recursion relies on the current maximum value, which is computed at the outer step. Such kind of recursive functions do not correspond to a simple divide-and-conquer algorithm. Developing an $O(n)$ work MapReduce algorithm for *elimSmallers* needs to resolve such computational dependency and avoid unnecessary and expensive I/O, which is not easy for many programmers. There are many applications (e.g., the *prefix-sum/scan* related problems [20]) in similar computation pattern

with *elimSmallers* and thus are also difficult to be resolved in MapReduce.

In this section, we propose a new accumulation framework for simplifying MapReduce programming on accumulative computations that cannot be expressed by using *map* and *reduce* functions in only one iteration of MapReduce processing. The programming interface [3] is designed to help users define recursive functions in the accumulative form, and then efficient and scalable MapReduce solutions can be automatically gained.

3.3.1 Accumulative Computations

Accumulative computation [69] plays an important role in describing a computation on an ordered list from left or right, when a later computation depends more or less on this computation. The data dependency can be captured by using an accumulative parameter that holds and delivers some information through the whole computation.

The accumulate skeleton [69] abstracts a typical pattern of *recursive functions* with an accumulative parameter, which can be defined as a function h in the following form.

$$\begin{aligned} h\ [\,]\ c &= g\ c \\ h\ (x:xs)\ c &= p\ (x,c) \oplus h\ xs\ (c \otimes q\ x) \end{aligned}$$

This definition provides a natural way to describe computations with data dependencies and can be understood as follows.

- If the input list is empty, the result is computed by applying some function g to accumulative parameter c.
- If the input list is not empty and its head and tail parts are x and xs respectively, then the result is generated by combining the following two values using some binary operator $\oplus$: the result of applying p to x (head value) and c (the accumulative parameter), and the recursive call of h to xs (the rest part of the input list) with its accumulative parameter updated to $c \otimes q\ x$.

Because h is uniquely defined by g, p, $\oplus$, q, and $\otimes$, so we write h with special parentheses $[\![\ ]\!]$ as:

[3] The accumulate skeleton has been implemented using MPI that is more flexible in programming model and does not have same constants as MapReduce has.

$$h\ xs\ c = [\![g, (p,\oplus), (q,\otimes)]\!]\ xs\ c.$$

Note that $(p,\oplus)$ and $(q,\otimes)$ correspond to two basic recursive forms foldr and foldl [53] respectively. The *elimSmallers* discussed in the introduction can be also written as follows.

$$\begin{array}{l} elimSmallers\ xs\ c = [\![g, (p,\oplus), (q,\otimes)]\!]\ xs\ c \\ \textbf{where}\ g\ c = [\,] \\ \qquad p\ (x,c) = \textbf{if}\ x < c\ \textbf{then}\ [\,]\ \textbf{else}\ [x] \\ \qquad \oplus = +\!+ \\ \qquad q\ =\ id \\ \qquad \otimes = \uparrow \end{array}$$

Since the function h in the above form represents the most natural recursive definition on lists with a single accumulative parameter, it is general enough to capture many algorithms [69] as seen below.

Scan. Recall to the Definition 2.9. Given a list $[x_1,x_2,x_3,x_4]$ and an associative binary operator $\odot$ with an identity element $\iota_\odot$, a function scan computes all its prefix sums yielding:

$$[\iota_\odot,\ x_1,\ x_1 \odot x_2,\ x_1 \odot x_2 \odot x_3,\ x_1 \odot x_2 \odot x_3 \odot x_4].$$

(Note that the head element of output is the unit of $\odot$, a bit different with scan in Definition 2.9). The function scan can be defined in terms of accumulate by giving an initial value $\iota_\odot$ to the accumulative parameter c:

$$\mathsf{scan}\ [\,]\ c = [\cdot]\ c$$

$$\mathsf{scan}\ (x : xs)\ c = ([\cdot] \circ \mathsf{snd})(x,c) +\!+ \mathsf{scan}\ xs\ (c \odot\ (\mathsf{id}\ x)).$$

The scan in Definition 2.9 only consider about non-empty list as input. Here, if the input is an empty list then scan will return a singleton list containing the unit of $\odot$.

Line-of-Sight Problem. The well known line-of-sight problem [20] is that given a terrain map in the form of a grid of altitudes and an observation point, find which points are visible along a ray originating at the observation point. For instance, we use a pair (d,a) to

represent a point, where a is the altitude of the point and d is its distance from the observation point. The function $\mathsf{angle}\,(d,a) = a/d$ computes the tangent of an angle. If the list is $[(1,1),(2,5),(3,2),(4,10)]$, then the point (3,2) is invisible. The function *los* [74] solves a simplified line-of-sight problem which counts the number of visible points.

$$
\begin{array}{l}
los\ xs\ c = [\![g, (p,+), (q,\uparrow)]\!]\ xs\ c \\
\quad \textbf{where}\ g\ c = 0 \\
\qquad p\ (x,c) = \textbf{if}\ c \leq angle\ x\ \textbf{then}\ 1\ \textbf{else}\ 0 \\
\qquad q\ x = angle\ x
\end{array}
$$

Maximum Prefix Sum Problem. Intuitively, the maximum prefix sum problem is to compute the maximum sum of all the prefixes of a list. Given a list $[\underline{3,-4,9,2},-6]$ the maximum of the prefix sums is 10, to which the underlined prefix corresponds. We can define a function *mps* that solves this, in terms of $\mathsf{accumulate}$.

$$mps\ xs\ c = [\![id, (snd,\uparrow), (id,+)]\!]\ xs\ c$$

Tag Matching Problem. The tag matching problem is to check whether the tags are well matched or not in a document, e.g., an XML file. There is an accumulative function tagmatch introduced by [74] for the tag matching problem.

$$
\begin{array}{l}
tagmatch\ xs\ cs = [\![\mathsf{isEmpty}, (p,\wedge), (q,\otimes)]\!]\ xs\ cs \\
\quad \textbf{where} \\
\quad p\ (x,cs) \\
\qquad = \textbf{if}\ \mathsf{isOpen}\ x\ \textbf{then}\ True \\
\qquad\quad \textbf{else if}\ \mathsf{isClose}\ x\ \textbf{then} \\
\qquad\quad notEmpty\ cs\ \wedge\ match\ x\ (top\ cs) \\
\qquad\quad \textbf{else}\ True \\
\quad q\ x = \textbf{if}\ \mathsf{isOpen}\ x\ \textbf{then}\ ([x],1,0) \\
\qquad\quad \textbf{else if}\ \mathsf{isClose}\ x\ \textbf{then}\ ([\,],0,1) \\
\qquad\quad \textbf{else}\ ([\,],0,0) \\
\quad (s_1,n_1,m_1) \otimes (s_2,n_2,m_2) \\
\qquad = \textbf{if}\ n_1 \leq m_2\ \textbf{then}\ (s_2, n_2, m_1 + m_2 - n_1)
\end{array}
$$

Listing 3.2 Scan Representation

```
1 public class Scan<T> {
2     public AssociativeBinaryOP<T> oplus;
3
4     public Scan(AssociativeBinaryOP<T> op) {
5         this.oplus = op;
6     }
7 }
```

else $(s_2 +\!\!+ drop\ m_2\ s_1, n_1 + n_2 - m_2, m_1)$

3.3.2 The Programming Interface for Accumulation

We provide two parallel skeletons scan and accumulate in our framework. These two skeletons and related binary operators are represented as Java classes, in the object-oriented style.

Scan Interface. Listing 3.2 shows the representation Java class for scan. Users need to define the associative binary operator to create an instance of the scan computation. There is an example of an associative binary operator `Plus` in Listing 3.3, which returns the sum of two integers. The `evalute` method takes two arguments and returns one value. The `id` method returns the identity element. To execute a scan on MapReduce cluster, users need to write the client codes like Listing 3.3. The Java class `ScanExample` extends `ScanMRHelper` and overrides the method `createScanIns` which creates an instance of the scan computation. In the `main` function, the method `runScanMR` which takes two arguments — one is an instance of scan, and the other is the `args` (the input, output paths given by users) from `main`, — will execute the scan computation on the Hadoop cluster.

Accumulation Interface. An accumulate can be defined by implementing the abstract class `Accumulation` (as shown in Listing 3.4). There are five functions/operators according to the definition of the accumulate, and an accumulative parameter `c`. The Java class `MapReduceExample` (Listing 3.5) which extends `MRAccHelper` shows how to write the client code. Similarly to scan, the method `createAccuIns` needs to be override, in which an instance of accumulate is created. In the `main` function, the `runAccMR`

Listing 3.3 An Example of Using the Scan Programming Interface

```
public class ScanExample extends MRScanHelper<Int> {
    // type Int is a wrapping class for Java int
    public class Plus extends AssociativeBinaryOP<Int> {
        public Int evaluate(Int a, Int b) {
            return new Int(a.val + b.val);
        }
        public Int id() {
            return new Int(0);
        }
    }

    @Override
    public void createScanIns() {
        // instance an scan computation
        this.scan = new Scan(new Plus());
    }

    public static void main(String[] args) throws Exception {
        // Create and run on MapReduce
        int res = runScanMR(new ScanExample(), args);
        System.exit(res);
    }
}
```

Listing 3.4 Accumulation Representation

```
public abstract class Accumulation<T0, T1, T2> {
    public T1 c;
    public UnaryFunction<T1, T2> g;
    public AssociativeBinaryOP<T2> oplus;
    public AssociativeBinaryOP<T1> otimes;
    public BinaryOperator<T0, T1, T2> p;
    public UnaryFunction<T0, T1> q;
}
```

Listing 3.5 An Example of Using the Accumulation Programming Interface

```
1 public class MapReduceExample extends MRAccHelper<Int,Int,
    IntList> {
2     public void createAccuIns( ) {
3         //instance an accumulate computation
4         this.accumulate = new ElimSmallers(new Int(50));
5     }
6     public static void main(String[] args)
7                                 throws Exception {
8         //Create and run on MapReduce
9         int res = runAccMR( new AExample(), args);
10        System.exit(res);
11        }
12    }
13 }
```

method should be invoked to execute the accumulate. The Listing 3.6 shows an example to define the *elimSmallers* computation.

3.4 A Generate-Test-Aggregate Parallel Programming Pattern on MapReduce

The Generate-Test-Aggregate (GTA) programming pattern and powerful fusion optimization [45, 46] have been proposed as an algorithmic way to synthesize MapReduce programs from naive specifications (GTA programs) in the following form.

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

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.

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:

Listing 3.6 Difination of Accumulation for ElimSmallers

```
1 public class ElimSmallers extends Accumulation<Int, Int, IntList> {
2
3     public ElimSmallers(Int x) {
4         c = x;// new Int(50);
5
6         oplus = new AccociBinaryOP<IntList>() {
7             @Override
8             public IntList evaluate(IntList left, IntList right) {
9                 if (left == null || left.get() == null)
10                    left = new IntList(new ArrayList<Int>());
11                if (right != null && right.get() != null
12                        && right.get().size() > 0) {
13                    left.get().addAll(right.get());
14                }
15                return left;
16            }
17            @Override
18            public IntList id() {
19                return new IntList();
20            }
21        };
22
23        otimes = new AccociBinaryOP<Int>() {
24            @Override
25            public Int evaluate(Int left, Int right) {
26                if (right.get() < left.get())
27                    return left;
28                else
29                    return right;
30            }
31            @Override
32            public Int id() { // id * x = x
33                return new Int(Integer.MIN_VALUE);
34            }
35        };
36
37        g = new UnaryFunction<Int, IntList>() {
38            @Override
39            public IntList evaluate(Int obj) {
40                return new IntList();
41            }
42        };
43
44        p = new BinaryOperator<Int, Int, IntList>() {
45            @Override
46            public IntList evaluate(Int x, Int c) {
47                ArrayList<Int> val = new ArrayList<Int>();
48                if (x.get() < c.get()) {
49                    return null;
50                } else {
51                    val.add(x);
52                    return new IntList(val);
53                }
54            }
55        };
56
57        q = new UnaryFunction<Int, Int>() {
58            @Override
59            public Int evaluate(Int da) {
60                return da;
61            }
62        };
63
64    }
65 }
```

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

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 [45, 46] 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 [4].

To get a better grasp of the meaning of all this, let us review several important concepts.

Definition 3.1 (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 an identity element $\iota_\oplus$,*
- *$\otimes$ is associative operator with an identity element $\iota_\otimes$ and $\otimes$ distributes over $\oplus$,*
- *$\iota_\oplus$ is a zero of $\otimes$.*

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

[4] The efficient MapReduce program only produces $O(n)$ intermediate data.

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

Definition 3.2 (Semiring homomorphism). *Given an arbitrary semiring* $(S,\oplus,\otimes)$ *and a function* $f :: A \to S$*, the function* $\mathsf{shom} :: \wr[A]\int \to S$ *is a semiring homomorphism from* $(\wr[A]\int,\uplus,\times_{+\!\!+})$ *to* $(S,\oplus,\otimes)$*, iff the following hold.*

$$\begin{aligned}
\mathsf{shom}\ (x \uplus y) &= \mathsf{shom}\ x \oplus\ \mathsf{shom}\ y \\
\mathsf{shom}\ (x \times_{+\!\!+} y) &= \mathsf{shom}\ x \otimes\ \mathsf{shom}\ y \\
\mathsf{shom}\ \wr\,[a]\int &= f\ a \\
\mathsf{shom}\ \wr\int &= \imath_{\oplus} \\
\mathsf{shom}\ \wr\,[]\int &= \imath_{\otimes}
\end{aligned}$$

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

Since a semiring homomorphism consumes a bag of lists and produces a value, it is used as an aggregator in the GTA program. An example of semiring homomorphisms is the aggregator maxsum f using the max-plus semiring $(\mathbb{Z},\uparrow,+)$ to find the maximum among f-weighted sums of lists in a given bag:

$$\begin{aligned}
\mathsf{maxsum}\ f\ (x \uplus y) &= \mathsf{maxsum}\ x \uparrow \mathsf{maxsum}\ y \\
\mathsf{maxsum}\ f\ (x \times_{+\!\!+} y) &= \mathsf{maxsum}\ x + \mathsf{maxsum}\ y \\
\mathsf{maxsum}\ f\ \wr\,[a]\int &= f\ a \\
\mathsf{maxsum}\ f\ \wr\int &= -\infty \\
\mathsf{maxsum}\ f\ \wr\,[]\int &= 0
\end{aligned}$$

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

Now, let us introduce a class of generators that have good fusibility with semiring homomorphisms.

Definition 3.3 (Semiring Polymorphic Generator [45]). *A polymorphic function over semirings* $(S,\oplus,\otimes)$

$$\mathit{generator}_{\oplus,\otimes} :: (A \to S) \to [A] \to S$$

is called a semiring polymorphic generator.

The semiring polymorphic generator is parameterized by semirings, and given different semirings it does different computations. Particularly, using the semiring $(\wr[A]\int, \uplus, \times_{+\!\!+})$ of bags of lists, the function $\mathsf{generator}_{\uplus,\times_{+\!\!+}}\ (\lambda a.\wr[a]\int) :: [A] \to \wr[A]\int$ is a generator that can be used in the GTA program. For example, abstracting the semiring in the generator sublists, we get $\mathit{sublists} = \mathsf{sublists}'_{\uplus,\times_{+\!\!+}}\ (\lambda a.\wr[a]\int)$ where

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

Moreover, we have the following powerful theorem for fusing such a generator with an aggregator (semiring homomorphism).

Theorem 3.1 (Semiring Fusion [45]). *Given a semiring polymorphic function* $\mathsf{generator}_{\oplus,\otimes} :: (A \to S) \to [A] \to S$ *and a semiring homomorphism* $(\!|f,\oplus,\otimes|\!)$ *to* $(S,\oplus,\otimes)$*, the following holds.*

$$(\!|f,\oplus,\otimes|\!) \circ \mathsf{generator}_{\uplus,\times_{+\!\!+}}\ (\lambda a.\wr[a]\int) = \mathsf{generator}_{\oplus,\otimes}\ f$$

The left-hand side of the equation is a GTA program (without any tester): the generator is written as $\mathsf{generator}_{\uplus,\times_{+\!\!+}}\ (\lambda a.\wr[a]\int)$, and the aggregator is written as $(\!|f,\oplus,\otimes|\!)$. In this program, an exponential number of intermediate lists are possibly generated by the generator and 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 computationally heavy operator $\times_{+\!\!+}$ and rather uses a (possibly) light operator $\otimes$. For example, the theorem says that the program $\mathsf{maxsum}\ (\lambda a.a) \circ \mathit{sublists}$ for computing the maximum of sums of all sublists of a certain list is equivalent to program $\mathsf{sublists}'_{\uparrow,+}\ (\lambda a.a)$. This is easily verified because the latter program computes the 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 3.4 (Homomorphic Tester [45]). *If a* test *is a filter with a predicate (defined by composing a function* ok *and a list homomorphism* $(\![f,\odot]\!)$*, namely,* $\mathsf{test} = \mathsf{filter}\ (\mathsf{ok} \circ (\![f,\odot]\!))$*, 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 much total weights is a homomorphic tester:

$$\mathsf{removeInvalidSelection} = \mathsf{filter}\ ((\leq w) \circ (\!|\mathsf{getWeight}, +|\!)).$$

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.

The filter embedding works as follows:

Theorem 3.2 (Filter Embedding [45]). *Given a homomorphic tester* $\mathsf{filter}\ (\mathsf{ok} \circ (\!|f, \odot|\!))$ *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* $\mathsf{postprocess}_{ok}$ *such that the following holds.*

$$\{\!|f, \oplus, \otimes|\!\} \circ \mathsf{filter}\ (\mathsf{ok} \circ (\!|f, \odot|\!)) = \mathsf{postprocess}_{\mathsf{ok}} \circ \{\!|f^M, \oplus^M, \otimes^M|\!\}$$

This filter embedding is useful because we can remove the tester between the generator and aggregator so that we can fuse them by using the semiring fusion. Interested readers can read the paper [45, 46] for details.

According to the theory of GTA, i.e., the combination of filter embedding and semiring fusion, a GTA program consisting of those components can be eventually transformed to an efficient program $\mathsf{postprocess} \circ \mathsf{generator}_{\oplus^M, \otimes^M}\ f^M$. 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 costs only $O(n)$.

3.4.1 The GTA Programming Interface

Our library is implemented in Scala [109] and provides an easy-to-use programming interface for users to write GTA expressions using the GTA components i.e., generators, testers, aggregators. Shown as Figure 3.2, the library can automatically transform a user-specified GTA program to an efficient MapReduce program and execute it. The transformation has two phases: in the first phase, a user-specified GTA program is transformed to an instance of `MapReduceable` that is a Scala trait adapting the list homomorphism to the MapReduce, and its definition is shown in Listing 3.7. Then the `MapReduceable` is used by a MapRe-

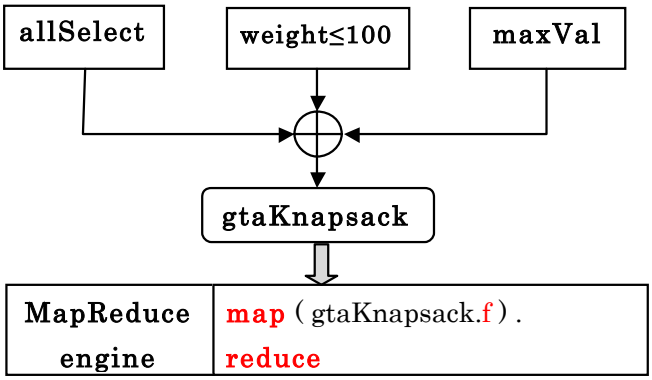


Fig. 3.2 The GTA Program Transformation

Listing 3.7 MapReduceable is the Scala trait of Almost-List-Homomorphism

```
1 trait MapReduceable[A,M,+R] extends ListHomomorphism[A,M]{
2   override def f(a:A):M
3   override def combine(l:M,r:M):M
4   def postProcess(a:M):R
5 }
```

duce driver-program that invokes the `f` (for mapping) method and `combine` (for reducing) method in MAP and REDUCE phase computing, respectively.

The Scala trait 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 output of the REDUCE processing as a final processing [5]. The MapReduceable can be used in any MapReduce engine or parallel frameworks that provides MapReduce style APIs. A fully scalable MapReduce program can be written simply by using the f in the MAP processing and combine in the REDUCE processing.

The library provides a top level class named `GTA` for wrapping the GTA programming environment. Every GTA expression must be defined inside the scope of the `GTA` class. The user should extend `GTA` to write his GTA program in terms of GTA expressions.

A GTA expression (in Scala) is written as:

$$\texttt{val gta} = \texttt{generate}(\ldots)\ \texttt{filter}(\ldots)\ \texttt{aggregate}(\ldots).$$

[5] Such an extended list homomorphism is called an *almost list homomorphism* [35, 67]

`val gta` is a GTA object (MapReduceable) that can be executed in parallel. For "(...)"s, the user should choose proper parameters respectively to the generator, tester, and aggregator. Here, to make this more readable, we give the formal definition of the GTA expression in extended Backus–Naur form (EBNF).

$$
\begin{array}{lcl}
\textit{expr} & = & \textit{genTerm},\ \textit{spl},\ \{\ \textit{filtTerm},\ \textit{spl}\ \},\ \textit{aggTerm}\ ; \\
\textit{genTerm} & = & \text{'}\textbf{generate}\text{'},\ \text{'('},\ \texttt{GeneratorCreater},\ \text{')'}\ ; \\
\textit{filtTerm} & = & \text{'}\textbf{filter}\text{'},\ \text{'('},\ \texttt{Predicate},\ \text{')'}\ ; \\
\textit{aggTerm} & = & \text{'}\textbf{aggregate}\text{'},\ \text{'('},\ \texttt{Aggregator},\ \text{')'}\ ; \\
\textit{spl} & = & \text{? white space characters ?}\ |\ \text{' . '}\ ;
\end{array}
$$

Here the generate, filter and `aggregate` are three keywords (actually, they are Scala functions). Each of them has an argument that is an instance of `GeneratorCreater`, `Predicate` or `Aggregator`, respectively. Note that one GTA expression may have multiple testers inside. A single-line GTA expression is allowed to be written on multiple lines such that each line is one or more terms and with a '`.`' between any two terms.

To get a concrete image what GTA programming looks like, let us look at an example of computing the *maximum sum of all segments (contiguous sublists) of an integer list* shown in Listing 3.11. We will explain the details of allSegments and maxSum later in Section 3.4.2.

To make the programming easier, we have predefined commonly used generators, testers and aggregators in the library. Users can compose various of programs by these generators, testers and aggregators. Table 3.1 list some especially useful ones. In the table, there are four generators, which, given a list, can generate all sublists (sublists), all prefix lists (prefixes), all continuous segments (segments), or paint colors (attach some information) on each element (coloring). 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 sum (`maxSum`), minimum sum (`minSum`) , maximum probability (`maxProbability`) or the list that is the solution of above aggregations (select). Here we simply list their generic names; concrete examples (using more appropriate names, according to the context) are given in the following sections.

Table 3.1 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

Listing 3.8 A GTA program sloving the 0-1 Knapsack problem, which computes the possible maximum total value

```
1 .../* omitted */
2  val allSelects = new AllSelects[KnapsackItem]
3  val withLimit_100 = new WeightLimit(100)
4  object maxTotalVal extends MaxSum[KnapsackItem] {
5    def f(a: KnapsackItem): Int = a.value
6  }
7 /* GTA expression */
8  val gta = generate(allSelects) .
9              filter (withLimit_100) .
10               aggregate (maxTotalValue)
11 println(                                  +
12  gta.postProcess(input.map(gta.f).reduce(gta.combine))
13 )
```

3.4.2 Solving Problems with GTA

We will use the Knapsack problem and its variants as examples to show how to use the GTA library. In each example, we show how to choose proper generators, testers, and aggregators to resolve the problem. Usually, one GTA program can be written in only a few lines of Scala code, by making use of the GTA components provided in the library.

The 0-1 Knapsack problem First, recall the 0-1 Knapsack problem in the beginning of Section 3.4. We can apply the GTA algorithm that first generates all possible candidates, then filters them using the predicate of weight limitation, at last computes the total value of every remained candidate and chooses the one that has the maximum total value. This problem can be programmed by using `allSelects`, `maxTotalValue` (a modified `maxSum` for Knapsack problem,) and `WeightLimit`, as shown in Listing 3.8. The final output of this program is the maximum total value of all the candidates. The Listing 3.9 shows another GTA-Knapsack program that gives the best selection (the one with maximum total value under the weight limitation). In this program we use a `selectiveAgg` in the GTA

Listing 3.9 A GTA program sloving the 0-1 Knapsack problem, which gives the best selection

```
1 ... /* omitted */
2  val bagAgg = new BagAggregator[KnapsackItem]
3  val selectiveAgg = new SelectiveAggregator(maxTotalVal, bagAgg)
4 /* GTA expression */
5  val gta = generate(allSelects) .
6              filter (withLimit_100) .
7                aggregate (selectiveAgg)
8  println(                              +
9   gta.postProcess(input.map(gta.f).reduce(gta.combine))
10 )
```

Listing 3.10 Extended 0-1 Knapsack Problem

```
1 ...
2  val allSelects =new AllSelects[KnapsackItem]
3  val withLimit_100 =new WeightLimit(100)
4  val lessThan_10_items =LengthLimit[KnapsackItem](9)
5  object maxTotalVal extends MaxSum[KnapsackItem] {
6         def f(a: KnapsackItem): Int = a.value
7  }
8  /* define a GTA */
9  val gta = generate(allSelects) .
10             filter (withLimit_100) .
11             /* add a new filter */
12              filter (lessThan_10_items) .
13               aggregate (maxTotalValue)
14 println( /* x is input, postProcess returns the result*/
15  gta.postProcess(x.map(gta.f).reduce(gta.combine))
16 )
```

expression, which aggregates the selections to one (not just computes the total value). The implementation of `SelectiveAggregator` aggregator will be explained later in Section 6.3.

At a glance, the cost of the GTA algorithm is exponential in the number of items because it seems that an exponential number of candidates are generated. However, the GTA library optimizes the naive process and the real cost is just linear with the number of items (and quadratic with respect to the capacity of the knapsack).

Variants of the Knapsack problem A more complex example, the *multi-constraint Knapsack problem*, is shown in Listing 3.10. Here, new constraint on *the maximum number of*

Listing 3.11 A GTA program which computes the Maximum-Segment-Sum problem

```
1 package Examples
2 import GTAS._
3
4 object userApp extends GTA[Int] with App {
5     /* Spark job configuration */
6     def ctx(contex:SparkContext,input:spark.RDD[Int]) = {
7        /* GTA expression */
8         val gta=generate(allSegments) aggregate (maxSum)
9        /* compute using the Spark-MapReduce API.*/
10        val rst=input.map(gta.f).reduce(gta.combine(_, _))
11        println(     )
12   }
13 }
```

items in a knapsack is given: the predicate `LengthLimit` checks the length of the given list in a similar way to `WeightLimit`. Conceptually, an arbitrary number of testers can be used in a GTA expression. More constraints can be introduced, for example, 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 to or less/more than c. Still, we can add another constraint on *the minimum number of items in a knapsack* to extend the problem more. In all above examples, we are asked to compute the best solutions (the maximum/minimum one), and we also can find the $\texttt{k}_{\texttt{th}}$`-best` solution by slightly extending these GTA programs. Examples of $\texttt{k}_{\texttt{th}}$`-best` problems can be found in the package of our Library.

The examples presented below use more complex `generators`, `testers`, and `aggregators`.

Maximum segment sum problem Let us consider the famous maximum segment sum (`mss` for short) problem [13, 36, 67, 104, 106, 116]: given a list of integers, find the maximum sum of its all segments (contiguous sublists). This is a simplified problem of finding an optimal period in a history of changing values.

The GTA algorithm for `mss` is simple. First, choose `allSegments` as the generator that generates all the segments [67] of the input list. Then choose `maxSum` as the aggregator to compute the maximum sum of all segments. Only a few lines of Scala code are needed for this problem. Listing 3.11 shows the GTA program for `mss` problem.

The `mss` problem has many variants. For example, the segment should only contain at most one negative number, or the maximum sum has to be divisible by 3. Similar to extend Knap-

Listing 3.12 ViterbiTest is the implementation of a tester, which tests if a transition is valid or not

```
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 3.13 MaxProdAggregator is the aggregator for computing the maximum probability

```
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
}
```

sack problems, more additional predicates can be used to resolve extended `mss` problems.

Viterbi algorithm More complex problems can also be encoded by GTA. The Viterbi algorithm [128] is a dynamic programming algorithm for finding the most likely sequence of hidden states, i.e., the Viterbi path, from a given sequence of observed events. In detail, given a sequence of observed events $E = (x_1, x_2, ..., x_n)$, a set of states $S = (z_1, z_2, ..., z_k)$ in a hidden Markov model [113], probability $P_{yield}(x_i \mid z_j)$ of event $x_i \in E$ being caused by state $z_j \in S$, and probability $P_{trans}(z_i \mid z_j)$ of state z_i appearing immediately after state z_j, the algorithm computes the most likely sequence of $(z_1, z_2, ..., z_n)$ formalized as follows.

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

In [46], the GTA approach to compute above specification is introduced as follows. First, we remove the index $i-1$ in the specification. To this end, we let the expression range over

Listing 3.14 A GTA program for Viterbi algorithm, which runs on the Spark cluster

```
1 object ViterbiSpark extends GTA[Action, Tuple2[Action, (Weather,
     Weather)],Id]{
2   ...
3   val gta = generate(assTransGen) filter (viterbiTest) aggregate(
       viterbiAgg)
4       val rst = gta.postProcess(x.map(gta.f(_).get).reduce(gta.
           combine(_, _)))
5       println(                + rst)
6       System.exit(0)
7 }
```

pairs of hidden states in $S \times S$ and introduce a predicate *trans* to restrict the 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 is false otherwise. By introducing a function,

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

the above expression can be transformed into the following equivalent one:

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

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

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

The implementation of `MarkingGenerator` will be discussed in Section 6.3.

Among those associations, we want to take only those with valid transitions. To this end, *trans* is implemented as `ViterbiTest` in Listing 3.12. The method `f` extracts the mark, i.e., the associated pair of states (a pair of states corresponds to a transition between states, and the type of pair is `Trans[State]`). 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. The method `postprocess` finally checks whether the input list has a valid transition or not.

The aggregator `ViterbiMaxProdAggregator` in Listing 3.13 computes the maximum probability by using the semiring $([0,1], max, \times)$ of 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. The other parts are straightforward implementation of the semiring. Finally, the above components are composed into the GTA program as shown in Listing 3.14. Similar to the knapsack problem, this program is optimized into a linear-cost parallel algorithm on run-time. It is worth noting that we can compute the Viterbi path by replacing the aggregator with another one based on a semiring [57].

3.5 Related Work

The research on parallelization via derivation of list homomorphisms has gained great interest [37, 120, 135]. The main approaches include the function composition based method [30, 49, 71], the third homomorphism theorem based method [59, 104], and the matrix multiplication based method [117].

Algorithmic skeletons for parallel programming have been well studied from 1989 [33], and a lot of frameworks have been developed to provide those algorithmic skeletons [9, 27, 27, 81, 95]. In particular, scan is a very useful skeleton because it enables us to reuse the partial results in reduce. For example, a set of *maximum marking problems* [17] on lists can be resolved by using scan skeletons. The matching of a regular expression over a single large document in parallel has been resolved by [79] using accumulative skeletons. With the scan or accumulate computations, we can extend the technique to retrieve substrings.

The accumulate as an algorithmic parallel computation pattern has been implemented by

using MPI [74] and now is a part of the Sketo library[95] that provides a simple programming interface and efficient parallel implementation. To the best our knowledge, there is no other MapReduce implementation for the accumulative computing.

Automatic optimization of MapReduce programs is an important problem and studied by many people such as [63, 75]. These studies more or less using the traditional database query optimization approaches.

GTA proposed by [45, 46] is a new approach to systematic development of efficient parallel programs and/or list homomorphisms in which features of semirings are maximally exploited to connect naive designs and efficient implementations, so that it dramatically simplifies the development of efficient parallel algorithms. However, there is a lack of implementations that can support practical MapReduce programming. Our work is a continuation of previous research on GTA with the goal of making it work for common MapReduce frameworks.

There are also several high-level domain specific languages built upon MapReduce (Hadoop), such as Google's Sawzall [111], Apache Pig Latin [50]. They wrap MapReduce (Hadoop) and provide optimization functionalities, but they do not have optimizations similar to GTA fusion. For example, Pig has query optimizers for generating logic plan, physic plan and also MapReduce-execution plan; Manimal [75] is a framework which can analyze MapReduce programs (Java code) and apply appropriate optimizations. We believe that GTA can be imported into the designs of these languages as a primitive optimization choice.

Emoto et al. implemented the *Generators of Generators* (GoG) library [47], which basically is a limited version of GTA framework. It is specialized to a few specific generators and a limited class of predicates, but is equipped with more powerful optimization. Furthermore, it is implemented in Fortress, and has no focus on MapReduce.

Chapter 4

Calculational Approach to Constructing Parallel Programs on Graphs

There are many realistic problems on huge graphs such as analyst of social networks like Facebook, Twitter. It is a crucial problem that how large scale graphs can be efficiently analyzed. Many frameworks are proposed for this goal [56, 90]. However, there is a class of combinatorial optimization problems that are important in practice but are NP-hard, such as *Maximum Weighted Independent Set* problem, *Target Set Selection* problem, *Influence Maximization* problem and etc. These problems are hard to be computed directly regardless using parallel frameworks like MapReduce or Pregel. However, if we could transform graphs to a special form of trees called *tree decompositions*[114] of graphs, then there exist efficient dynamic programming algorithms (in polynomial time [6, 22, 131]) on such special form of trees for those NP-hard problems.

In this chapter, our main concern is: assume we can obtain sufficient tree decompositions of large graphs, how to resolve a class of combinatorial optimization problems by using the existing polynomial-time DP algorithms, and how to automatically synthesis efficient parallel programs.

We mainly resolved the following two problems. The first one is how to represent the parallel data structure for tree decompositions, so that we can store a very large tree decomposition (e.g., a tree decomposition with millions of nodes and requires one terabytes disk storage space) in a distributed file system like HDFS and do parallel computation on it. The second problem is how to do efficient parallel tree reduction/contraction in a distributed

parallel environment such as a Hadoop cluster.

4.1 Tree Decomposition of Graph

There are many problems on graphs which are important in practice but difficult to compute. However, many graph problems can be solved efficiently if the graph is a tree, because these problems often require only a bottom-up or top-town traversal of the nodes with constant work at each node.

Intuitively, tree decompositions [114] of graphs answer the question “How much does a given graph resemble a tree?" — tree decompositions of graphs exploit the “tree-like” structure for graphs.

Definition 4.1 (Tree decomposition [114]). *A tree decomposition of a graph $G = (V,E)$ is a pair $(\{X_i, i \in I\}, T)$ where $X_t \subseteq V, I = \{1,...,n\}$, and $T = (I,F)$ is a tree such that the following conditions are satisfied:*

- *the union of the subsets X_t equals the vertex set $V (1 \leq i \leq n)$, i.e. $\bigcup_{i \in I} X_t = V$;*
- *for every edge $(v,u) \in E$, there is a $i \in I$ with $u,v \in X_t$; and*
- *for every $v \in V$, if X_i and X_j contain v for some $i, j \in \{1,2,...,n\}$, then X_k also contains v for all k on the (unique) path in T connecting i and j. In other words, the set of nodes whose subsets contain v form a connected subtree of T .*

The subsets X_i are often referred to as *bags* of vertices. The *width* of a tree decomposition $(\{X_t, t \in I\}, T)$ is $max_{t \in I}|X_t| - 1$. The *treewidth* $\omega(G)$ of G is the minimum width over all tree decompositions of G. We use $U_{BD}(G)$ denoting the upper-bound of width of the tree decompositions of G, and use $L_{BD}(G)$ to denote the lower-bound. Obviously, $\omega(G) \leq L_{BD}(G) \leq U_{BD}(G) \leq n$ holds. 4.1 shows an example of a tree decomposition of width two.

A *nice tree decomposition* is a special tree decomposition, defined as follows.

Definition 4.2 (Nice Tree Decomposition). *A nice tree-decomposition T is a tree decomposition represented as a rooted binary tree. Each node of T contains exactly $k+1$ vertices. Each node belongs to one of the following three types:*

- *Leaf — a node with no child.*

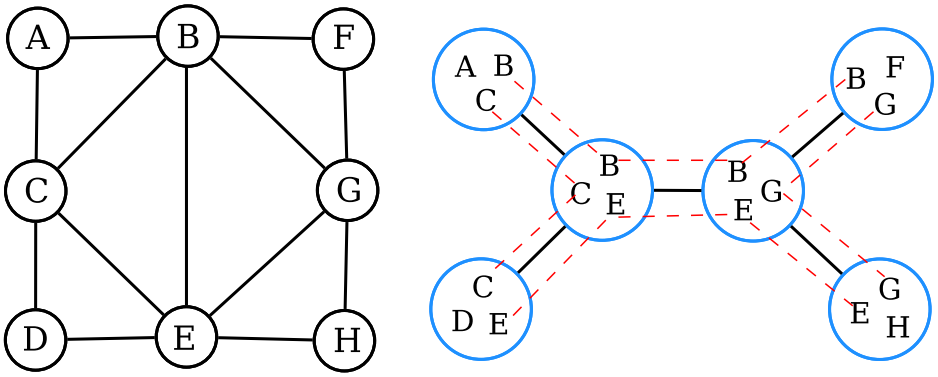


Fig. 4.1 An example of a tree decomposition of width two: blue circles (big circles) denote the bags; red dashed lines connect the same vertices between adjacent bags.

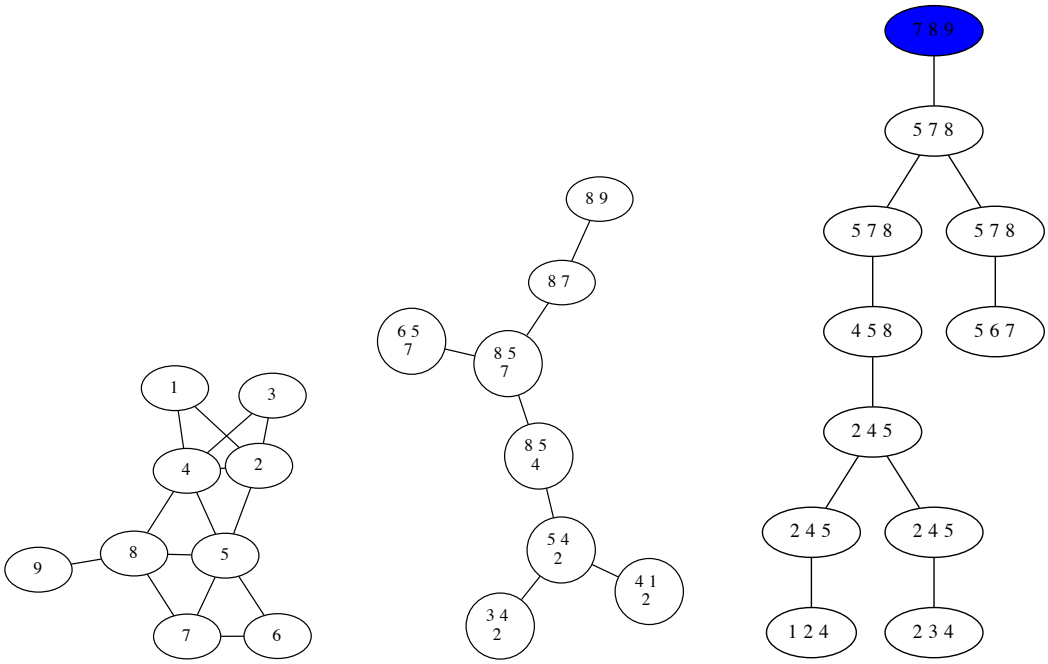
- *Replace — a node i has only one child j, and there exists a $u \in X_i$ but $\notin X_j$, and also there exists a $v \in X_j$ but $v \notin X_i$.*
- *Join — a node i has two children: j_1, j_2, and $X_i = X_{j_1} = X_{j_2}$.*

Given a tree decomposition of width $w-1$ for G, one can obtain in linear time a *nice tree decomposition* for G with the same width and with $O(wn)$ nodes. Figure 4.2 shows an example of *nice tree decomposition* of a graph. Figure 4.3 shows examples of replace node and join node. Using *nice tree decompositions*, the DP algorithms can be easier to explain [22, 24].

4.2 Parallel Computation on Tree Decompositions of Graphs

The tree contraction algorithms are very important parallel algorithms for efficient tree manipulations. It was first introduced by Miller and Reif [99], and later extended with an optimal and practical algorithm on EREW-PRAM developed by Abrahamson et al. [1]. Furthermore, implementations on hypercubes have been developed [97, 98]. Much effort has been devoted to developing tree homomorphisms for various problems, such as queries on trees [21, 34], dynamic programming [31], etc. [48, 59, 77, 89].

The original tree contraction algorithm of [99] consists of two primitive operations called rake and compress. Figure 4.5 shows examples. The rake operation merges a leaf with its parent, and the compress operation merges an internal node that has only one child with its child. Several tree contraction algorithms have been developed under the assumption



(a) A Graph G (b) A Tree Decomposition (c) A Nice Tree Decomposition

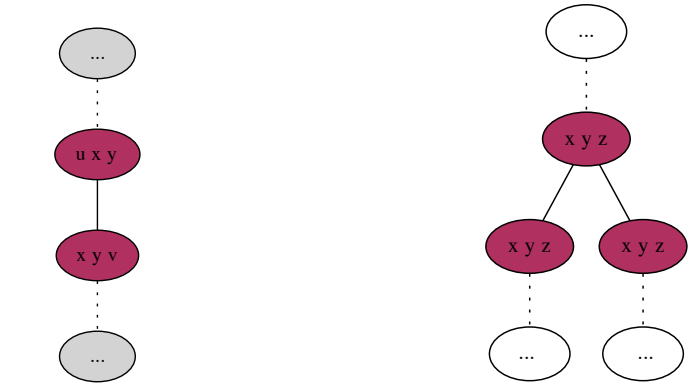
Fig. 4.2 A Graph G, A Tree Decomposition of G and A Nice Tree Decomposition of G.

of binary trees. The shunt contraction algorithm developed by Abrahamson et al. [1] uses two symmetric operations instead, namely contractL and contractR, which are successive calls of the rake operation followed by the compress operation. The contractL operation is applied to a node whose left child is a leaf, and removes two nodes and two edges from the tree as shown in Figure 4.4. The contractR operation is symmetric to the contractL operation.

4.2.1 Transforming Trees to Zippers

A zipper is a list whose elements are contexts that are left after a walk [103] on a binary tree. Figure 4.6 shows a zipper on a binary tree in this view. All the subtrees in a zipper have a uniform structure: each subtree has a hole and the hole is either the left child or the right child of the root node. A hole indicates the spot where the subtree was split. Because for each subtree one of its child is a hole, we omit the holes in Figure 4.6.

For a tree, if the order of the children of a node is not significant during the computation, we can consider the hole is always the rightmost child of a node.



(a) Replace Node (b) Join Node

Fig. 4.3 A Replace Node and A Join Node in Nice Tree Decompositions.

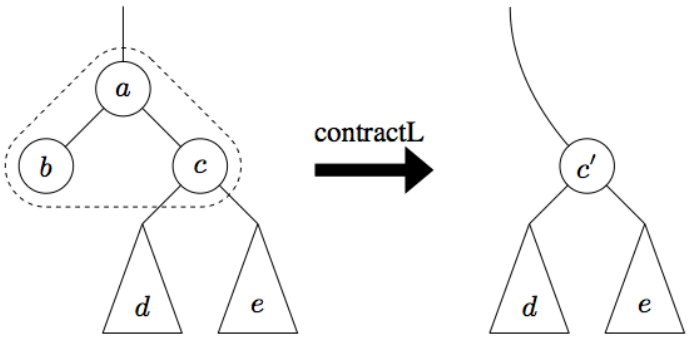


Fig. 4.4 The contractL operation

The data type for the tree elements in the zipper can be defined as:

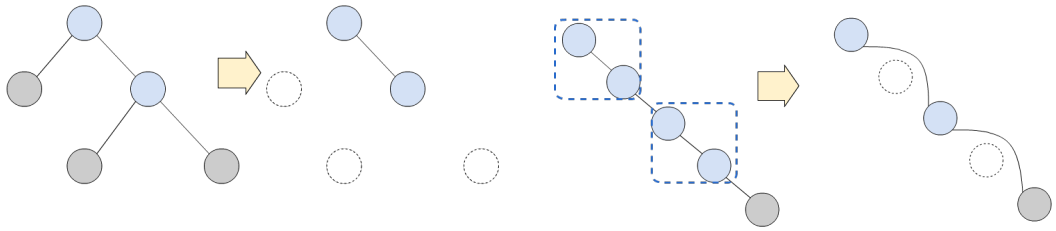
$$\textbf{data}\ \mathsf{Tree}'\ b = \mathsf{Node}'\ b\ [\mathsf{Tree}'\ b] \mid \mathsf{Leaf}'.$$

As a tree decomposition is usually not a binary tree, we extend the definition of zipper on binary trees to that on arbitrary rooted-trees. For a zipper on a tree decomposition, the elements in the list are trees with one hole. The zipper structures for trees can be specified in the following type.

$$\textbf{type}\ \mathsf{Zipper}\ b = [\mathsf{Tree}'\ b].$$

We use function walk to construct a zipper from a tree.

$$\mathsf{walk} :: \mathsf{Tree} \rightarrow \mathsf{Zipper}$$



(a) rake (b) compress

Fig. 4.5 The rake and compress Operations

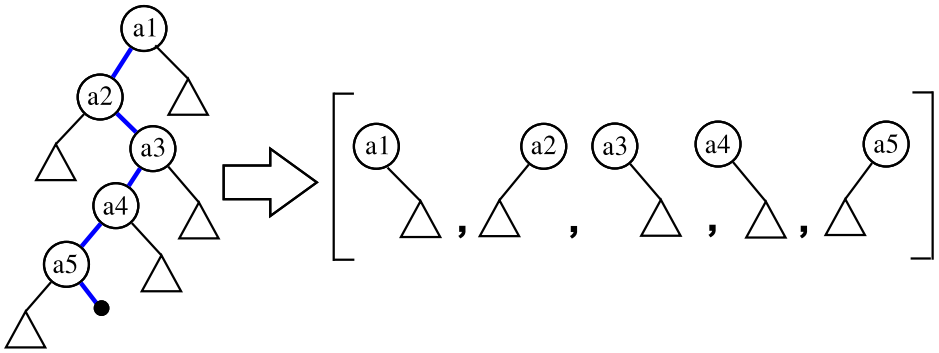


Fig. 4.6 An example of a zipper on a bianry tree, which expresses a path from the root to the black leaf. The path is shown in the blue (thicker) line.

To restore a zipper to a tree (not necessarily the original one), we use a leftward combination on the zipper to fill the hole of the previous element as the rightmost child.

$$
\begin{array}{lcl}
\mathsf{z2t} & :: & \mathsf{Zipper} \to \mathsf{Tree} \\
\mathsf{z2t}\ [\,] & = & \mathsf{Leaf} \\
\mathsf{z2t}\ ([\mathsf{Node}'\ b\ ts] +\!\!+ l) & = & \mathsf{Node}\ b\ (ts +\!\!+ \mathsf{z2t}\ l\ ).
\end{array}
$$

By using zippers, we may map a function on a tree to another function on a zipper structure which actually can be seen as a list.

4.2.2 Partition Binary Trees by Zippers

We discuss in detail about how to partition a tree decomposition to a zipper-based structure and how to apply parallel algorithms on the zipper-based structures.

There are two goals in partitioning a tree: one is to partition a tree evenly, so that the tree can be computed in parallel with good load balance; the other is to minimize the dependencies

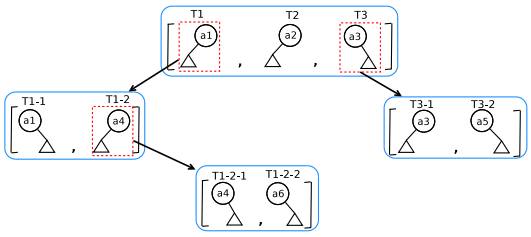


Fig. 4.7 An example of a hierarchical zipper: the subtrees in red dotted rectangles are partitioned to new zippers pointed by the arrows; the id of a subtree is shown on its top.

Listing 4.1 The Programming Interface For Botton-up Dynamic Programming Algorithms on Tree Decompositions.

```
1  public interface Homomorphism<T,R,E> {
2
3      public R unit();
4
5      public R compute(T tree);
6
7      public R combine(R r1, R r2);
8
9      public E extract(R result);
10
11     public boolean isHierarchyIrrelevant();
12
13     public boolean isSiblingIrrelevant();
14 }
```

between partitioned trees so that communication between processors can be decreased. To achieve the two goals in partitioning a general tree, we extend the zipper for binary trees to a *hierarchical zipper* for general trees. Figure 4.7 gives an example of a hierarchical zipper.

Our idea is: to keep a uniform structure, we only choose the leftmost child or the rightmost child when selecting a path from the root to a leaf node; if the size of a subtree in a zipper is larger than a *threshold*, we partition the subtree again to a new zipper. Such recursive partition forms a *hierarchical zipper* (see 4.7) which is a tree. Each node in the tree is a zipper.

Path selection strategy Here, we describe two strategies in walking downward from the root node to a leaf node.

The first strategy is called *Random strategy*. We randomly pick the leftmost child or the rightmost child as the next node in the path. In this strategy, we don't need preprocessing

on the tree.

The second strategy is *Maximum descendants strategy*. Each time, we choose the child node with the maximum number of descendants. This strategy can decrease the height of the resultant hierarchical zipper tree in most cases. However, this strategy needs preprocessing on the tree: for each node, we need to record the size of the tree rooted at the node, i.e. the number of descendants.

Deciding threshold In our partition, we limit the size of each subtree to a threshold T. The threshold T is decided by: $T = N/(P*2)$, where N is the number of tree nodes, and P is the number of processors and T is the threshold value.

Underlying implementation We describe our underlying implementations of the hierarchical zipper.

Each subtree in the hierarchical zipper has an id in the form of X-Y. X is the id of the zipper the subtree belongs to and Y is its index in the zipper. The id of a zipper is the same with the subtree the zipper is partitioned from. We add a T to the head of an id for easy description. For example, in Figure 4.7, the id T1-2-1 means the subtree is the first element in the zipper for subtree T1-2.

Each subtree has a flag which records whether a subtree is the last element in a zipper. This flag is used in the combination of the results of subtrees in a zipper. The combination is finished if the result of the last subtree in the zipper has been combined, For example, in Figure 4.7, the flag for subtree T3-1 is false because T3-1 is the first element in the zipper.

Each subtree has a flag in which each bit records whether a subtree is the last element in its zipper. This flag is used in the results combination process for a zipper. If the result of the last element in the zipper has been combined, the combination is finished. For example, in Figure 4.7, the flag for subtree T3-1 is 1-0 which means T3-1 is not the last element in zipper T3 but subtree T3 is the last element in the zipper.

All the subtrees are stored in a list in a post-order traversal of the hierarchical zipper. The partitioned subtrees (in red dotted rectangles) are not collected. For example, the subtrees in Figure 4.7 are stored as [T1-2, T1-2-1 ,T1-2-2 ,T2, T3-1, T3-2].

4.2.3 Programming Interface for Parallel Algorithms on Hierarchical Zippers

To implement a parallel algorithm on the hierarchical zipper, our library provides the following four functions. For a class of bottom-up dynamic programming algorithms, computation can be easily expressed in this programming interface, e.g *Maximum Weighted Independent Set Problem*, *Minimum Weighted Vertex Cover Problem* and *Minimum Weighted Dominating Set*.

$$\mathsf{compute} :: \mathsf{Tree}' \to B.$$

Function compute computes and returns the intermediate result of a subtree. Here, Tree' is the type of an element in Zipper (i.e., a subtree), B is the type of the intermediate result of a subtree.

$$\mathsf{combine} :: B \to B \to B.$$

Function combine merges the results of two subtrees.

$$\mathsf{recover} :: B \to B \to B.$$

Function recover recovers from the combined result of a zipper to the final result (of its original subtree).

$$\mathsf{extract} :: B \to A.$$

Functionextract computes the final result of the complete tree from the result of a hierarchical zipper. Here A is the type of final result for the problem.

When we design algorithms on zipper, we usually compute auxiliary information to help to combine subtrees. For example, the height example in [103] computes the height of a subtree as its first result and the depth of the hole as its second result. For a subtree t with height h, the result of the subtree compute t is $(h, 1)$. However, if we partition the subtree to another zipper z and combine the results of all the subtrees, i.e. reduce combine z, the result

tuple will be (h,x). Here, x is the final depth of the hole, which equals the size of subtrees in the zipper. Thus, we need a recover function to guarantee that the result on zipper can be recovered to the result on the subtree.

On the other hand, when a subtree is partitioned to a zipper, the root node of the subtree becomes the root node of the first subtree in the zipper (see node $a1$ in 4.7). As the hole is always in the root node and the first subtree contains the root node, we can recover the result from the result on zipper and the result of the first subtree in the zipper.

We give the Java-style implementations of them as follows. Listing 4.1 shows the Java Interface for representing the bottom-up dynamic programming algorithms. Users can use the five auxiliary functions to define a algorithm on tree structure, which will be computed in a bottom-up fashion on an input tree-decomposition. For example, the MIWS problem can be implemented by defining functions of this interface according to Section 4.3.2. Listing 4.2 shows first four functions of the MIWS-solving program.

4.2.4 Representations of MapReduce Programs

The parallel algorithm on hierarchical zipper consists of a map process and a reduce process.

Map. In the map process, we perform the compute function on each subtree and passes the intermediate results to the reduce process.

Reduce. In the reduce process, we group the received intermediate results by zipper id and sort the elements in each group by index. In each group, we apply the combine function on intermediate results with consecutive indices. If all the results in a group have been combined, we use the recover function to recover the result and sent the result to the reduce process. The reduce process is repeated until the result of the top-level zipper has been computed. Then the extract function is performed to compute the final result.

Apply to MapReduce. We show how to apply the parallel algorithm to the MapReduce model in an iterative manner. We divide the MapReduce passes (rounds) into a working pass and iterative passes. The iterative pass repeats until the top-level zipper has been computed. In the following, we summarize the two kinds of MapReduce passes.

Here, K is the type of the subtree id. The split function splits an id into a zipper id and an index, and return them in a pair with the zipper id as the first result. The comp function will return 1 if the first argument is greater than the second, 0 if the two arguments are equal and -1 otherwise.

The Working Pass of MapReduce. The first pass of MapReduce is called the *working pass*, which computes the results of all the subtrees and combines parts of the results. The input to the MAP phase is a list of key-value pairs of ids and subtrees, while the f_{MAP1} function takes one pair and performs compute on the subtree. In the SUFFLE&SORT phase, the f_{SHUFFLE} function is used to group results by zipper id and the f_{COMP} function is used to sort the elements in each group by index. Finally, the REDUCE phase combines the results in each group (refer to the formalization in Section 3.1).

The processing of *working pass* in MapReduce can be represented as follows:

$$
\begin{aligned}
&\mathsf{MapReduce}\ f_{\text{MAP1}}\ f_{\text{SHUFFLE}}\ f_{\text{COMP}}\ f_{\text{REDUCE}}\\
&\qquad \textbf{where}\\
&\qquad\qquad f_{\text{MAP1}} :: (K, \mathsf{Tree}') \to [(K,B)]\\
&\qquad\qquad f_{\text{MAP1}}\ (k,t) = [(k, \mathsf{compute}\ t)]\\
&\qquad\qquad f_{\text{SHUFFLE}} :: K \to K\\
&\qquad\qquad f_{\text{SHUFFLE}}\ k = \mathsf{fst}\ (\mathsf{split}\ k)\\
&\qquad\qquad f_{\text{COMP}} :: K \to K \to \{-1,0,1\}\\
&\qquad\qquad f_{\text{COMP}}\ k1\ k2 = \mathsf{comp}\ (\mathsf{snd}\ (\mathsf{split}\ k1))\ (\mathsf{snd}\ (\mathsf{split}\ k2))\\
&\qquad\qquad f_{\text{REDUCE}} :: (K,[B]) \to (K,B)\\
&\qquad\qquad f_{\text{REDUCE}}\ (k,as) = (k, \mathsf{recover}\ (\mathsf{reduce}\ \mathsf{combine}\ as))
\end{aligned}
$$

Iterative Processing. Other passes of MapReduce except the first one are iterative passes. The iterative passes combine remaining parts of the results. In an iterative pass, the MAP phase does no computation and the other two phases are the same as in the working pass. The iterative pass of MapReduce can be represented as:

$$\mathit{MapReduce}\ ([\cdot])\ f_{\text{SHUFFLE}}\ f_{\text{COMP}}\ f_{\text{REDUCE}}.$$

Result extraction When all the MapReduce passes end, we get a result key-value pair (k,b). Then the *extract* function is applied to compute the final result, which is represented as:

$$extract \circ snd.$$

4.3 On Resolving Maximum Weighted Independent Set Problem

In this section we introduce our solution for an important graph problems *Maximum Weighted Independent Set* problem, by using the techniques we have introduced in this chapter.

We assume each vertex in a graph is assigned with an int weight value. Given an undirected graph $G = (V,E)$, an independent set S is a subset of V that satisfies the condition: for any two virtices u,v in S, there is no such an edge $(u,v) \in E$.

The *Maximum Weighted Independent Set* problem is to find an independent set with the maximum total weight, which is defined as follows.

Definition 4.3 (Maximum Weighted Independent Set Problem). *For an undirected graph $G = (V,E)$, find an independent set S that the sum of weight of all vertices in S is maximum.*

4.3.1 A Generate-Test Algorithm for Combinatorial Problems on Graphs

First, we express an algorithm for the MWIS problem in the form of the generate-test (a.k.a brute force) algorithm. We use $g(vs,es)$ to represent an instance of graph with vertices vs and edges es. We use function $w(v)$ denotes the weight of v. Given a graph $g(vs,es)$, we can generate all subsets of vs and test whether a subset is a valid independent set, and then find the one with maximum sum of weight. We have defined the following functions to do so.

Function generate :: $V \to [V]$ lists all the possible selection sets and it is similar with function allSegments in Chapter 3.4.2.

Function test accepts two parameters: es of a graph g and one of its selection set xs, and

decides whether xs is an independent set.

$$\begin{aligned}&\mathsf{test} :: E \to V \to \mathsf{Bool}\\&\mathsf{test}\ es\ xs = \mathsf{not}\ (\mathsf{any}\ (\lambda\ x.\mathsf{member}\ x\ es)\ [\ (u,v)\mid u \leftarrow xs, v \leftarrow xs])\end{aligned}$$

Here, $\mathsf{not} :: \mathsf{Bool} \to \mathsf{Bool}$ and $\mathsf{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool}$ are standard Haskell [16] functions.

Function weight computes the total weight of all the selected vertices in a selection set.

$$\begin{aligned}&\mathsf{weight} :: V \to \mathsf{Int}\\&\mathsf{weight}\ xs = \mathsf{reduce}\ (+) \circ (\mathsf{map}\ w)\ xs\end{aligned}$$

The $\mathsf{mwis}_{\mathsf{gt}}$ function finds the one with the maximum weight of all the independent sets.

$$\begin{aligned}&\mathsf{mwis}_{\mathsf{gt}} :: V \to \mathsf{Int}\\&\mathsf{mwis}_{\mathsf{gt}}\ vs = \mathsf{maximum}\ [\mathsf{weight}\ xs \mid xs \leftarrow \mathsf{generate}\ vs, \mathsf{test}\ es\ xs]\end{aligned}$$

Here, function $maximum :: [a] \to a$ returns the maximum element of a list, which is a standard Haskell function.

In this example, if compute the MWIS problem on graph $G = (V,E)$ naively as the above algorithm, it will take $O(2^{|V|})$ time and actually not practical. An automatic optimization mechanism is developed by applying program transformation.

4.3.2 A Bottom-up Algorithm on Tree Decompositions

Using bottom-up dynamic programming algorithms on tree decompositions to solve MWIS has been discussed in [24, 62]. We first follow the idea and derive a bottom-up function *mwis* on tree decomposition using the functions that we have defined in Section 4.3.1. We define g_{b_t} as the induced subgraph of g on vertices in b_t.

$$
\begin{array}{lcl}
\mathsf{mwis} :: \mathit{Tree} \to [(V, \mathit{Int}]) & & \\
\mathsf{mwis}\ (\mathsf{Leaf}\ b_t) & = & [(xs, \mathsf{weight}\ xs) \mid xs \leftarrow \mathsf{generate}\ b_t, \mathsf{test}\ es_{b_t}\ xs] \\
\mathsf{mwis}\ (\mathsf{Node}\ b_t\ \mathsf{children}) & = & [(xs, \mathsf{weight}\ xs + (\mathit{reduce}(+) \circ \mathsf{map}(\mathsf{inherit}\ xs)\ [t' \mid t' \leftarrow \mathsf{children}])) \\
 & & \qquad \mid xs \leftarrow \mathsf{generate}\ b_t, \mathsf{test}\ es_{b_t}\ xs] \\
 & & \textbf{where} \\
 & & \quad \mathsf{inherit}\ xs\ t' = \mathsf{maximum}[(\mathsf{value}' - \mathsf{weight}(\mathsf{intersection}\ xs\ xs')) \\
 & & \quad \mid (xs', \mathsf{value}') \leftarrow \mathsf{mwis}(t'), \mathsf{consistent}(xs, xs')].
\end{array}
$$

Function mwis returns a list of tuples of selection set xs and its corresponding weight sum. Using graph in Figure 4.8 as the input, if we run function mwis on leaf b_0, it first generates all the possible selection sets of vertices in node b_0, then tests if they are independent sets in the induced graph g_{b_0}. For example, on leaf b_0: $\mathsf{mwis}\ (\mathsf{Leaf}\ b_0) = [([], 0), ([1], 1), ([2], 2)]$. Similarly, on leaf b_1 : $\mathsf{mwis}\ (\mathsf{Leaf}\ b_1) = [([], 0), ([4], 4), ([5], 5)]$.

Function consistent checks whether the same vertices appearing in two different nodes have the same selecting states. For each independent set xs, we choose the consistent selection set which maximizes the contributed weight in each of its children.

$$
\begin{array}{lcl}
\mathsf{consistent}(xs, ys) & = & \textbf{case}\ \mathsf{intersect}\ xs\ ys\ \textbf{of} \\
 & & \qquad xs \ \to \mathsf{True} \\
 & & \qquad ys \ \to \mathsf{True} \\
 & & \qquad \_ \ \to \mathsf{False}
\end{array}
$$

Carrying out function mwis on node b_2, similar to the procedure on a leaf node, first generates and tests all the possible independent sets in the induce graph g_{b_2}, then function inherit is used to pass up the weight contributions of the vertices in its child nodes.

$$\mathsf{mwis}\ (\mathsf{Node}\ b_2\ [b_0, b_1]) = [([\,], 2+5), ([2], 2+5), ([3], 3+1+5), ([4], 1+4)].$$

Finally, we can compute the maximum sum by as follows:

$$\mathsf{value}_{\mathsf{mwis}}\ \mathsf{tree} = \mathsf{maximum}\ ((\mathsf{map}\ \mathsf{snd})(\mathsf{mwis}\ \mathsf{tree})).$$

If the treewidth is w, there are at most $2^{(w+1)}$ many generating marking ways on each node, and there are $O(|V|)$ many nodes in a tree decomposition. The MWIS problem can be solved in $O(|V| \cdot 2^{(w+1)})$ time using the bottom-up algorithm.

4.3.3 The Parallel Algorithm on Zippers

In the bottom-up algorithm, we need to remember the selecting state xs in the root of current subtree, which is used for the further computation of ancestors (testing consistent condition). While on zippers, as shown in figure 4.9, we should remember the marking way of the leftmost and the rightmost subtree roots, for the leftward and rightward merging of partial results in the zipper.

We first duplicate the selecting state at the root of each subtree:

$$\mathsf{mwis'\ tree} = [(xs, xs, \mathsf{value}) | (xs, \mathsf{value}) \leftarrow \mathsf{mwis\ tree}].$$

We modify the bottom-up function mwis on tree decomposition T to get a leftward sequential function mwis_{up} on t's corresponding zipper.

$$\begin{array}{lcl}
\mathsf{mwis}_{up} & = & \mathsf{mwis'} \circ z2t \\
\mathsf{mwis}_{up}[(\mathsf{Node'}\ b_t\ \mathsf{children})] & = & \mathsf{mwis'}(\mathsf{Node}\ b_t\ \mathsf{children}) \\
\mathsf{mwis}_{up}([a] +\!\!+ ls) & = & [(xs_a, xs'_{ls}, \mathsf{value}_a + \mathsf{value}_{ls} - \mathsf{weight}\ (\mathsf{intersect}\ xs_a\ xs'_{ls})) \mid \\
 & & \quad (xs_a, xs'_a, \mathsf{value}_a) \leftarrow \mathsf{mwis}_{up}\ a, \\
 & & \quad (xs_{ls}, xs'_{ls}, \mathsf{value}_b) \leftarrow \mathsf{mwis}_{up}\ ls, \\
 & & \quad \mathsf{consistent}\ xs'_a\ xs_{ls}]
\end{array}$$

If the root of the rightmost subtree of zipper is considered as the root of its original tree decomposition, similar to mwis_{up}, we can compute the Maximal-Weighted-Independent-Set in rightward manner (similarly to $\mathsf{mwis}_{up}([a] +\!\!+ ls)$, we can get $\mathsf{mwis}_{down}(ls +\!\!+ [a])$ from $\mathsf{mwis}_{down} ls$ and $\mathsf{mwis}_{down} a$). When a function can be evaluated in both leftward and rightward manners, the third homomorphism theorem guarantees the existence of a parallel algorithm.

We, therefore, construct a parallel algorithm in the following way, with the definition of associative operator $\odot$ and also the mwis_{par} function as follows.

$$
\begin{array}{lcl}
\mathsf{mwis}_{par} & = & \mathsf{mwis}' \circ \mathsf{z2t} \\
\mathsf{mwis}_{par}[(\mathsf{Node}'\ b_t\ \mathsf{children})] & = & \mathsf{mwis}'(\mathsf{Node}\ b_t\ \mathsf{children}) \\
\mathsf{mwis}_{par}(a +\!\!+ b) & = & \mathsf{mwis}_{par}\ a \odot \mathsf{mwis}_{par}\ b \\
 & = & [(xs_a, xs'_b, \mathsf{value}_a + \mathsf{value}_b \\
 & & - \mathsf{weight}\ (\mathsf{intersect}\ xs'_a\ xs_b)\ \mid \\
 & & (xs_a, xs'_a, \mathsf{vlaue}_a) \leftarrow \mathsf{mwis}_{par}\, a, \\
 & & (xs_b, xs'_b, \mathsf{value}_b) \leftarrow \mathsf{mwis}_{par}\, b, \\
 & & \mathsf{consistent}\ xs'_a\ xs_b]
\end{array}
$$

Then maximum sum can be computed by value_{mwis} defined as follows:

$$\mathsf{value}_{mwis}\ \mathsf{tree} = \mathsf{max}\ ((\mathsf{map}\ thd\ )(\mathsf{mwis}_{par}(\mathsf{walk}\ \mathsf{tree}))).$$

For each subtree, we can use function mwis' to compute the partial results in parallel. Independent sets of two successive lists can be merged, if the selecting states of the rightmost root of the left list and the leftmost root of the right list are consistent. Listing 4.2 shows a Java implementation of the MIWS-solving program.

If there are p processors, and the size of zipper is n, it takes $O(|V| \cdot 2^{(w+1)}/p)$ time to compute the result of sub-list in parallel. A merging of two sub-list result takes $O(2^{2(w+1)})$ many computations. It takes $O((n\,log\,n)/p \cdot 2^{2(w+1)})$ in the merging procedure. From the practical view, the merging procedure is much faster, as the size of the pairs of selecting state can be largely reduced with the *consistent* condition.

4.3.4 Evaluation

We implemented a shared memory version of MapReduce using Java multi-thread, to evaluate our parallel algorithm for MWIS problem. Results are averaged over 5 tries.

Experiment Environment. All experiments are performed on a Linux (Ubuntu 12.04 64-bit) compute equipped with 8GB of RAM and two processors each with 4 cores (Intel®Xeon®CPU E5620 @2.40GHz).

Graph Data. For experiments on the Maximum Weighted Independent Set problem, our input is a partial k-tree generated by keeping 100% of the edges from a random 10-tree on 100,000 nodes. The 10-tree has $100,000$ nodes and $999,945$ edges.

We use an open-source tool, INDDGO [62], to construct tree decomposition of the graph. After construction, the tree decomposition of the graph has 72,523 nodes. The height of the tree decomposition is 22 and the maximum degree is 149.

Results. We compare our approach (MWIS-par) with a Java-implemented sequential program (MWIS-seq) using the dynamic programing algorithm described in [62]. The running time result of this experiment is shown in 4.10.

The speedup over the sequential program (MWIS-seq) is shown in 4.11. From the figure we can see that, the parallel algorithm for the MWIS problem achieves a nearly linear speedup over the sequential version.

The memory overhead of this experiment is shown in 4.12. As the number of subtrees increases with cores and we need to duplicate the selecting state at the root of each subtree for merging partial results in zipper both leftward and rightward, our approach requires more memory as the cores increases.

4.4 Related Work

Graph parallelization, especially on large scale graphs, has been studied intensively in recent years. In data-intensive computing domain, Pregel [90] is a bulk synchronous message passing abstraction in which all vertex-programs run simultaneously in a sequence of supersteps. GraphLab [88] is an asynchronous distributed shared-memory abstraction in which vertex-programs have shared access to a distributed graph with data stored on every vertex and edge. Gonzalez et al. [56] showed that the natural graphs commonly found in the real-world have power-law degree distributions, which challenge the assumptions made by these abstractions. So they proposed the PowerGraph [56] abstraction which exploits the Gather-Apply-Scatter model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs.

Tree contraction, which was first proposed by Miller and Reif [100], is a useful framework

for developing parallel programs on trees, and many computations have been implemented on it. However, parallel tree contraction is hard to use, because it requires a set of operations that satisfy a certain condition [103]. To this end, Matsuzaki et al. [91] proposed a systematic method of deriving efficient tree contraction algorithms from recursive functions on trees.

Tree reductions are often implemented with a tree contraction algorithm. Matsuzaki et al. [94] developed a code generation system based on tupled-ring property to automatically transform user's recursive reduction programs with annotations into parallel programs. Emoto and Imachi [48] proposed a MapReduce algorithm for tree reductions and implemented it on Hadoop.

Parallel skeletons provide parallelizable computational patterns in a concise way and conceal the complicated parallel implementations from users. Skillicorn [123] first formalized a set of binary-tree skeletons. Matsuzaki et al. [93] proposed an implementation of these parallel tree skeletons on binary trees on distributed systems, to help programmers to systematically derive efficient parallel programs using tree skeletons.

Listing 4.2 The Java Implementation of Botton-up Dynamic Programming Algorithms for MWIS Problem.

```
1
2
3     public List<MWISTriple> unit() { return unit; }
4
5     public List<MWISTriple> compute(MWISNode tree) {
6         MWISSolver mwis = new MWISSolver(graph,tree);
7         mwis.solve();
8         List<MWISTriple> items = new ArrayList<MWISTriple>();
9         for(Entry<Set<Integer>,Integer> entry: tree.dptable.
              entrySet()){
10            List<Pair<Integer,Boolean>> list = new ArrayList<
                  Pair<Integer,Boolean>>();
11            Set<Integer> indenpentSet = entry.getKey();
12            for(Integer vertex: tree.getData()){
13                list.add(new Pair<Integer,Boolean>(vertex,
                      indenpentSet.contains(vertex)));
14            }
15            items.add(new MWISTriple(list,CollectionUtils.clone(
                  list),entry.getValue()));
16        }
17        return items;
18    }
19
20    public List<MWISTriple> combine(List<MWISTriple> r1, List<
          MWISTriple> r2) {
21        List<MWISTriple> items = new ArrayList<MWISTriple>();
22        for(MWISTriple t1:r1){
23            for(MWISTriple t2:r2){
24                if(consistent(t1.second, t2.first)){
25                    int value = t1.third+t2.third-GraphUtils.
                          computeVertexWeightSum(getIntersection(t1
                          .second, t2.first), graph);
26                    items.add(new MWISTriple(t1.first,t2.second,
                          value));
27                }
28            }
29        }
30        return items;
31    }
32
33    public Integer extract(List<MWISTriple> result) {
34        int max = Integer.MIN_VALUE;
35        for(MWISTriple triple : result){
36            if(triple.third > max)
37                max = triple.third;
38        }
39        return max;
40    }
```

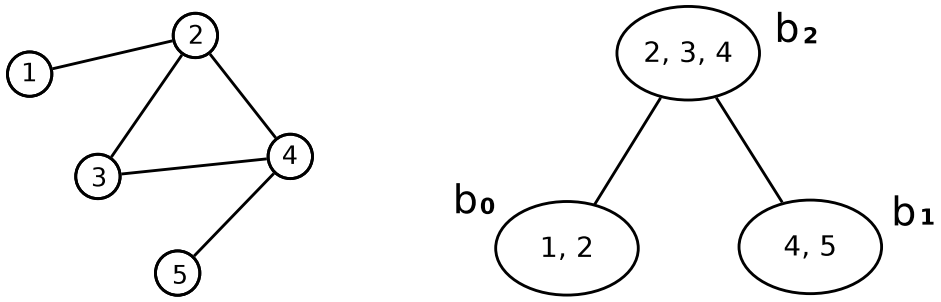


Fig. 4.8 A Weighted Graph and Its Tree Decomposition. The Weight of Each Node is Same as Its Id.

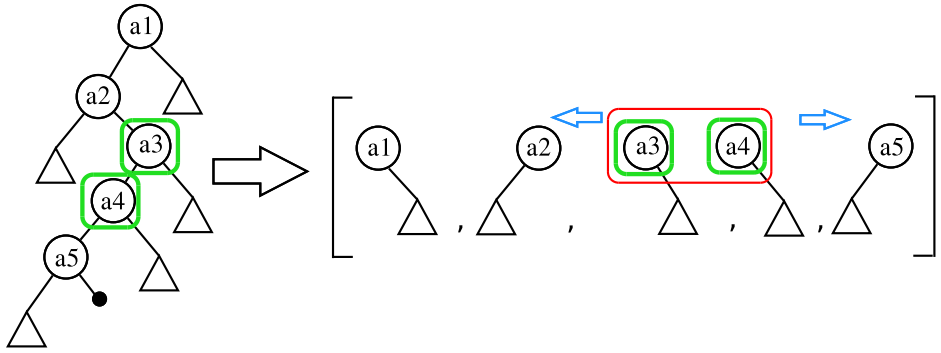


Fig. 4.9 An Example to Show Computation on a Zipper.

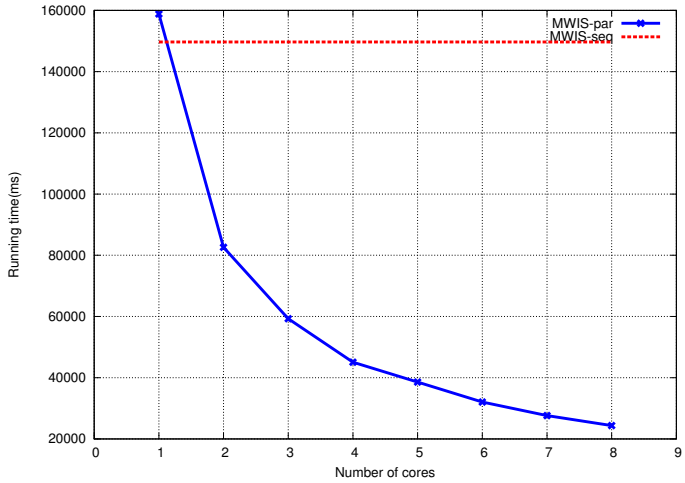


Fig. 4.10 Running time of the MWIS problem with cores.

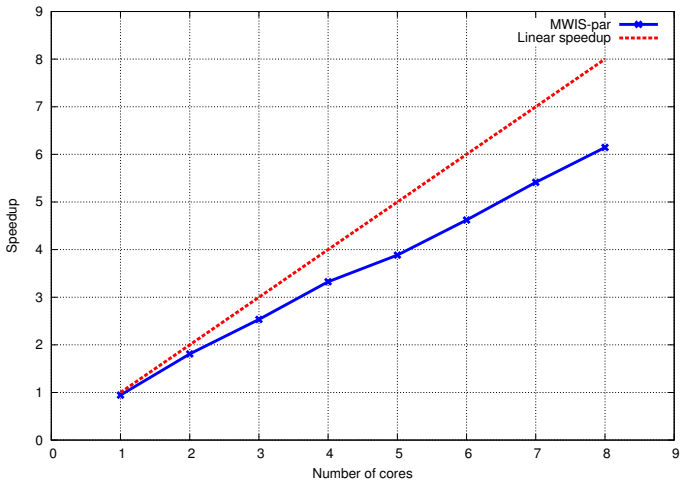


Fig. 4.11 Speedup of the MWIS problem with cores.

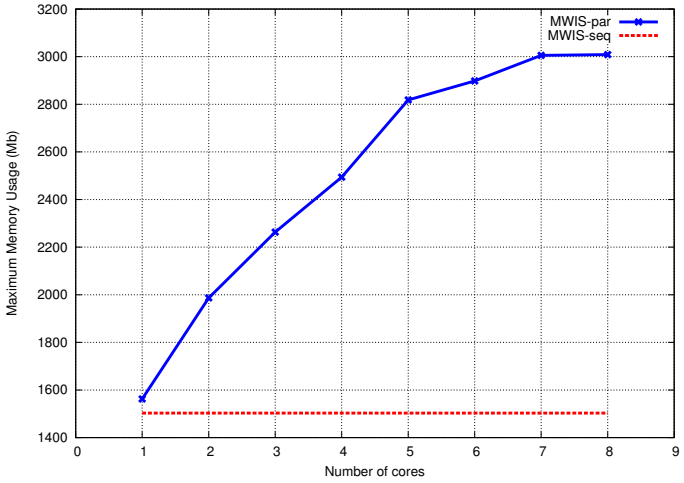


Fig. 4.12 Memory usage of the MWIS problem with cores.

Chapter 5

A Practical Approximation Approach to Optimization Problems

Polynomial time dynamic programming (DP) algorithms on tree decompositions of graphs [6, 22, 114] give us a possible way to resolve NP-hard problems on large input data sets. However, to obtain a tree decomposition (with a small enough width) of a graph is a problem in practice: it is a NP-complete problem to determine whether the treewidth of a graph is at most k [6, 114], and general graphs do not have bounded treewidth. Thus these DP algorithms are theoretically interesting but impractical in most cases. In this chapter, we introduce our approach of how to make these DP algorithms practical. We have invented a new way to put the DP algorithms introduced in [6, 22, 114] into practice. We have successfully resolved the *Target Set Selection* (TSS) problem by this approximation approach.

5.1 The Target Set Selection Problem

In social networks, individuals' behavior can influence others', which can be seen in the adoption of everyday decisions in public affairs, fashion, movie-going, and consumer behavior. This is so called "word-of-mouth" effects that the information migrates in a population through an *influential network*.

The *viral marketing* is such kind of marketing technique that imitates the diffusion process of "word-of-mouth" by intention. In viral marketing, a fundamental problem is to select a minimum initial "seed" set from the network such that the entire network adopts the products

promoted to the seed [78]. This problems can be defined as an optimization problem named the *Target set Selection* problem on graphs. Informally, the target selection problem is: given an undirected graph $G(V,E)$, and a threshold function $t : V \to \mathbb{N} \cup \{0\}$ showing the minimum number of activated neighbors to activate the vertex, find a smallest set of vertexes that can activate all vertexes in V.

An inactive vertex can be activated when $|N_{active}(v)| \geq t(v)$. Once a vertex becomes active it never falls back to inactive state [1]. The diffusion process is represented as a chain of subsets as follows.

$$\mathcal{A}[0] \subset A[1] \subset \cdots \subset \mathcal{A}[z],$$

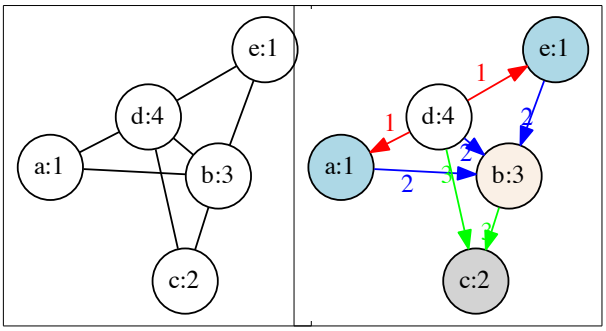
where $\mathcal{A}[i]$ denotes the set of active vertexes, $\mathcal{A}[z] = V$. $\mathcal{A}[i+1] = \{u | u \in \mathcal{A}[i]$ *or* $t(u) \leq |N_{active}(u) \cap \mathcal{A}[i]|\}$. $A[0] = S$. For $i = 1 \ldots m$, we say that S can activate $\mathcal{A}[i]$ in time step i.

TSS problem is proved NP-Complete and hard to approximate in that no polynomial approximation factor exists [28, 29, 78].

Many exact algorithms [8, 28, 84] have been proposed to resolve the TSS problem for some special graphs. For graphs with bounded treewidth [6, 114], an exact dynamic programming (DP) algorithm [8] is given in time complexity of $|V|^{O(w)}$, where w is the treewidth of the input graph. For the complete graphs (cliques) [108], a linear algorithm is given, and for trees, a polynomial-time DP algorithm is discussed [29]. Unfortunately, it is difficult to use these exact algorithms to deal with large graphs in practice. For example, for the DP algorithm in [8], the target large social networks usually do not have bounded treewidth. Moreover, computing the treewidth of a general graph is NP-hard [6, 114], and even if we can compute the treewidth it may be too large for practical use. This calls for good heuristic algorithms to solve the TSS problem [119].

In this chapter, we show, as far as we are aware, the first extension of the exact DP algorithm [8] to a novel practical heuristic algorithms for solving the TSS problem. The key ideas are a graph reduction algorithm to reduce the input graph without changing the size of the perfect target set, and an approximation algorithm to use a partial k-tree to approximate a graph. More specifically, we first apply the graph reduction algorithm to reduce the input graph. If the threewidth of the reduced graph is small enough, we apply the exact DP algorithm to compute an optimal perfect target set. Otherwise, we approximate the graphs by generating a partial k-tree and apply the exact DP algorithm to compute the result.

[1] We only refer to monotone processing. Non-monotone processing is also discussed in [78].



(a) A graph with threshold values (b) An activation procedure

Fig. 5.1 An example of TSS problem under the deterministic threshold model

Our practical implementation is in sharp contrast to the study in [8] whose interest is in theoretical complexity of the DP algorithm.

5.1.1 Formal Definition of TSS

Formally, the *Target set selection* problem can be defined as follows. Given an undirected graph $G(V,E)$, for each $v \in V$, let $deg(v)$ denote the degree of v, and $t : V \to \mathbb{N}$ be a function where $t(v)$ represents the threshold value of v for $1 \leq t(v) \leq deg(v)$. Let $N(v)$ denote the neighbors of v, and let $N_{active}(v)$ denote the active neighbors of v. Initially, the states of all vertices are inactive. Then set the state of vertexes in the *target set* (seeds) S to be active. The inactive vertexes can be activated if any of them has active neighbors no less than its threshold value. The goal is to find a minimum S that can activate V. Let $n = |V|$, then $z \leq n-1$. Note that S is not necessarily unique. In other words, there may be some sets $S_x \subseteq V, x \in \{0,1,\dots\}$ that all of them can active V, and $|S_i| = |S_j|, S_i \cap S_j \neq \emptyset, i,j \in \{0,1,\dots\}$.

As an example, if we assign the following threshold values to the vertexes in graph of Figure 5.1a like $a:1,\ b:3,\ c:2,\ d:4,\ e:1$, then if we choose d as the seed then it can active all other vertexes like Figure 5.1b: d firstly actives $\{a,e\}$, then $\{a,d,e\}$ can active $\{b\}$, at last $\{b,d\}$ can active c. Obviously, the perfect target set for this graph (with the given thresholds) is $\{d\}$.

For TSS problem we have the following observation [108].

Observation 1 (Terminated). *Let PS be a perfect target set of G. Then,* $\forall u,v,\{u,v\} \subseteq \mathcal{PS}$,

$\{u\}$ *cannot activate* $\{v\}$*, and vice verse.*

This observation means that using any one vertex in a perfect target set $\mathcal{PS}$, we cannot activate a set of vertexes which contain any other vertex that are also in $\mathcal{PS}$, otherwise these two cannot exist in $\mathcal{PS}$ simultaneously.

5.1.2 An Extension of the TSS Problem

We can extend the TSS problem to a more general version as follows.

Definition 5.1 (An Extension to TSS). *For a graph $G(V,E)$ with thresholds of all vertexes, a set $V' \subseteq V$, find a minimum set $S \subseteq V$, so that S can active V'.*

This is a variant of original TSS problem. It makes sense when we do not intend to active the whole population in the network but a set of targeted customers (vertexes). However, we can not just simply remove other vertexes to make it be a TSS problem (otherwise the connection information of targeted customers in the network will be lost). Moreover, the initial seeds are not limited to these targeted customers. This variant TSS (VTSS) is at least as hard as TSS (let $V' = V$, this problem is reduced to TSS).

5.2 Using Treewidth-Bounded Partial Graphs to Approximate TSS

We propose a new approach based on dynamic programming to resolve the TSS problem on large scale social networks under the deterministic linear threshold model. Our approach is based on the following observation.

Observation 2. *For a graph $G(V,E)$, if $H(V,E_H)$ is a partial graph of G and a set $S \subseteq V$ is a perfect target set of H, then S is also a target set of G.*

For a graph $G(V,E)$, an integer k and thresholds of every vertex as input, we firstly generate a bounded treewidth partial graph $H(V,E')$, where $E' \subseteq E$ that means we only remove some edges of G but keep all vertices. The treewidth of H is guarantied to be k according to our algorithm. Then we apply an exact algorithm on H to compute the target set S' which is a

subset of V. We show that S' can active V in both H and G, thus S' is an approximation of target set S for graph G. The soundness of this approach is given by Observation 2.

Our solution for TSS contains three modules. The first module mainly contains a reduction algorithm that reduces a large graph $G(V,E)$ to a subgraph graph $G_R(V_R,E_R)$ where $V_R \subset V, E_R \subset E$ and get a set of vertices $P, P \subseteq V \backslash V_R$, such that if S_R is a perfect target set of G_R then $S_R \cup P$ must be a perfect target set of G. We use this algorithm as pre-processing.

The second one contains an algorithm that, given an integer k and a graph $G(V,E)$, it finds a partial k-tree $H(V,E_V)$ form $G(V,E)$. Based on our experiments on real-world social networks, in many cases, the $H(V,E_V)$ obtained by this algorithm contains high percentage of edges of G, even with $k=2$ or $k=3$. We formalize this problem as the *Maximum Partial K-Tree* problem.

The third module contains a DP algorithm that computes the exact target set of a sub k-tree with bounded threshold. This algorithm is based on the DP algorithm proposed by [8]. Originally, the DP algorithm in [8] is impractical algorithm, we made some modifications and make it practical under the condition that the thresholds are bounded by a small value.

The three modules are explained as follows.

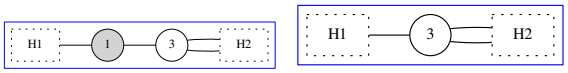
5.2.1 Graph Reduction for TSS

Data reduction [6, 23] is a polynomial-time preprocessing, which is a core tool in the development of fixed-parameter algorithms. In order to simplify the problem, we apply some useful data reduction rules on an input graph G, so that we can reduce G to smaller/simpler subgraphs and the target set for G can be obtained form those subgraphs.

On TSS problems we have some simple observations which are meant for later reference throughout this chapter. These observations are straightforward and do not need proofs.

Observation 3 (Terminated). *Let S be a perfect target set of G. Then $\forall u,v, \{u,v\} \subseteq S$ that $\{u\}$ cannot activate $\{v\}$, vice versa.*

Using any one vertex in a perfect target set $\mathcal{PS}$, we cannot activate a set of vertices which contains any other vertex that also in S, otherwise these two cannot exist in $\mathcal{PS}$ simultaneously.



(a) Before reduction (b) After reduction

Fig. 5.2 Structure “a vertex connects to a $t(x)$=1 vertex"

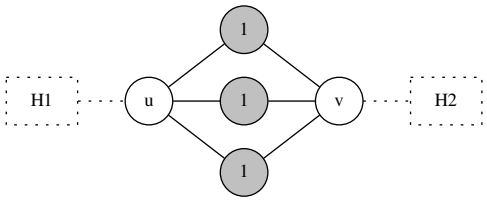


Fig. 5.3 Structure “two vertices has common $t(x)$=1 vertices in between"

Non-adjacent vertices with the same open neighbourhood (such two vertices are so-called twins) are interchangeable in any target set.

Observation 4 (Twin Exchange). *Let $G = (V,E)$ denote a graph and let $u,v \in V$ denote two twins of G with $t(u) \geq t(v)$. In addition, let S denote a target set with $v \in S$ and $u \notin S$. Then, $S' := (S \setminus \{v\}) \cup \{u\}$ is a target set for G.*

A clique K in a graph is a critical clique if all its vertices have the same closed neighbourhood and K is maximal with respect to this property. Observation 4 indicates that non-adjacent vertices with the same open neighbourhood are interchangeable in any target set.

We introduce two parameter-independent reduction rules.

Reduction Rule 1. *Let $G = (V,E)$ denote a graph, and let T be an empty set. $\forall u \in V$, if $t(u) > deg(u)$, add u to T, delete u form G and decrease the thresholds of all its (original) neighbors by one. For any rest vertex v in graph G, if $t(v) \leq 0$, then delete v and decrease the thresholds of all its neighbours by one.*

Repeatedly apply rule 1 until no more vertex can be removed, we get a smaller graph $\mathcal{H}$ which is a subgraph of G, and any perfect target set $\mathcal{PS}$ in $\mathcal{H}$ is also a perfect set for G.

For graphs contains structures like Figure 5.2, we can apply following reduction rule.

Reduction Rule 2. *Let $G = (V,E)$ denote a graph. $\forall (x,u) \in \{(x,u) | (x,u) \in E$, and $t(x) = 1$, $N(u) \cap N(x) = \emptyset\}$, delete x from G. Then $\forall v \in N(x) \setminus \{u\}$ add an edge (v,u).*

We can apply the reduction rule 2 on a graph G to remove all the vertices initially with degree one, and find a set T which belongs to a perfect target set. Then we use T to active a set of vertices. Finally, T and $Active[T]$ can be removed form the graph G. After such reduction, we get the graph G may become not connected. We can repeatedly apply this rule on all the connected components of G until no more degree-one vertex can be found. The merge of all sets of T when we apply the reduction rule 1 contains a partial perfect target set and rest parts of this perfect target set distributed in the connected components of reduced G. The reduction rule 1 can be applied exhaustively in linear time.

The reduction rule 1 is a more general rule. Similar rule appears in [108]. However in our context, it cannot be directly applied on a graph without applying rule 2. Because initially, there is no such a vertex in the input graph, which satisfies the condition $t(v) > deg(v)$.

For graphs contains structures like Figure 5.3, we can apply following reduction rule.

Reduction Rule 3. *In graph $G = (V,E)$, if $\exists (u,v), u \in V, v \in V, (u,v) \notin E$ such that $\mathcal{J} := N(u) \cap N(v)$ is not empty, and $\forall x \in \mathcal{J}, deg(x) = 1$. Then delete v and $\mathcal{J}$ form G (assume $\theta_u \geq \theta_v$), for every vertex $y \in N(v) \setminus \mathcal{J}$ add an edge (x,u).*

For graphs contains structures like Figure 5.4, we can apply following reduction rule.

Reduction Rule 4. *For a connected graph $G = (V,E)$, if $\exists u \in V, v \in V$ are twins. Then if u can active v, then delete v and $\mathcal{J}$ form G (assume $\theta_u \geq \theta_v$), for every vertex $y \in N(v) \setminus \mathcal{J}$ add an edge (x,u).*

According to Observation 4, [108] introduces another reduction rule that can reduce a critical clique in the input graph.

Definition 5.2 (Critical Clique). *A clique K in a graph is a critical clique if all its vertices have the same closed neighbourhood and K is maximal with respect to this property.*

Reduction Rule 5. *Let S denotes a perfect target set of G, let K be a critical clique in G, and let $N_G(K)$ denotes the neighbors of K in G. If $N_G(K)$ cannot active K then there must exist a set $S_K \subseteq K$ such that $S_K \subseteq S$. We can remove S_K and decrease the threshold of all $v \in N(S_K)$ by one. Also, we remove the set $Active[S_k]$ before removing S_K.*

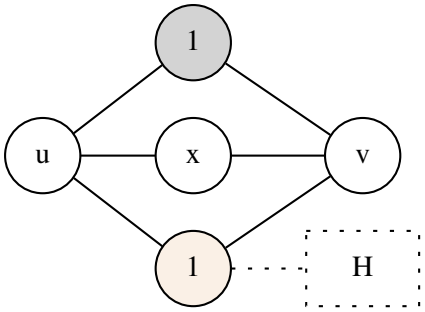


Fig. 5.4 Structure “twins that can be reduced"

Table 5.1 Reduction of subgraphs of *facebook-sg*

Total E	V_{LCC}	E_{LCC}	$V_{reduced}$	$\frac{\lvert V_{reduced}\rvert}{\lvert V_{LCC}\rvert} \times 100\%$	$E_{reduced}$	$\frac{\lvert E_{reduced}\rvert}{\lvert E_{LCC}\rvert} \times 100\%$
100,000	15,534	17,392	1,751	11.27%	3,606	20.73%
500,000	281,061	336,826	47,429	16.87%	103,084	30.60%
2,000,000	1,526,297	1,819,949	251,774	16.50%	544,428	29.91%
10,000,000	7,562,368	9,819,124	1,530,535	23.09%	3,986,513	40.46%

S_K is part of some perfect target of G. Note that $N_G(K)$ separates K from the rest of G. If activating all vertices in $N_G(K)$ is not enough to activate all vertices of K, then every target set of G has to contain some vertices of K. By Observation 4 (Twin Exchange), we can assume without loss of generality that those vertices have the highest thresholds among all vertices of K. S_K can be computed in linear time [108]. The rule 5 can be applied in $O(n(n+m))$ time [108].

For social networks, after applying reduction rule 1 - 5 , the remained part usually becomes several several orders of magnitude smaller than the population size without affect the exact solutions of TSS.

Table 5.1 shows the result of some experiments on the *facebook-sg* that is from *The Koblenz Network Collection*[76]. This time we also took subgraphs of *facebook-sg* and evaluated on them.

In Table 5.1, the column $\frac{\lvert V_{reduced}\rvert}{\lvert V_{LCC}\rvert} \times 100\%$ and $\frac{\lvert E_{reduced}\rvert}{\lvert E_{LCC}\rvert} \times 100\%$ show the results that after reduction, the remained graphs have only small parts of edges and vertices compared with original graphs.

Our strategy is firstly use above reduction rules to reduce a large (connected) graph as small as possible (may be not connected any more). Meanwhile, we get a set of vertices that must belongs to some perfect target set. Then each connected component of rest part of the input graph can be analyzed independently.

5.2.2 The Maximum Partial K-Tree Problem

For an arbitrary graph G, its treewidth is unknown and difficult to compute, that makes the linear/polynomial time DP algorithms like [6, 8, 22, 114] being impractical. Even in case we could efficiently compute the treewidth of a large graph, if the treewidth is large than 20, those DP algorithms are still impractical because of the huge constant factor [115]. Such problem is a fundamental roadblock to making the tree decomposition based DP algorithms like [8] being practical. Our approach is reducing G to be a partial k-tree by only removing a few edges, with a small k (e.g., k is 3 is a good choice for many social networks). Procedure 1 indicates the details of the algorithm.

The input is a graph $G(V,E)$ and an integer k and we assume $k \leq |V|$. First, we find top k vertices which have maximum degree in graph G, and add them to H, then apply a function $clique_H$ on them making them a clique. Here the function $clique_G(S)$ means that add edges to each pair vertices of S in the graph G, and make S be a clique (that induces a complete graph). If an edge added to H during applying $clique_H$, connected two vertices that are not connected in graph G, then we remember it to a set R and will remove it form H after the procedure is finished. After that, we continue choosing some vertex $v \in G \backslash H$ and a k-clique Clq_k in H, then adding v to H and applying $clique_H(Clq_k \cup \{v\})$ to add some new edges to H.

Function $commEdges(clq_H, G)$ computes the common edges of the complete graph induced by a clq in H and graph G. We can add all other vertices of G to H one by one in such an order that we always choose a new vertex v that make sure the $commEdges(clq_H, G)$ is maximal. Finally, we remove all edges of H, for each of which there is no such an edge between two vertices in G.

Because any partial k-tree has bounded tree k, we can obtain its tree decomposition of width k. For real-world social networks, our greedy algorithm can find quite nice partial graphs that contains most high percentage of edges of original graphs, even with very small k/. Table 5.2 shows evaluation of this algorithm on a social network *facebook-sg*. The

Procedure 1 A greedy algorithm to find a partial k-tree inside a graph

Input: a graph $G(V,E)$ and an integer $k \leq |V|$.
Output: a partial $H(V_H,E_H)$, and $\omega(H) \leq k, V_H = V$, and $E_H \subseteq E$.

1: **function** GROW(G,k)
2: find k vertices $Clq_0 = \{v_1,...,v_k\}$ that $\underset{v \in V \backslash V_H}{\arg\max} \sum_{i=1}^{k}(d(v_i))$
3: $V_H \leftarrow Clq_0$
4: $E_H \leftarrow \emptyset$
5: $R \leftarrow \emptyset$
6: $clique_H(Clq_0)$
7: **for all** $e_H \in E_H$ *but* $e_H \notin E$ **do**
8: $R \leftarrow R \cup \{e_H\}$
9: **end for**
10: **while** $|V_H| < V$ **do**
11: find a $v \in G \backslash H$ and a k-clique clq_k in H that $\underset{v \in V \backslash V_H}{\arg\max} commEdges(\{v\} \cup clq_k, G)$
12: $V_H \leftarrow V_H \cup \{v\}$
13: $clique_H(\{v\} \cup clq_k)$
14: **for all** $e_H \in E_H$ *but* $e_H \notin E$ **do**
15: $R \leftarrow R \cup \{e_H\}$
16: **end for**
17: **end while**
18: H remove all the edges that are also in R
19: **return** $H(V,E')$
20: **end function**

data set of *facebook-sg* contains 59,216,214 vertices, 92,522,012 edges stored in a text file and it describes friendship relations between users of the social network Facebook. It was collected in April of 2009 through data scraping from Facebook. We took first 50,000 3,000,000 edges (column *Total E* and *Total V*, representing subgraphs of the network) of the file and find out the largest connect component (*LCC*) in them. Then we applied the greed algorithm on each largest connect component and found a partial 3-tree, respectively. The column of *Percentage* is the percentage of the edges that each 3-tree contains, i.e $\frac{|E_k|}{|E_{lcc}|} \times 100\%$. We also computed the upper bound of treewidth of each *LCC*. For those very large subgraphs (last four rows) the upper bound of treewidth will take too long time to compute by the tools we have, so that we remain them being unknown.

We can generate the k-trees very efficiently by an optimized version of Procedure 1. Figure 5.5 shows the running time of this algorithm, using different size subgraphs of the *facebook-sg* as input.

Table 5.2 Evaluation of *Grow* on subgraphs of *facebook-sg*.

Total E	Total V	V_{LCC}	E_{LCC}	E_k	K	Percentage	TW upper-bound
50,000	46,071	5,574	6,220	6,115	4	98.31%	5
100,000	91,235	15,534	17,392	17,123	3	98.45%	5
150,000	134,681	42,019	50,251	48,923	3	97.36%	14
200,000	177,029	63,121	77,178	74,550	3	96.59%	18
400,000	350,546	199,534	240,957	231,913	3	96.25%	116
500,000	438,084	281,061	336,826	322,077	3	95.62%	116
800,000	698,725	528,425	626,447	599,066	3	95.63%	unknown
1,000,000	871,572	701,006	828,324	793,904	3	95.84%	unknown
2,000,000	1,710,774	1,526,297	1,819,949	1,530,535	3	94.10%	unknown
3,000,000	2,531,088	2,343,853	2,821,078	2,347,001	3	93.20%	unknown

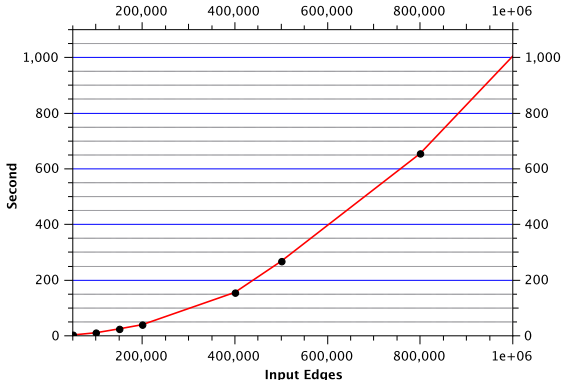


Fig. 5.5 The running time of the partial k-tree generating algorithm

Since we can use small k to generate a partial k-tree and get its tree decomposition, the DP algorithm on this tree decomposition can be efficient enough for large graphs.

For many social networks, even let $k = 2$ or $k = 3$ the partial k-tree still contains more than 90% edges and keeps similar important properties such as average degree and average local clustering coefficient. Table 5.3 shows the comparison of properties of generated 3-trees and the original input graphs. The input graphs are some connected components of the *facebook-sg*.

5.2.3 An Exact Algorithms for Computing TSS

When we get a partial graph $H(V, E_H)$ (with bounded treewidth k) of $G(V, E)$, we can compute the exact TSS S_H on the tree decomposition of H. The target set S_H can active H thus it also can active G.

Table 5.3 Comparison of properties of generated 3-trees and the original input graphs.

Input(V/E)	$\bar{d}$	$\bar{d}_{3-tree}$	$\frac{\bar{d}_{3-tree}}{\bar{d}} \times 100\%$	$\bar{C}$	$\bar{C}_{3-tree}$	$\frac{\bar{C}_{3-tree}}{\bar{C}} \times 100\%$
1,984/2,167	2.18	2.15	98.25%	0.057	0.052	91.83%
5,574/6,220	2.23	2.20	98.44%	0.081	0.074	92.26%
15,534/17,392	2.23	2.18	97.72%	0.081	0.068	84.23%
63,121/77,178	2.24	2.18	97.21%	0.086	0.082	95.03%
528,425/626,447	2.45	2.35	96.00%	0.159	0.150	94.69%

Our exact algorithm is inspired by [8] and we improved the original algorithm and makes it practical for large graphs with threshold values that the upper bound of threshold values is small (typically, less than 10).

Before we introduce the DP algorithm, let us firstly introduce some preliminaries.

The DP algorithm is based on a nice tree decomposition (Definition 4.2). The input is a nice tree decomposition $T(I,X)$ of a graph G and threshold values for each vertex of G. If X is the root node of a subtree, the vertices in X are called *boundary vertices* of G_X.

Definition 5.3 (Threshold Vector). *Let G_X denotes a subgraph of G, and it is induced by a tree τ rooted by the node X. A vector $tv \in [n]^w$, $[n] = \{0,1,2,...,n\}$ is called a threshold vector for X (also for G_X).*

The DP algorithm needs to consider different threshold assignments to the boundary vertices. For a graph G, without any limitation, the possible values of threshold are in $[0,n]$. Let T be the set of all possible threshold vectors, then $|T| = (n+1)^w$. If we give a bound of the threshold values b, then $|T_b| = (b+1)^w$. We only consider the cases that the bound of the threshold values b is small, otherwise our algorithm becomes quite inefficient for large graphs.

Definition 5.4 (Activation Order). *Let $[n]$ denotes the set $\{0,1,...,n\}$. X is a node of tree ($X = \{v,u,...\}$). Let A is a function $A = \{a_0,a_1,...\}$, an activation order a_i is a function $a_i : X \to [w-1]$ that maps a set of vertices X to $[w-1]$.*

For an activation process constrained by a_j, if $a_j(u) > (or\ <,=)a_j(v)$ means vertex u must be activated after (or before, same as) v. A is the set of all activation orders of set X.

Definition 5.5 (The Information Table). *The boundary vertices of G_X constrained by an activation order A, corresponding to a perfect target set of G_X represented as $OPT_{G_X}[T,A]$.*

The algorithm computes the perfect target set form bottom to up through the nice tree decomposition $\mathcal{T}$. For each leaf node L, it computes a table $OPT_{G_L}[T,A]$ by brute-force that means find all possible combinations of vertices with threshold values under every activation order. It will take $(d+1)^{O(w)}$ time for each leaf node.

Then for *replace nodes* and *join nodes* we do the following operations (follows the descriptions of [8]), respectively.

Replace Nodes Suppose X is a replace node with child Y in $\mathcal{T}$. Suppose G_X is obtained by adding a new boundary vertex u to G_Y , and removing another boundary vertex v from the boundary (but not from G_X). By the second condition of Definition 4.2, u can only be adjacent to other boundary vertices of G_X. Let d denote the number of these neighbors of u in G_X, and assume that they are ordered. Also, let G_X^i , for $i = 0,...,d$, denote the subgraph of G_X obtained X by adding the edges between u and and all of its neighbors in X, up-to and including the i_{th} neighbor. The table OPT_{G_X} is actually obtained by computing $OPT_{G_X^i}$ in increasing values of i, letting $OPT_{G_X} := OPT_{G_X^d}$.

When $i = 0$, u is isolated, and thus it must be included in any perfect target set when it has threshold greater than 0. For any threshold vector T for X, let T^{uv} denote the threshold vector for Y obtained by setting: $T^{uv}(w) := T(w) \forall w \neq v$, and $T^{uv}(v) := T(u)$. For an order A for X, let A^{uv} denote the set of all orderings A' for Y with $A'(w) := A(w)$ for all boundary vertices $w \neq u,v$. We allow $A'(v)A(u)$. According to the above, when X is a replace node we get for $i = 0$, we apply the following rule.

Operation Rule 1 (Replace Nodes Step-1 [8]).

$$OPT_{G_X}[T,A] = \min_{A' \in A^{uv}} \begin{cases} OPT_{G_Y}[T^{uv},A'], & \text{if } T(u) = 0 \\ OPT_{G_Y}[T^{uv},A'] \cup \{u\}, & \text{if } T(u) \neq 0 \end{cases}$$

Now if $i > 0$, then G_X^i is obtained from G_X^{i-1} by connecting u to some boundary vertex $w \in X$. For any threshold vector T, let T denote the threshold vector obtained by setting $T^{u-}(u) := maxT(u) - 1,0$, and all remaining thresholds the same. Define T^{w-} similarly. Since the edge $\{u,w\}$ can only influence v if $A(w) < A(v)$, and vice versa, we have the following rule.

Operation Rule 2 (Replace Nodes Step-2 [8]).

$$OPT_{G_X^i}[T,A] = \begin{cases} OPT_{G_X^{i-1}}[T,A], & \text{if } A(w) = A(u) \\ OPT_{G_X^{i-1}}[T^{u-},A], & \text{if } A(w) < A(u) \\ OPT_{G_X^{i-1}}[T^{w-},A], & \text{if } A(w) > A(u) \end{cases}$$

Join Nodes Let X be a join node with children Y and Z in $\mathcal{T}$. G_Y and G_Z are two subgraphs who share the same boundary vertices $Y = Z$, G_X is obtained by taking the union of these two subgraphs. This means that there is no edge between $V(GY)\backslash Y$ and $V(G_Z)\backslash Z$ in G_X. For a boundary vertex $v \in X$,let $N_{G[X]}(v)$ denote the set of boundary vertices that are connected to v in G_X. For $v \in X$, and an activation order A, let $A \leq v$ be the set of all boundary vertices u such that $A(u) < A(v)$. Given an order A, and a pair of threshold threshold T_Y and T_Z , define the threshold vector $T_Y \oplus_A T_Z$ as the vector T where a coordinate $T(v)$ for $v \in X$ is defined by

$$T(v) := T_Y(v) + T_Z(v) - |N_{G[X]}(v) \cap A^{\leq v}|.$$

We thus can compute $OPT_{G_X}[T,A]$ using the following equation:

Operation Rule 3 (Join Nodes[8]).

$$OPT_{G_X}|[T,A] = \min_{T_Y \oplus_A T_Z = T} OPT_{G_Y}|[T_Y,A] \cup OPT_{G_Z}|[T_Z,A].$$

Apply the above rules until the $\mathcal{T}$ only remains the root node, then in the table $OPT|[T,A]$ all perfect target sets can be found.

5.2.4 Implementation and Evaluation for the Target Set Selection Problem

We have implemented all algorithms introduced in this chapter, in Java. Parallel implementation for TSS can be done in the similar way as described in Chapter 4, but it is not finished yet at the current stage.

Experiment Environment. All experiments are performed on a Linux (Ubuntu 12.04 64-bit) compute equipped with 8GB of RAM and two processors each with 4 cores (Intel®Xeon®CPU E5620 @2.40GHz).

Table 5.4 Small data sets (subgraphs of *facebook-sg*)

ID	Input_E	Input_V	LCC_E	LCC_V
1	2,000	1,710	662	663
2	5,000	4,505	839	840
3	5,000	4,505	839	840
4	10,000	9,048	2,167	1,984
5	10,000	9,048	2,167	1,984
6	20,000	18,142	3,395	3,054
7	20,000	18142	3,395	3,054
8	50,000	46,071	6,220	5,574

Graph Data. We use a data set *facebook-sg*[55] form the real-world social network Facebook. The data is gathered by *The Koblenz Network Collection*[76]. This network describes friendship relations between users of the social network Facebook. It was collected in April of 2009 through data scraping from Facebook and contains 59,216,214 vertices, 92,522,012 edges. The adjacency matrix of the network in space separated values format, with one edge per line.

Results. We firstly evaluate our algorithm on some small data sets that are a cut of first sever lines of the data set *facebook-sg*. Then we find a largest connected component (LCC) for each subgraph as the input. Table 5.4 lists the subgraphs and their largest connected components.

We compared the target sets computed by our algorithm and that computed by using algorithm in [119]. We set the bound of threshold values to be 3. Table 5.5 shows the comparison [2]. The first column shows the 3-trees we generated has how much percentages of edges of input graphs. The column $|TS_A|$ is the size of target set computed by our algorithm. The column $|TS_B|$ is the size of target set computed by algorithm of [119]. On these small data sets, the fist several data set (from No.1 to No.3) two algorithms can both find the exact solution (as the size equals to 1), but on other cases, our algorithm performs significantly better than the algorithm of [119] (the size is between 38% to 57% of [119]).

We also evaluate our algorithm on bigger data sets that generated in similar way of Table 5.4. These data sets contain 500,000 to 2,000,000 edges, listed in Table 5.6. The target sets we computed are around 1% percent of all population. Currently, our algorithm find larger

[2]We generate 3-trees of the largest connected components.

Table 5.5 Comparison of results of our algorithm and [119](on data sets of Table 5.4)

ID	$\frac{\lvert E_{3-t}\rvert}{\lvert LCC_E\rvert}\%$	$\lvert TS_A\rvert$	$\lvert TS_B\rvert$	$\frac{\lvert TS_B\rvert}{\lvert TS_B\rvert}\%$
1	100%	1	1	100%
2	100%	1	1	100%
3	100%	1	1	100%
4	96.4%	3	8	38%
5	98.6%	3	8	38%
6	97.24%	9	14	64%
7	98.38%	6	14	43%
8	96.03%	16	28	57%

Table 5.6 Evaluation on big data sets (subgraphs of *facebook-sg*)

ID	Input_E	Input_V	TS	$\frac{\lvert TS\rvert}{\lvert V\rvert}\%$	Time (sec.)
1	500,000	438,084	1,710	1.03%	663
2	1,000,000	871,572	4,505	1.02%	840
3	1,500,000	1,710,774	4,505	1.02%	1,340
4	3,000,000	2,531,088	9,048	1.14%	1,984

target sets compared with [119] and take more time.

5.3 Related Work

Arnborg et al. [6] showed that many NP-hard problems posed in monadic second-order logic can be solved in polynomial time using dynamic programming techniques on input graphs with bounded treewidth. Many problems on graph, such as graph optimization problems (e.g., *Minimum Vertex Cover*, *Maximum Weighted Independent Set*), can be solved via tree decomposition using bottom-up dynamic recursive algorithms. Many researchers have investigated possibility for applying treewidth in innovative ways to help solve their problems [3, 8, 62, 80, 130]. Sullivan et al. [126] was the first one to parallelize algorithms for optimization problems. Their task-oriented bottom-up dynamic programming approach is shared-memory environment centered and is hard to be imported to MapReduce-like frameworks thus hardly handle larger datasets that beyond the capacity of a single machine. However, since the parallelization of their approach is only on the leaf nodes, the performance would be very inefficient if the shape of the tree decomposition is ill-balanced.

Many existing algorithms [8, 28, 84] for TSS are not practical for large graphs. Recently

[119] gives a good heuristic algorithm that is quite efficient and scales well, but no algorithmic guaranty is given. Exact algorithms of the *Target set selection* problem are given for some special cases. An exact dynamic programming (DP) algorithm algorithm for graphs with bounded treewidth is given by [8], in time complexity of $|V|^{O(w)}$, where w is the treewidth of the input graph. Chen [28] gives a polynomial-time DP algorithm for trees. The problem for the DP algorithm in [8] is that large scale social networks usually do not have bounded treewidth or the treewidth is too large for practical use, and also, computing the treewidth of a general graph is NP-hard [114].

Chapter 6

Implementation and Evaluations

In this chapter the details of implementation of libraries in our framework are introduced. For programming with list homomorphisms, accumulation and GTA, we provide user friendly programming interfaces respectively. The experimental results on large clusters or maulti-core machines are given.

6.1 List Homomorphism Wrapper of MapReduce

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

6.1.1 Manipulation of Ordered Data

The computation of a list homomorphism relies on the order of elements in the input list, while the input data of MapReduce are given as a set of records stored in the distributed file system. We have to represent a list as a set.

As we have explained in Section 3.2.1, in Screwdriver, we represent each element of a list as a pair of $(index, value)$ where *index* is an integer indicating the position of the element. For example, a list $[a,b,c,d,e]$ may be represented as a set $\{(3,d),(1,b),(2,c),(0,a),(4,e)\}$. Note that the list can be restored from this set representation by sorting the elements in

terms of their indices. Such indexed pairs permit storing data in arbitrary order on the distributed file systems

6.1.2 Implementing Homomorphism by Two Passes of MapReduce

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

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

$$
\begin{aligned}
&hom_{\mathrm{MR}} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \{(Int, \alpha)\} \to \beta \\
&hom_{\mathrm{MR}}\ f\ (\oplus) = getValue \circ MapReduce\ ([\cdot])\ g_{\mathrm{SHUFFLE}}\ comp\ g_{\mathrm{REDUCE}} \\
&\qquad\qquad \circ MapReduce\ ([\cdot])\ f_{\mathrm{SHUFFLE}}\ comp\ f_{\mathrm{REDUCE}}
\end{aligned}
$$

where

$$
\begin{aligned}
&f_{\mathrm{SHUFFLE}} :: Int \to Int \\
&f_{\mathrm{SHUFFLE}}\ k = k/const \\
&f_{\mathrm{REDUCE}} :: (Int, [\alpha]) \to (Int, \beta) \\
&f_{\mathrm{REDUCE}}\ (k, as) = (k, hom\ f\ (\oplus)\ as) \\
&g_{\mathrm{SHUFFLE}} :: Int \to Int \\
&g_{\mathrm{SHUFFLE}}\ k = 1 \\
&g_{\mathrm{REDUCE}} :: (Int, [\beta]) \to (Int, \beta) \\
&g_{\mathrm{REDUCE}}\ (1, bs) = (1, hom\ id\ (\oplus)\ bs) \\
&getValue :: \{(Int, \beta)\} \to \beta \\
&getValue\ \{(1, b)\} = b
\end{aligned}
$$

First pass of MapReduce: The first pass of MapReduce divides the list into some sublists, and computes the result of the homomorphism for each sublist. Firstly in the MAP phase, we wrap the key-value pair into a singleton list. Then in the SHUFFLE&SORT phase, we group the pairs so that the set-represented list is partitioned into some number of sublists and sort each grouped elements by their indices. Finally, we apply the homomor-

phism to each sublist in the REDUCE phase.

Second pass of MapReduce: The second pass of MapReduce computes the result of the whole list from the results of sublists given by the first pass of MapReduce. The MAP phase is the same as the MAP in the first pass. Then in the SHUFFLE&SORT phase, we collect the intermediate results into a single set and sort them by the their indices. Finally, we reduce the intermediate results using the associative operator of the homomorphism.

Finally, by the *getValue* function, we extract the final result from the set (of single value).

6.1.3 Implementation Issues

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

6.1.4 Experiments and Evaluation

We evaluated the scalability of programs implemented by Screwdriver, the overhead of our framework compared with native Hadoop programs, and the overhead of the non-trivial parallel program compared with sequential program.

We configured clusters with 2, 4, 8, 16, and 32 virtual machines (VM) inside the *Edubase-Cloud* of National Institute of Informatics. Each VM has one CPU (Xeon E5530@2.4GHz, 1 core), 3 GB memory, and 5 GB disk space. We installed Hadoop (version 0.20.2.203) on each VM. Three sets of programs are used for the evaluation: *SUM* computes the sum of 64-bit integers; *VAR* computes the variance of 32-bit floating-point numbers; *MPS* solves the maximum-prefix-sum problem for a list of 64bit-integers. We both implemented the programs with the Hadoop APIs directly (*SUM-MR*, *VAR-MR*, *MPS-MR*), and with our

Screwdriver (*SUM-LH*, *VAR-LH*, *MPS-LH*). Also a sequential program is implemented (*MPS-Seq*). The input for SUM and MPS was a list of 10^8 64bit-integer elements (593 MB), and the input for VAR is a list of 10^8 32bit-floating-point numbers (800 MB). Note that the elements of lists are indexed (each element has a 64bit-integer index), stored in the Avro data format, and put in the HDFS.

The experiment results are summarized in Fig. 6.1 and Table 6.1. Note that the relative speedup is calculated with respect to the result of 2 nodes. The execution of the parallel programs on our framework and on Hadoop failed on 1 node, due to the limitation of disk space for the intermediate data.

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

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

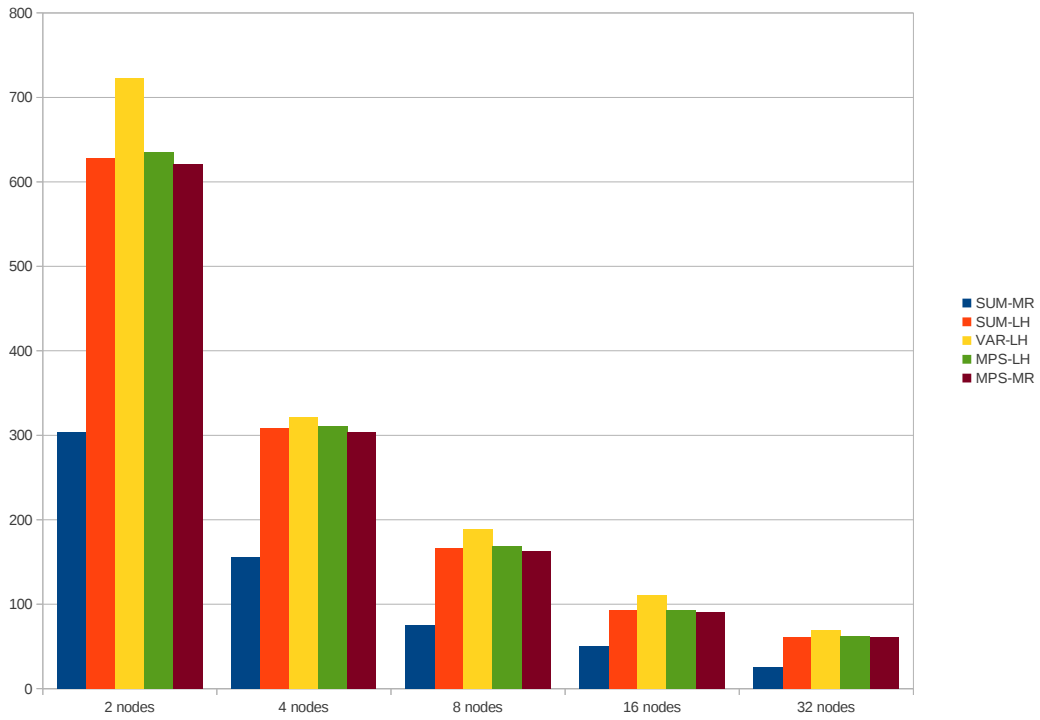


Fig. 6.1 Time Consuming of Applications with Different Numbers of Computing Nodes.

6.2 Implementation of The Accumulation Library Based on MapReduce

6.2.1 A Two-pass MapReduce Algorithm for Accumulation

From the diffusion theorem [70, 74], an accumulative function $h = [\![g, (p, \oplus), (q, \otimes)]\!]$ can be transformed into the following compositional form using the parallel skeletons scan,

Table 6.1 Execution Time (second) and Relative Speedup w.r.t. 2 Nodes

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

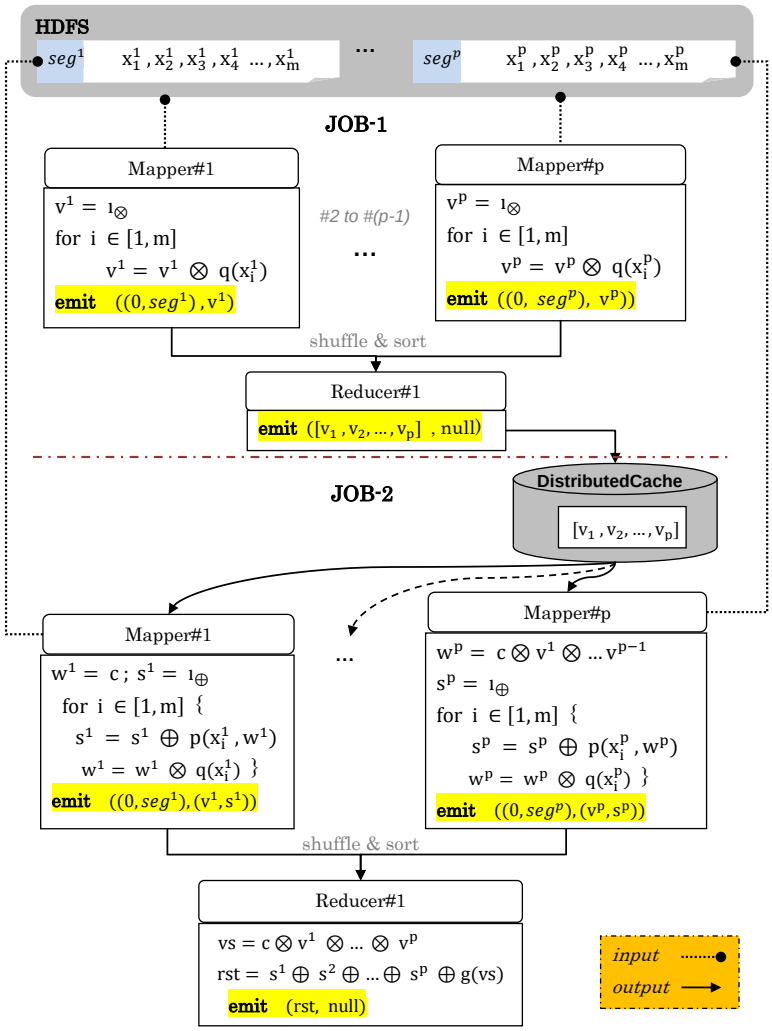


Fig. 6.2 The 2-pass MapReduce Accumulation

map, reduce and zip.

$$
\begin{aligned}
&h\ xs\ c = \mathsf{reduce}\ (\oplus)\ (\mathsf{map}\ p\ as) \oplus g\ b \\
&\quad \textbf{where} \\
&\qquad bs +\!\!+ [b] = \mathsf{map}\ (c \otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ q\ xs)) \\
&\qquad as = \mathsf{zip}\ xs\ bs
\end{aligned}
$$

In this form, $\mathsf{map}\ (c \otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ q\ xs))$ can be firstly computed to get $bs +\!\!+ [b]$, then $\mathsf{zip}\ xs\ bs$ to obtain as, and finally $\mathsf{reduce}\ (\oplus)\ (\mathsf{map}\ p\ as) \oplus g\ b$ to get the result. However, directly doing in this way will generate a lot of intermediate data such as $bs +\!\!+ [b]$, as (these are much bigger than the input), so that it is uncomputable in the MapReduce-like environments where input data are usually in terabytes. The previous MPI implementation [74] was based on this form but using a fusion technique to avoid generating large intermediate data, but mapping that fusion to MapReduce is difficult because MapReduce lacks flexible communication/synchronization mechanisms as MPI. So we developed a new “fusion" algorithm on MapReduce to efficiently compute the above compositional form.

6.2.2 The MapReduce Implementation for General Accumulation

Our approach is to divide the computation into two MapReduce phases and restrain the data transportation between the two. Suppose input list xs is split to p sublists, i.e., $xs = chk_1 +\!\!+ chk_2 +\!\!+ \ldots chk_p$. The k^{th} split chk_k has m elements $[x_1^k, x_2^k, ..., x_m^k]$ and its offset is seg_k. Our two-pass MapReduce algorithm (shown in Figure6.2) actually avoids generating large intermediate data and thus it is efficient. We introduce the details in the following paragraphs.

The First MapReduce Job. There are p Map tasks spawned for each split, in the first MapReduce job. In general, for each sublist chk_k ($k \in [1, p]$), the first MapReduce computes:

$$mapRed_{map}\ chk_k = \mathsf{reduce}\ (\otimes)\ (\mathsf{map}\ q\ chk_k).$$

We do the above computation during Map phase and just use one reducer to collect the result. In detail, each Map task iterates over the elements of its input and applies the following f_{MAP} function on each input record $(x_i^k, \_)$ ($i \in [1, m]$).

$$f_{\text{MAP}}\ (x_i^k,\ \_\ ) = (\ (0, seg_k)\ ,\ q(x_i^k))$$

Different with general MapReduce applications, here once the f_{MAP} function was applied on an input pair $(x_i^k,\ \_\ )$, the output did not be emitted immediately, but aggregated to a value v^k: $v^k = \iota_{\otimes} \otimes q(x_1^k) \otimes q(x_2^k) \otimes ... \otimes q(x_m^k)$. After the iterations, each Map task emits only one key-value pair: $(\ (0, seg_k)\ ,\ v^k)$. Here the key itself is a pair consisting of a constant value 0 and the offset seg_k. The outputs of Map tasks are grouped (by the constant value) and sorted by the offset.

In the reduce phase we only spawn one reducer. We use a special group function that groups records by first element of keys, so that the reducer collects all $(\ (0, seg_k)\ ,\ v^k))$ $(k \in [1, p])$ and sort them by the offsets seg_k. Then the reducer just emits a key-value pair $([v^1, v^2, ..., v^p],\ \_\ )$ (the key is a list and value part is useless) to the distributed file system. The f_{REDUCE} function is defined as follows.

$$f_{\text{REDUCE}}\ (0,\ [v^1, v^2, ..., v^p]\ ) = ([v^1, v^2, ..., v^p],\ \_\ )$$

Then the final result of the first MapReduce is a list that contains p elements, say $vs = [v^1, v^2, ..., v^p]$, and vs is guaranteed being in the correct order.

The Second MapReduce Job. After the first MapReduce, we initialize the second MapReduce: each second-Map task reads $vs = [v^1, v^2, ..., v^p]$ (the result list of the first reducer[1]) from HDFS, in addition to the same input data as the first Map task. After initialization, in general, for each sublist chk_k each Map task in the second MapReduce computes:

$$\begin{aligned} &mapRed_{map}\ chk_k = \mathsf{map}\ p\ (\mathsf{zip}\ chk_k\ ws) \\ &\quad \textbf{where} \\ &\qquad ws = \mathsf{map}\ (vk \otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ q\ chk_k)) \\ &\qquad vk = \mathsf{reduce}\ (\otimes)\ [c, v^1, v^2, \ldots v^{k-1}]. \end{aligned}$$

The only one Reduce task in the second MapReduce computes:

$$\begin{aligned} &mapRed_{red}\ ss = (\mathsf{reduce}\ (\oplus)\ (ss)) \oplus\ g(vp) \\ &\quad \textbf{where} \\ &\qquad ss = [s^1, s^2, \ldots s^p] \end{aligned}$$

[1] We use the *DistributedCache* function of Hadoop to implement such initialization.

$$vp = \mathsf{reduce}\ (\otimes)\ [v^1, v^2, \ldots v^p].$$

In detail, each Map task computes in a loop $s^k = p(x_1^k, w^k) \oplus p(x_2^k, w^k \otimes q(x_1^k)) \oplus ... \oplus p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$, where $w^k = c \otimes v^1 \otimes \ ... \otimes v^{k-1}$.

The output of a Map task is a nested key-value pair whose key is the same $(0, seg_k)$, and the value is (v^k, s^k). The f_{MAP} function is defined as follows:

$$f_{\text{MAP}}\ ((x_i^k,\ \_\ ) = (\ (0, seg_k)\ ,\ p(x_i^k, w^k \otimes q(x_{i-1}^k))\ ).$$

Similar to the first pass MapReduce, the outputs of all Map tasks are grouped/sorted, and we spawn a single reducer in the second MapReduce. The final result is $s^1 \oplus s^2 \oplus \cdots \oplus s^p \oplus g(c \otimes v^1 \otimes v^2 \otimes \cdots \otimes v^p)$. The f_{REDUCE} function is defined as follows:

$$f_{\text{REDUCE}}\ (0,\ ss) = (\mathsf{reduce}\ (\oplus)\ (ss +\!\!+ g(vp)\ ),\ \_\ ).$$

Here $ss = [s^1, s^2, \ldots s^p]$, and $vp = \mathsf{reduce}\ (\otimes)\ [v^1, v^2, \ldots v^p]$.

A Running Example for Simulating the Accumulation Procedure. As a concrete example, let us demonstrate the above algorithm to compute the *elimSmallers* problem on a two-nodes cluster. An input list is given as $[11, 15, 8, 9, 20, 25, 12, 23]$, the initial value of parameter $c = -\infty$, and the list is split to two (with the offset 0 and 10, respectively). The processing is represented in the Table 6.2, step by step.

6.2.3 Discussions on Efficiency

Our two-pass MapReduce algorithm for $\mathsf{accumulate}\ \ [\![g, (p, \oplus), (q, \otimes)]\!]$ has two parallel Map phases and two sequential Reduce phases, and it only generates p intermediate data $(v^1, v^2, ..., v^p)$ and duplicates them p times through networks (copied to p Map tasks using *parallel-copy*). Consider that p i.e., the number of input splits, is not a very huge value (for 1 TB data, if the chunk size of HDFS is 128MB then the p is 7813), so that if all v^k and s^k are in small constant size, then the two Reduce phases will not be bottlenecks and also the communication cost is low. This algorithm has been proved to be efficient and scalable by our evaluations. However, there is still a restriction on operators $\otimes$ and $\oplus$, in practice. Under the assumption that the input data are larger than the storage capability of any single node in

Table 6.2 An Running Example for Simulating the Accumulation Procedure

	node[1]	node[2]
input	$0:[11,15,8,9]$	$10:[20,25,12,23])$
1^{st} Map	$-\infty \uparrow 11 \uparrow 15 \uparrow 8 \uparrow 9 = 15$ **output** = $((0,0),15)$	$-\infty \uparrow 20 \uparrow 25 \uparrow 12 \uparrow 23 = 25$ **output** =$((0,10),25)$
1^{st} Reduce	emit directly **output** = $([15,25],\_)$	 N/A
2^{nd} Map	$p(11 \uparrow \underline{-\infty}) ++ p(15 \uparrow \underline{11}) +$ $+ p(8 \uparrow \underline{15}) ++ p(9 \uparrow \underline{15}) =$ $[11,15]$ **output** =$((0,0),(15,[11,15]))$	$p(20 \uparrow \underline{-\infty}) ++ p(25,\uparrow \underline{20}) +$ $+ p(12 \uparrow \underline{25}) ++ p(23 \uparrow \underline{25}) =$ $[20,25]$ **output**=$((0,10),(25,[20,25]))$
2^{nd} Reduce	$[11,15] ++ [20,25] ++ g(-\infty \uparrow$ $15 \uparrow 25)$ **output** =$([11,15,20,25],\_)$	 N/A

the cluster, that $\otimes$ must not be $++$ (or any other that has similar effect), otherwise in the Map phases of the first MapReduce job, the result of $v^k = \iota_{\otimes} \otimes q(x_1^k) \otimes q(x_2^k) \otimes ... \otimes q(x_m^k)$ may be too large to be stored in the DistributedCache nor be transported via networks, unless function q can filter out (returns an empty list) most of the elements of the input. If $\otimes$ is not $++$ but $\oplus$ is $++$, then whether the accumulate is efficient depends on the size of s^k. Here $s^k = p(x_1^k, w^k) \oplus p(x_2^k, w^k \otimes q(x_1^k)) \oplus ... \oplus p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$, and $w^k = c \otimes v^1 \otimes \ ... \otimes v^{k-1}$. For function p, if it can filter out most of its input then using only one reducer in the second MapReduce will not be a big problem, otherwise we have to do special optimization for such case by using multiple reducers.

The Optimized MapReduce Implementation for Specialized Accumulation. If the emitted intermediate data are small enough, then they can be efficiently transferred to one reducer via network otherwise the computation will be very costive or the data are too large to be manipulated by only one reducer.

In order to improve the performance for some special cases such that (in the Map phase of the second MapReduce job), $s^k = p(x_1^k, w^k) ++ p(x_2^k, w^k \otimes q(x_1^k)) ++ ... ++ p(x_m^k, w^k \otimes q(x_1^k) \otimes q(x_2^k) ... \otimes q(x_{m-1}^k))$ is a long list (suppose the input is split to p sublists), we have optimized the implementation. We do not group all output of Map phase to one reducer but use multiple reducers instead. The number of reducers can be adjusted to fit the practical problems and data. Same as the general case in Subsection 6.2.2, output of Map are sorted by seg_k ($k \in [1,p]$), but grouped to t reducers. Intuitively, let $r = p/t$, reducer^k receives $[s^{k*r+1}$,

Table 6.3 Data Sets for Evaluating Examples on Hadoop Clusters with Different Number of Working Nodes

Program	Input Length	Input Size
scan (+)	5000×2^{20}	9.77 GB
elimSmallers	5000×2^{20}	9.77 GB
los	5000×2^{20}	10.54 GB
tagmatch	5000×2^{20}	9.77 GB
mps	5000×2^{20}	9.77 GB

s^{k*r+2} ..., $s^{(k+1)r}$], and emits $s^{k*r+1} +\!\!+ s^{k*r+2} \ldots +\!\!+ s^{(k+1)r}$, $(k \in [0, r-1))$. The reducer^t receives [s^{p-r+1}, s^{p-r+2} ..., s^p] and reads [v^1, ..., v^p] from *DistributedCache*, and computes $w = c \otimes v^1 \otimes v^2 \otimes \cdots \otimes v^p$. At last the reducer^t emits $s^{k*r+1} +\!\!+ s^{k*r+2} \ldots +\!\!+ s^{(k+1)r} +\!\!+ g(w)$,. Each output from the t reducers contains part of the final result.

The Optimized Implementation for Scan. The scan skeleton is a special case of accumulate: $scan = [\![[\cdot], ([\cdot] \circ snd, +\!\!+), (id, \otimes)]\!]$, i.e., $g = [\cdot]$, $p = [\cdot] \circ snd$, $\oplus = +\!\!+$, and $q = id$. The MapReduce implementation of scan can be optimized and efficiently computed, if $\otimes$ is not $+\!\!+$. Because the result of scan is $s^1 +\!\!+ s^2 +\!\!+ \cdots +\!\!+ s^p$ we do not need the Reduce phase in the second MapReduce, and just let each mapper emit (seg_k, s^k) $(k \in [1, p])$. The seg_k denotes the offset of sublist handled by the k^{th} mapper, so that these pairs can be sorted by seg_k and form the final result.

6.2.4 Experiments and Evaluations

We evaluated the performance and scalability of the example programs with manually generated data sets shown in Table 6.5. We configured Hadoop (cdh3u5) clusters with up to 32 virtual machines (VMs) inside the *EdubaseCloud* system at National Institute of Informatics. Each VM has 2 CPUs (a CPU is one core of the Xeon E5530@2.4GHz), 6 GB RAM. The total parallel-task slots in Hadoop are configured to be equal to the total number of CPUs in the cluster [2].

[2]We made this configuration in order to simplify the analysis of scalability. In fact, optimizing the configurations of the Haddop cluster, e.g., allowing more mappers running simultaneously in each VM, can obtain much better performance.

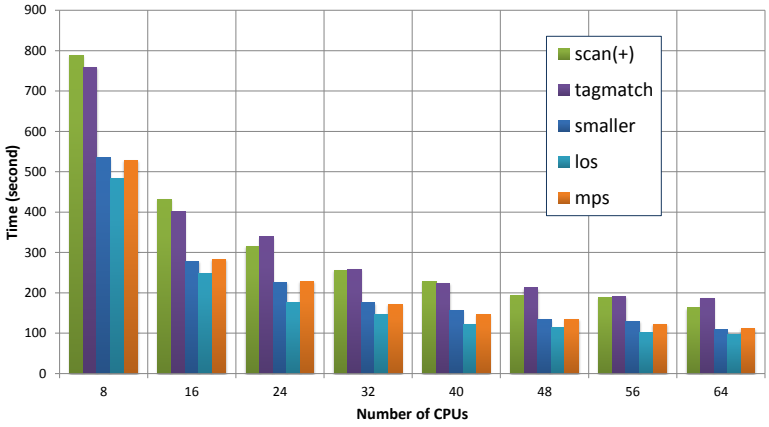


Fig. 6.3 Running Times of Each Accumulative Programs

Table 6.4 Large Data Sets for Comparing Two Kinds of Implementation: with/without Using our Framework

Input Data	Length	Size
Numbers	$1\times10^5\times 2^{20}$	195.32 GB
Pairs	$1\times10^4\times 2^{20}$	233.21 GB
Tags	$1\times10^5\times 2^{20}$	195.32 GB

Scalability. The experiment results are summarized in Figures 6.3 and 6.4. Letting the input data be fixed size (shown as Table 6.3) while increasing the working nodes of the cluster (i.e., increasing VMs), all programs have almost twice speedup when the number of CPUs increases from 8 to 16. This indicates that parallel programs written with our framework have good scalability. When the number of CPUs keeps increasing, the running-time becomes shorter and approaches to a constant value which is the time of fixed sequential parts computation and system overhead of Hadoop. As a summary, the relation between speedup (y) and number of CPUs (x) approximately fits to a linear curve $y = Ax + B$, e.g, in case of *scan* (+), $A = 6.49 \times 10^{-2}$, $B = 0.779$.

In Comparison with Vanilla Hadoop Programs Finally we discuss the programmability and relative efficiency by comparing the programs written by using our framework and those written by directly using the Hadoop API. As mentioned before, programs written by using our *accumulate* API will be transformed to programs that are exactly equivalent to those manually written by using vanilla Hadoop (i.e., without using our framework). The com-

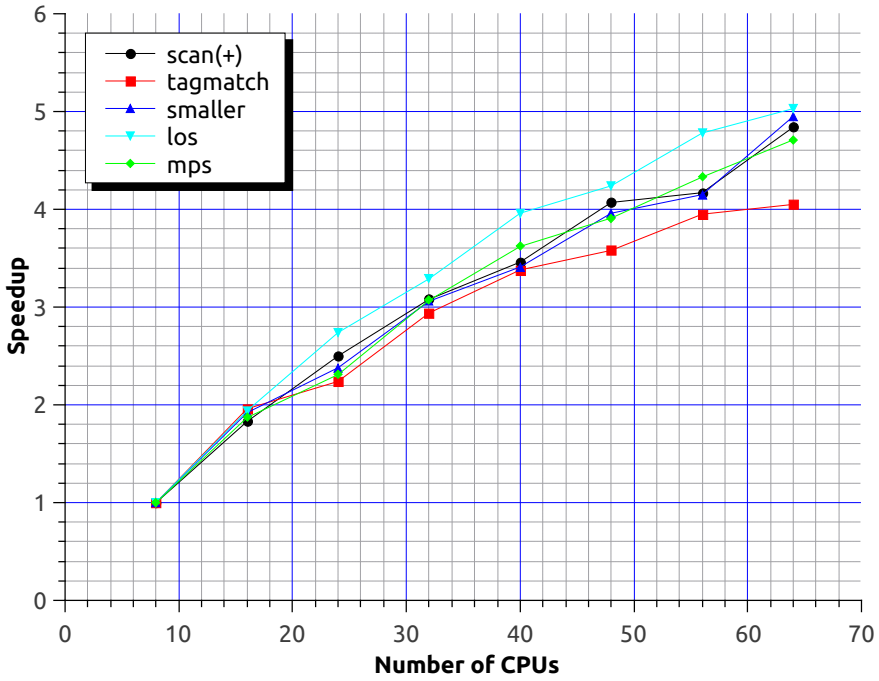


Fig. 6.4 Relative Speedup Calculated with Respect to The Result on 8 CPUs

parison we gave just shows the benefits that how much our framework saves programmers efforts and low overhead of the high-level abstraction in our framework.

For each problem in Table 6.5, we made a new version using only vanilla Hadoop, which implemented the same two-phases MapReduce algorithm in Section 6.2.1. Table 6.5 shows the comparison of length of source code and running times of the two classes of programs (on the same 64-CPU cluster). The source code is formatted by the Eclipse code formatter, and counted by using Google CodePro Analytix [3]. We used much larger data sets listed in Table 6.4 as input for the evaluation.

In Table 6.4, the column *Lines(vanilla)* is for the lengths of programs implemented without using our framework, and the column *Lines* is for the lengths of programs implemented by using our framework. The column *Time(vanilla)* and the column *Time* are running times of the two versions.

The results show that all vanilla Hadoop programs are much longer than programs written by using our *accumulate* API (3.2 – 5.6 times longer). The system overhead caused by the generic abstraction and wrapping of Hadoop API can be almost negligible. In addition, pro-

[3]https://developers.google.com/java-dev-tools/codepro/

Table 6.5 In Comparison of Length of Code and Performance to Vanilla Hadoop Programs (Using the Data Sets Listed in Table 6.4)

Problems	Lines (*vanilla*)	Lines	Time (vanilla)	Time
scan (+)	163	29	1995 s	1988 s
elimSmallers	368	107	2012 s	2013 s
los	348	81	1583 s	1587 s
tagmatch	346	107	2902 s	2903 s
mps	347	75	1793 s	1791 s

Listing 6.1 The Scala trait Monoid is the implementation for algebraic monoids

```
1 trait Monoid[T] extends BaseType {
2   def combine(l: T, r: T): T
3   def id: T
4 }
```

grams implemented by using our framework can still handle a larger input data set (nearly 20 times larger compared to each data set in Table6.3) very well.

Generally, the main difficulty for a Hadoop programmer to implement a MapReduce algorithm for problems such as *elimSmallers*, is about finding the scalable divide-and-conquer algorithm. Furthermore, even when he knows the algorithm, the implementation of the cumbersome Hadoop code is still probably very time consuming.

6.3 Implementation of The GTA Framework

We chose Scala to implement our library not only because it is a functional language with a flexible syntax and strong type system, but also because of its performance and portability (Scala is JVM based so it is compatible with most popular Java systems). We used

Listing 6.2 The Scala trait Semiring is the implementation for algebraic semirings

```
1 trait Semiring[S] extends BaseType {
2   def times(l: S, r: S): S
3   def id: S
4   def plus(l: S, r: S): S
5   def zero: S
6 }
```

Listing 6.3 The Scala trait Morphism implementats a function f mapping type `M` to type S

```
1 trait Morphism[M, +S] extends BaseType {
2    def f(in: M): Option[S]
3 }
```

Listing 6.4 The Scala class ListHomo is the implementation for list homomorphisms

```
1 abstract class ListHomomorphism[A,M]
2                 extends BaseType with Monoid[M] with Morphism[A,
                      M]{
3   /* given an input list x, the list-homomorphism can run on it
        */
4   def run(x: List[A]):M
5        = x.filter(f(_)!=None).foldLeft(id)((l,a) => combine(l,f(
             a).get))
6   /* ... omitted */
7 }
```

Spark [133] as the MapReduce engine because it is implemented in Scala and can be seen as an alternative to the Hadoop [5] framework.

6.3.1 System Architecture

The GTA library provides a domain-specific language style programming interface. All the users have to do is writing a regular Scala program consisting of GTA expressions. Each GTA expression actually defines a list homomorphism. The Scala class `MapReduceable` adapts list homomorphisms to MapReduce, as we have described in Section 3.4.1. The

Listing 6.5 The Scala trait Aggregator is the super class of all semiring homomorphisms

```
1 trait Aggregator[A, S] extends Semiring[S] with Morphism[A, S] {
2   def singleton(a: A): Option[S] = f(a)
3   def bagUnion(l: S, r: S): S = plus(l, r)
4   def nilBag: S = zero
5   def crossConcat(l: S, r: S): S = times(l, r)
6   def bagOfNil: S = id
7 }
```

Listing 6.6 MaxSum can be used to instance an aggregator that computes the maximum sum of a bag of lists

```
1 abstract class MaxSum[T] extends Aggregator[T, Int] {
2    def plus(l: Int, r: Int) = l max r
3    def times(l: Int, r: Int) = l + r
4    def f(a: T): Int
5    val id: Int = 0
6    //zero is -infty
7    val zero: Int = Int.MinValue
8 }
```

architecture of the library has three levels as Figure 6.5. The top level is the programming interface by which users write their scripts consisting of GTA expressions. The second level is the GTA fusion/transformation level. The scripts of GTA expressions are compiled to instances of `MapReduceable` (i.e., list homomorphisms). The third level is the MapReduce driver program that is the adapter of the GTA library to a MapReduce engine. The driver program tries to make use of the maximum parallelism of the engine. For example, if there are n computing nodes and each has a split of the input data, the driver program spawns n MAP processes simultaneously. Then, after MAP phase, n local-REDUCE processes will be spawned, and finally one global REDUCE process is spawned. In the future, such scheduling could be made more flexible for better parallel execution performance.

6.3.2 Algebraic Data Structures and Building Blocks of GTA

We have defined the necessary algebraic structures for GTA, as Scala traits/classes such as `Monoid` (Listing 6.1), `Semiring` (Listing 6.2), `Morphism` (Listing 6.3), and `ListHomomorphism` (Listing 6.4). The building blocks (generators, testers, and aggregators) of GTA are constructed by using such basic algebraic structures. For example, the `ListHomomorphism[A,M]` is constructed from `Monoid[M]` and `Morphis[A,M]` [4] (by using the Scala mix-in class composition mechanism).

Aggregator By definition, `aggregator` is a semiring homomorphism. The library has a generic Scala trait `Aggregator[A,S]` (Listing 6.5) which is the super class of all aggregators. The Scala trait `Aggregator[A,S]` is constructed form `Semiring[S]` and `Morphism[A,S]`.

[4] Here, `BaseType` is designed for serialization and also it hides some implicit type conversions.

Listing 6.7 SelectiveAggregator is used to make an aggregator that finds the solution

```
1 // assumption: S1 is selective: i.e., outerCombine(a, b) = a or
      b
2 class SelectiveAggregator[I,S1,S2] (selector:Aggregator[I,S1],
3                                     producer:Aggregator[I,S2])
4       extends Aggregator[I,(S1,S2)] {
5   def plus(l:(S1,S2), r:(S1,S2)) = {
6     val v1 = selector.plus(l._1, r._1)
7     if (v1 == r._1) r
8     else l
9    }
10  def times(l:(S1,S2), r:(S1,S2)) = {
11    val v1 = selector.times(l._1, r._1)
12    val v2 = producer.times(l._2, r._2)
13    (v1, v2)
14   }
15  def f(a: I) = (selector.f(a), producer.f(a))
16  val id = (selector.id, producer.id)
17  val zero = (selector.id, producer.id)
18 }
```

Here, the type parameters `A` and `S` indicate the semiring homomorphism is of type $\wr[A]\int \to S$. Its methods `plus` and `times` correspond to the operators of a semiring, and `zero` and `id` are their identity elements, respectively. For example, the abstract class `MaxSum` (Listing 6.6) is an aggregator that is used for finding the maximum among weighted sums of lists in a given bag. The method `f` determines the weights.

The generic Scala class `SelectiveAgg` (Listing 6.7) is used to make an aggregator that can be used to produce the solution. For example, in the example of GTA-Knapsack,

Listing 6.8 Predicate and FinitePredicate

```
1 /*
2  * Predicate is an almost-listhomomorphism
3  * whose postProcess returns Boolean */
4 trait Predicate[M,T] extends MapReduceable[M,T,Boolean]
5
6 /* FinitePredicate is Predicate on finite monoid*/
7 trait FinitePredicate[M,T] extends Predicate[M,T]
8                                with Finite[T]
9
10 trait Finite[T] extends Iterable[T] with Countable[T]
```

Listing 6.9 WeightLimit extends Predicate, testing if the total weight of a list is beyond the limitation

```
1 /*
2  * tests if the total weight is <= the limit w
3  */
4 object WeightLimit (w : Int) extends Predicate[KnapsackItem, Int
      ]{
5   def postProcess(t: Int) = t <= w
6   def combine(l: Int, r : Int) = (l + r) min (w+1)
7   def f(i: KnapsackItem) = (i.weight) min (w+1)
8   val id = 0
9 }
```

Listing 6.10 GeneratorCreater can be used to define a polymorphic generator

```
1 trait GeneratorCreater[I,A,P[_]] extends BaseType{
2  def gen[S](s:Aggregator[A,S]):MapReduceable[I,P[S],Any]
3 }
```

`maxTotalValue` computes the maximum sum, in order to find a solution (a set of items) which has this maximum sum, we can use a selective aggregator:

val getSolution = new SelectiveAggregator(maxTotalValue, new BagAggregator[KnapsackItem]).

`SelectiveAgg` tuples two aggregators together, the first one is the `selector` and the second one produces solutions.

Listing 6.11 allSelects is a generator which produces "all selects" of an int-list

```
1 object allSelects extends GeneratorCreater[Int,Int,Id]{
2        def makeGenerator[S](s:Aggregator[Int, S]) = new
            MapReduceable[Int,Id[S],S]{
3                override def f(i: Int):Id[S] = Id(s.plus(s.f(i),
                    s.id))
4                override def combine(l: Id[S], r: Id[S]): Id[S] =
                     Id(s.times(l, r))
5                override def postProcess(a: Id[S]): S = a
6                override val id: Id[S] = s.id
7        }
8 }
```

Listing 6.12 Prefixes is a genric Scala class for generators which produce "all prefixes"

```
1 class Prefixes[K] extends GeneratorCreater[K,K,Pair] {
2  def makeGenerator[S](s: Aggregator[K,S]) =
3        new MapReduceable[K, Pair[S], S] {
4            def f(i: K) = Pair[S](s.plus(s.id, s.f(i)), s.plus(s.
                 zero, s.f(i)))
5            def combine(l: Pair[S], r: Pair[S]):Pair[S]=
6                    Pair[S](s.plus(l.l, s.times(l.r, r.l)), s.
                        times(l.r, r.r))
7            val id: Pair[S] = Pair[S](s.zero, s.id)
8            def postProcess(a: Pair[S]): S = a.l
9     }
10 }
```

Tester The library implements homomorphic testers that 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`, and its `postProcess` method always returns a boolean value, as shown in Listing 6.8 [5]. For example, the Scala class `WeightLimit` in Listing 6.9, is a tester that checks whether the total weight of a given list is less than a given value w. It checks this condition by computing the sum of weights (in the form of a list homomorphism $(\!|f, +|\!)$) and comparing the sum with w. Since we do not need an exact value of the total weight greater than w, the implementation uses the cut-off by $\texttt{min}(\texttt{w}+1)$. The reason why we use the cut-off will be explained later in Section 6.3.3.

Generator By Definition 3.3, a generator is a polymorphic list homomorphism parameterized by a semiring. In the library, we define a generic class named `GeneratorCreater` (shown in Listing 6.10) which has a generic (polymorphic) function to produce an instance of `MapReducable`:

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

The generic function `gen[S]` takes an aggregator (`SemiringHomomorphism`) as its parameter, and produces an instance of `MapReduceable`.

Four type parameters appear in `gen` function. The type parameter `I` is determined from the input list (of type `List[I]`). `A` and `S` correspond to the type parameters of the aggregator

[5] `FinitePredicate` is explained in Section 6.3.3

Listing 6.13 Segments is a genric class for generators which produce "all segments"

```
1 class Segments[I] extends GeneratorCreater[I,I,T4] {
2  def makeGenerator[S](s: Aggregator[I, S]) = new MapReduceable[I
      ,T4[S],S]{
3    override def f(i: I) = new T4(s.f(i), s.plus(s.id, s.f(i)),
4                                  s.plus(s.id,s.f(i)), s.plus(s.
                                       zero,s.f(i)))
5    override def combine(l: T4[S], r: T4[S]): T4[S] = {
6       val ss = s.plus(s.plus(l._1, r._1), s.times(l._2, r._3))
7       val tails = s.plus(r._2, s.times(l._2, r._4))
8       val inits = s.plus(l._3, s.times(l._4, r._3))
9       val all = s.times(l._4, r._4)
10      /* T4 is a type of four-tuple: (T,T,T,T)*/
11      new T4( ss , tails, inits, all)
12     }
13   override val id: T4[S] = new T4(s.zero,s.zero,s.zero, s.id)
14   /*The 1st element of the four-tuple is the result*/
15   override def postProcess(a: T4[S]): S = a._1
16  }
17 }
```

(`Aggregator[A,S]`). The fourth type parameter `P` is used for the data type of the output, i.e., the result of the list homomorphism is of a higher-order type `P` over `S`. For example, `P[S]` can be `Id[S]` (equivalent to `S`), `Pair[S]` (equivalent to `(S,S)`), or `Triple[S]` (equivalent to `(S,S,S)`), etc. Thanks to the expressive Scala syntax, such generic programming techniques can express complex type relations inside the polymorphic generator. By extending the `GeneratorCreater`, one can define a generator by implementing the function `gen`. (Notice that `gen` is a generic function such that the instance of `MapReduceable` can only be produced by the methods of `s` whose type is `Aggregator[A,S]`.) Therefore, the Scala class `GeneratorCreater` is functionally equal to the function $generator_{\oplus,\otimes} :: (A \to S) \to [A] \to S$.

As an example, the generator `sublists` (actually, `sublists'`) can be implemented as the Scala object `allSelects` shown in Listing 6.11, which simply implements the definition. The function `gen` has slightly different type compared to the definition in Definition 3.3 [6], because we are focusing on generators whose computation patterns are suitable for the MapReduce model, while the original definition has no assumptions on computation patterns.

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

Listing 6.14 MarkingGenerator is a generic class for generators which generate new sequences with marks on each element

```
1 class MarkingGen[EV,ST](val states:Set[ST])extends
2         GeneratorCreater[E,Tuple2[EV,(ST, ST)],Id]{
3   val marks=for(x <- states ; y <-states) yield (x,y)
4   type Marked=Tuple2[EV, (ST, ST)]
5   def makeGenerator[S](s:Aggregator[Marked,S])= new
       MapReduceable[E,Id[S],S] {
6       def f(i:E): Id[S]=(s.zero /: marks) {
7                           (z:S,mk:Mark)=>s.plus(z, s.f((i, mk)))
8                         }
9       def combine(l:Id[S],r:Id[S]):Id[S]=s.times(l,r)
10      val id:Id[S]=s.id
11      def postProcess(a:Id[S]):S=a
12    }
13 }
```

The generic function `gen` returns an instance of `MapReduceable[I,P[S],S]` whose computation consists of a list homomorphism from $[I]$ to $P[S]$ and a `postProcess` from $P[S]$ to S. We designed such an interface so that users can feel comfortable writing generators like `Prefixes` shown in Listing 6.12. `Prefixes` is a generic Scala class used for instancing a generator which takes a list $xs = [x_0, x_2, ..., x_n]$ as input and produces all prefixes of xs. The standard definition of function `inits` [13, 68], which computes $inits\ xs = [[x_0], [x_0, x_1], ..., [x_0, x_1, x_2, ..., x_n]$, is as follows.

$$
\begin{array}{ll}
inits & = fst \circ inits' \\
inits'\ [\,] & = \wr[(\,,[\,])]\int \\
inits'\ [x] & = (\wr[[\,]]\int \uplus \wr\,[x]\int, \wr[\,]\int \uplus \wr\,[x]\int) \\
inits'\ (xs +\!\!+ ys) & = inits'\ xs \odot\ inits'\ ys \\
(\wr[a]\int, \wr[b]\int) \odot (\wr[c]\int, \wr[d]\int) & = (\wr[a]\int \uplus (\wr[b]\int \times_{+\!\!+} \wr\,[c]\int), \wr[b]\int \times_{+\!\!+} \wr\,[d]\int).
\end{array}
$$

The GTA `Prefixes` is just a polymorphic version of `inits`. The methods `f`, `combine`, and `id` in `Prefixes` are defined by using `plus` and `times` of the parameter `Aggregator[K,S]`. The generator `Surfixes` to produce all suffixes can be implemented in a similar way as `Prefixes`. The generator `Segments` in Listing 6.13 is similar to `allSelects` and `Prefixes`, but more complex because of its intermediate data structure. The type of intermediate data is a four-tuple `type T4[T] = (T,T,T,T)`. The details of how to construct a list homomorphism `segs` that produces all segments can be found in [67]. The GTA generator `Segments` is a polymorphic version of `segs`, similar to `Prefixes`. The implementation of

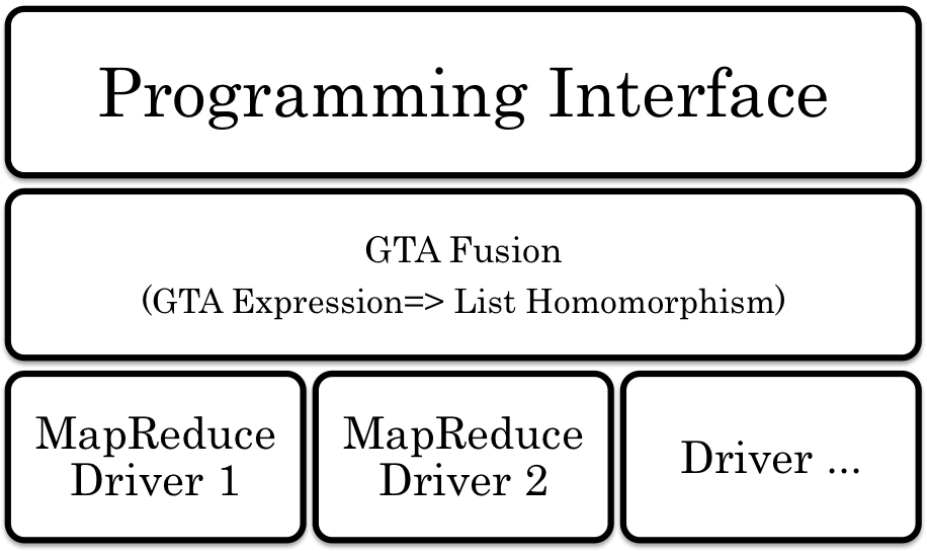


Fig. 6.5 Architecture of Our GTA Library

`MarkingGenerator` is almost the same as that of `allSelects`. The difference is that its method `f` sums up all possible associations of 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.).

Combined with various testers and aggregators, these generators can express a lot of problems. More examples can be found in the source code of our library.

6.3.3 Semiring Fusion and Filter Embedding

The GTA fusion progress can be described as a deterministic automaton (see Figure 6.6). When `generate` is invoked (by being given a polymorphic generator as the parameter), an instance of `GEN` is created. `GEN` has two methods, `filter` and `aggregate`, and it keeps the polymorphic generator. When `filter` is invoked, it composes the new predicate with the previous one. When the method `aggregate` is invoked, it embeds the predicate into current aggregator to form another aggregator with the lifted semiring (filter embedding), and substitutes it in the polymorphic generator to produce a final `MapReduceable` instance (semiring fusion).

An introduction to semiring fusion and filter embedding is given in [45, 46]. Unlike semiring fusion, which simply involves replacing an inefficient semiring with an efficient one in the

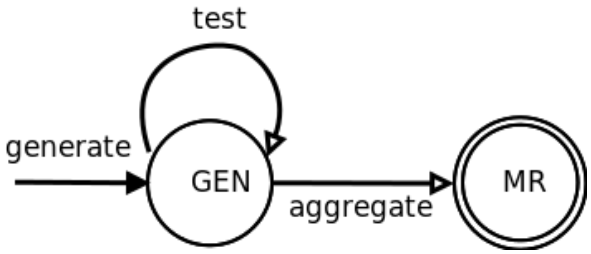


Fig. 6.6 Automaton of GTA Fusion

polymorphic generator, filter embedding is rather more complicated and thus we should discuss how it is implemented.

Filter tupling Every `filter` clause in a GTA expression introduces 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 them all at once. Tupling two list homomorphisms into one is simple [134]. Let $hh = (\!|f_1, \odot_1|\!) \vartriangle (\!|f_2, \odot_2|\!)$. We have:

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

As a result of 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 practical filter embedding, the carrier set M of a monoid $(M, \odot)$ should be finite in order to guarantee the efficiency of the final program that uses the semiring lifted by M. The key point of implementing the lifted semiring is to define a Scala class that can wrap the finite monoid and semiring. In order to define the operators $\oplus_M$ and $\otimes_M$, 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 instances of `FinitePredicate` (or its subclasses).

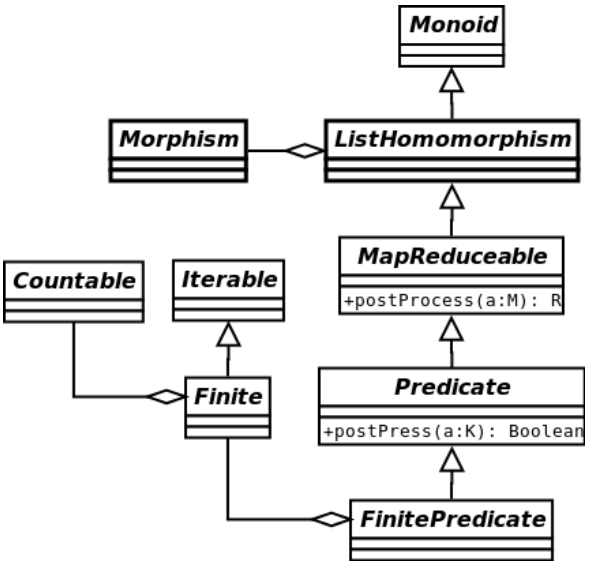


Fig. 6.7 Finite Predicate

Finite monoid and finite predicate In our implementation, a finite monoid is a monoid whose domain is a finite set. We can use the `Countable` and `Iterable` traits in Scala to define such a set. An object that inherits `Countable` must implement a `count` method. `Iterable` requires all its concrete subclasses to implement an iterator. We defined a Scala class named `FinitePredicate` (Listing 6.8). It is a `Predicate` with a finite domain. Figure 6.7 shows the class inheritance. To guarantee that the final program has a linear cost, the filter clauses in a GTA expression have to be under the constraint: all the `filters` take `FinitePredicate` as their parameters.

The rest of filter embedding is to use the tupled `tester` together with the `aggregator` in order to construct the lifted semiring (Definition 3.2) [45, 46]. 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. Interested readers can find details in the source code.

6.3.4 Serialization

For MapReduce frameworks like Spark and Hadoop, the outputs of MAP and REDUCE need to be serialized for saving in the file system or being transfered through networks. The data structures of the input/output data should be serializable, and thus a serialization framework which supports generic programming is needed. The intermediate data structures produced by GTA also need to be serialized for fault tolerance.

Table 6.6 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

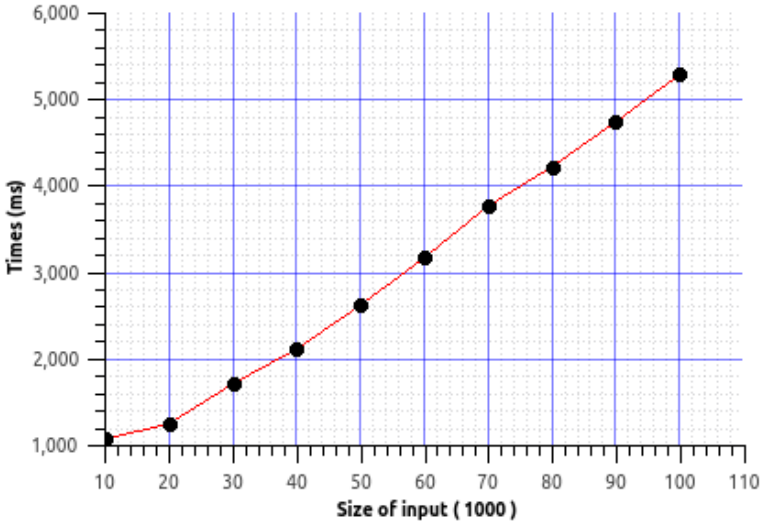


Fig. 6.8 Execution Time of GTA Programs on Single CPU

6.3.5 Performance Evaluation of Our GTA Framework

We evaluated our GTA library on sequential and parallel (distributed) models, and results proved the efficiency and scalability of GTA programs.

The algorithmic efficiency of our GTA library is evaluated by running a GTA-Knapsack program in a sequential model. The machine we used had a 2 GHz Intel Core Duo CPU (two cores) with 2 GB RAM, and the Java VM heap size was set as: $JAVA\_OPTS = -Xmx1024m \quad -Xms256m$. The Scala version was *2.9.2-final*. The knapsack items were generated and kept in memory, and the range of weights was (0, 10], the capacity of the knapsack (W) was 100.

The first test case compared the GTA-Knapsack program with a naive-Knapsack program which implemented the brute-force algorithm: generated all sublists, then filtered them and finally made max-sum on the rest ones . Table 6.6 compares the running times. Obviously,

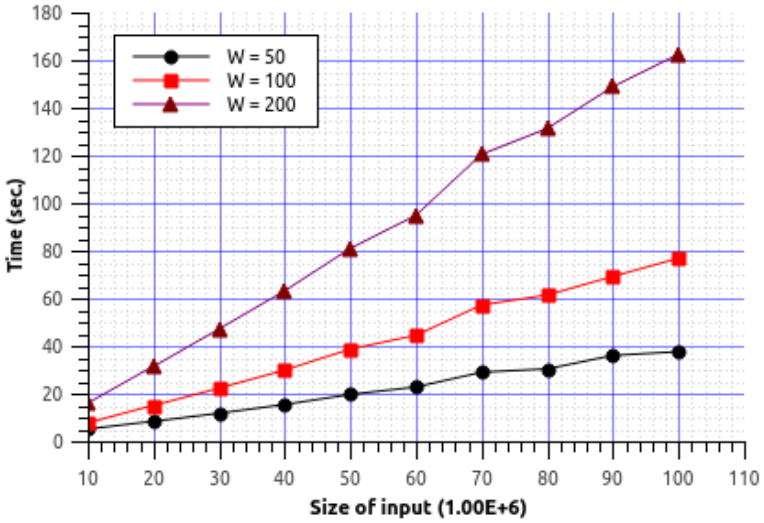


Fig. 6.9 Execution Times of GTA-Knapsack Program for Different W (64 CPUs)

the GTA-Knapsack was much faster than the naive one. The optimized dynamic programming solution for the 0-1 Knapsack problem can run in $O(nW)$ time, which is theoretically faster than our GTA-Knapsack ($O(nW^2)$ time in sequential, and $O((p+n/p)W^2)$ in parallel [45]). But without careful optimization, a dynamic programming solution could be even slower. A notable superior point of our GTA library is that its optimization is transparent to programmers.

The second experiment evaluated the GTA-Knapsack program with different size of input data. The running time of GTA-Knapsack was linear with the increasing of input data size (from 1×10^4 to 1×10^5). Figure 6.8 shows the linear algorithmic efficiency of GTA-Knapsack.

Cloud computing environment Our MapReduce clusters are built on the Edubase-Cloud (at the National Institute of Informatics, Japan). Edubase-Cloud is a cloud computing environment similar to Amazon EC2. We have authority to use up to 32 virtual-machine (VM) nodes. Each VM has two single-core CPUs (Intel® Xeon® X5650), 6 GB RAM, and 30 GB of hard disk space. We configured Spark-clusters (Spark-0.7.0) with 8, 16, 24, 32, 40, 48, 56, and 64 CPUs. During the experiments, testing data sets were generated and cached in the memmory of the cluster.

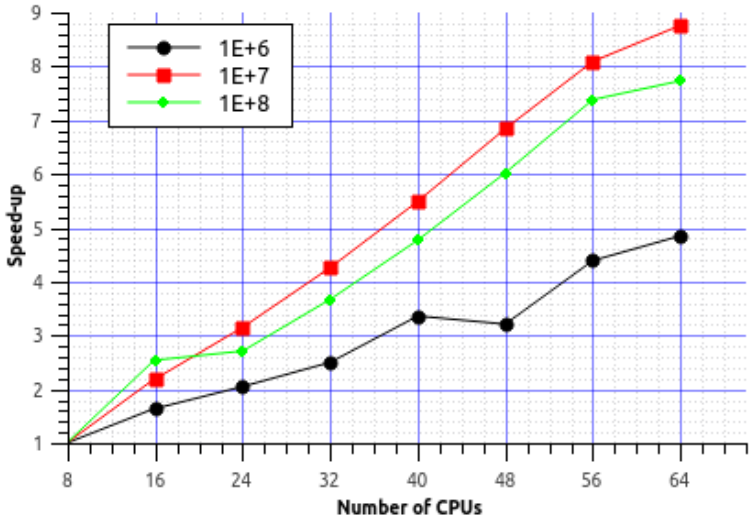


Fig. 6.10 Speedup of GTA-Knapsack on Spark Clusters

Knapsack problem and its variants First, we prepared testing-data sets as randomly generated Knapsack items, in order to illustrate scalability of the programs. The size of input data was form 1×10^7 (10 million) to 1×10^8 (100 million). Figure 6.9 compares the performance of GTA-Kampsack programs with different capacity limitations W. The results show that the larger W is, the slower the processing becomes. Figure 6.10 plots the linear speedup of GTA-Knapsack versus the number of CPUs. We tested it by using three different sets of input data.

Maximum segments sum (MSS) The input data for MSS and its variants were randomly generated signed-integers. The size of testing-data was form 1×10^7 to 1×10^8. We evaluated a GTA MSS and two extended-GTA-MSS programs: MSS-E1 requires that a valid segment should be shorter than 100. MSS-E2 requires that number of even integers in any valid segment must be less than 100. The results of execution times are shown in Figure 6.11. The GTA MSS is very efficient because it does not have a `filter` term so that the GTA fusion in it does not make complex intermediate tables, and the operators used in the REDUCE phase are $+$ and $\uparrow$. Given 100 million integers as its input, the GTA MSS can finish in around 1 second. Figure 6.11 shows the liner execution times of MSS-E1 and MSS-E2, with the increasing of input data size.

Vitirbi Algorithm The experiments on the Vitirbi algorithm were done in a similar manner to above experiments. We used a program generating all input data under a simulated

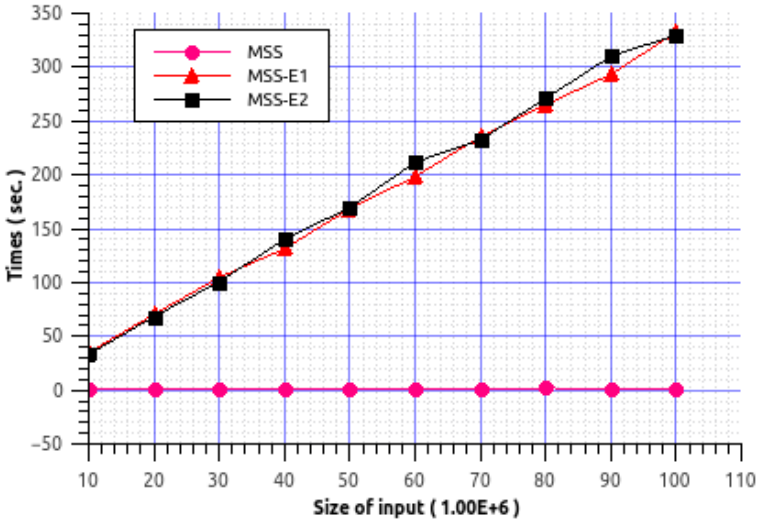


Fig. 6.11 Execution Times of GTA-MSS Programs on Spark Cluster (64 CPUs)

HMM model. Figure 6.12 shows the execution times for different numbers of CPUs.

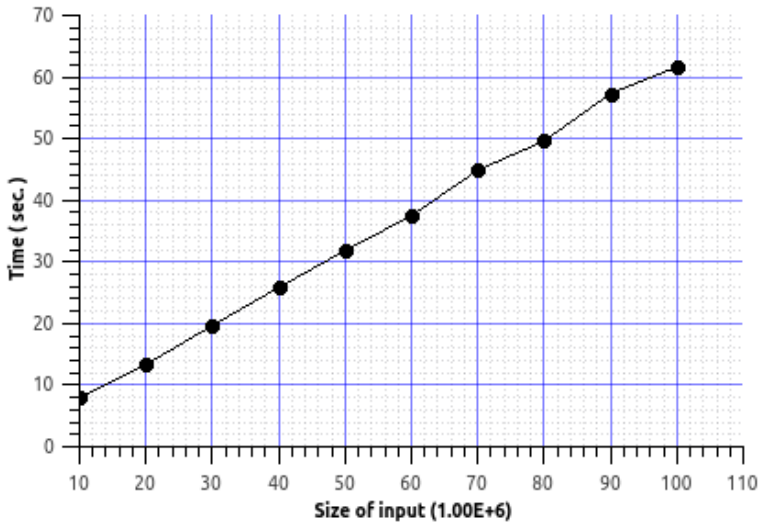


Fig. 6.12 Execution Times of GTA Vitirbi Programs on Spark Cluster ((64 CPUs))

Chapter 7

Conclusion

7.1 Summary of the Thesis

We have developed a high-level programming framework that supports efficient parallel processing of various data structures such as lists, sets, trees, and graphs. The framework provides a simple programming interface that parallelism is not user-concerned. It can do light-weight program transformations to produce efficient and scalable parallel programs. Moreover, it enforces deterministic semantics and simplifies composing, optimizing, porting, reasoning about, debugging, and testing parallel programs.

In the first part of this thesis, we have studied the structural recursions on lists, and trees. We introduced how to transform such structural recursions to efficient parallel programs that are implemented by algorithmic skeletons such as MapReduce. The automatic optimization mechanism makes it possible that the users do not need to pay cautions about parallelism or scalability.

In the second part, we have introduced the approach to mapping graphs to tree decompositions and the approach to making efficient parallel programs on those tree decompositions. Many NP-hard problems have polynomial time DP algorithms on their tree decompositions, theoretically. To make the DP programming being efficient, the width of the tree decomposition should be as small as possible. For piratical use, e.g, to analyze a social network, the width of the tree decompositions must be kept in 20 [64], otherwise the constant factor of the DP algorithms will be too large. However, arbitrary graphs do not have bounded treewidth and so that those efficient DP algorithms are usually not practically useful. The solution we

have given in this thesis is that we generate a partial graph of the input graph with bounded treewidth, so that an approximate solution can be obtained form this partial graph. Our approach wraps the complexity of DP programming on tree decompositions by providing simple programming interface. The experimental results show that we can systematically develop efficient programs from specifications of problems.

We have also proposed the general design of libraries with optimization capabilities based on the calculational approach in our theories. The implementation of the program transformation is lightweight in the sense that it does not need deep analysis of code of programs. The specifications are deterministically mapping to parallel skeletons. Our programming framework has already been used in practice to handle difficult problems[1].

7.2 Future Work

Firstly, we have shown some strategies to derive efficient parallel programs from simple (naive) but clear specifications for a class of problems. However, the specifications must be represented in the form of structural recursions on lists or trees, and must use the provided (also restricted) associative operators of our framework. The third homomorphism theorem is a key idea to develop such associative operators. In practice, how to automatically deriving binary associative operators for list homomorphisms is still a open problem.

Secondly, we have made some efforts to develop a sophisticated DSL embedded in Scala in our framework, which is in the form of *generate-test-aggregate*. For problems on lists and graphs, many computations can be specified in GTA form. However, for more general cases, a uniform embedded domain specific language for high-level parallel programming is demanded. We still need to make more efforts on extending GTA to more general cases.

Thirdly, our results on the *Target Set Selection Problem* show the profits of our approximation approach. However, the approximation factor is difficult to analyze and thus it is not given. In the future, we are going to do more theoretical analysis on the algorithmic guarantees. Furthermore, we will try to extend our current approximation approach to resolve a more general class of optimization problems.

[1] We have some reports that show some programmers use our library to resolve their realistic problems in USA. We public our libraries as free and open-source projects that hosted in GitHub and Google Code.

References

[1] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 10(2):287–302, 1989.

[2] S. Adachi. Design of algorithmic parallel skeletons and its implementation in mpi, 2001.

[3] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 144–155, 2012.

[4] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16:87–121, 2001. Gordon & Breach (Taylor & Francys group).

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

[6] A. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems on graphs embedded in k-trees. Technical Report TRITA-NA-8404, Royal Institute of Technology, 1984.

[7] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a structured high-level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[8] O. Ben-Zwi, D. Hermelin, D. Lokshtanov, and I. Newman. Treewidth governs the complexity of target set selection. *Discrete Optimization*, 8(1):87–96, 2011.

[9] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eskel. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 761–770, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[11] R. Bird and O. Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer Berlin Heidelberg, 1993.

[12] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.

[13] 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.

[14] R. S. Bird. Lectures on Constructive Functional Programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory, 1988.

[15] R. S. Bird. Algebraic identities for program calculation. *Comput. J.*, 32(2):122–126, apr 1989.

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

[17] R. S. Bird. Maximum marking problems. *J. Funct. Program.*, 11(4):411–424, 2001.

[18] R. S. Bird. Loopless functional algorithms. In T. Uustalu, editor, *MPC*, volume 4014 of *Lecture Notes in Computer Science*, pages 90–114. Springer, 2006.

[19] H. Bischof and S. Gorlatch. Double-scan: Introducing and implementing a new data-parallel skeleton. In B. Monien and R. Feldmann, editors, *Euro-Par 2002*, volume 2400 of *LNCS*, pages 640–647. Springer, 2002.

[20] G. E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, 1989.

[21] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language. W3C Working, April 2005.

[22] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *In Mathematical foundations of computer science*, page pages. Springer, 1998.

[23] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Preprocessing for treewidth: A combinatorial analysis through kernelization. In *Proceedings of the 38th International Colloquim Conference on Automata, Languages and Programming - Volume Part I*, ICALP'11, pages 437–448, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] H. L. Bodlaender and A. M. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[25] H. L. Bodlaender and B. van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Information and Computation*, 167(2):86 – 119, 2001.

[26] R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.

[27] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society, 1996.

[28] N. Chen. On the approximability of influence in social networks. In S.-H. Teng, editor, *SODA*, pages 1029–1037. SIAM, 2008.

[29] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD*, January 2010.

[30] 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.

[31] W.-N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *ICCL*, pages 153–, 1998.

[32] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In M. Odersky and P. Wadler, editors, *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM, 2000.

[33] M. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, 1989.

[34] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

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

[36] 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.

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

[38] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004.

[39] S. Curtis. *A Relational Approach To Optimization Problems*. PhD thesis, Oxford University Computing Laboratory, 1996.

[40] S. A. Curtis. The classification of greedy algorithms. *Sci. Comput. Program.*, 49(1-3):125–157, 2003.

[41] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *Parallel Architectures & Languages Europe*. Springer-Verlag, 93.

[42] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theor. Comput. Sci.*, 269(1-2):135–162, 2001.

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

[44] K. Emoto. *Homomorphism-based Structured Parallel Programming*. PhD thesis, University of Tokyo, 2009.

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

[46] 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.

[47] K. Emoto, Z. Hu, K. Kakehi, K. Matsuzaki, and M. 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.

[48] K. Emoto and H. Imachi. Parallel tree reduction on mapreduce. In H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *ICCS*, volume 9 of *Procedia Computer Science*, pages 1827–1836. Elsevier, 2012.

[49] 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.

[50] 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.

[51] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[52] J. Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.

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

[54] J. Gibbons, W. Cai, and D. B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

[55] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in Facebook: A case study of unbiased sampling of OSNs. In *Proc. Conf. on Computer Communications*, pages 2498–2506, 2010.

[56] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

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

[58] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[59] 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.

[60] 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.

[61] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim et al., editors, *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings*, Lecture Notes in Computer Science 1586, pages 123–137, 1999.

[62] C. S. Groer, B. D. Sullivan, and D. P. Weerapurage. INDDGO: Integrated network decomposition & dynamic programming for graph optimization. Technical Report ORNL/TM-2012/176, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2012.

[63] S. B. Herodotos Herodotou. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment (PVLDB)*, 4(11):1111–1122, 2011.

[64] F. Hüffner, R. Niedermeier, and S. Wernicke. Developing fixed-parameter algorithms to solve combinatorially explosive biological problems. In J. Keith, editor, *Bioinformatics*, volume 453 of *Methods in Molecular Biology*, pages 395–421. Humana Press, 2008.

[65] Z. Hu. Calculational parallel programming. In *HLPP'10: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, page 1. ACM, 2010.

[66] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418. Springer-Verlag, 1996.

[67] 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.

[68] 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.

[69] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *Proc. 2002 European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 2002.

[70] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *11th European Symposium on Programming (ESOP 2002)*, pages 83–97. Springer Verlag, 2002. Lecture Notes in Computer Science 2305.

[71] 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.

[72] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 85–94. BRICS Notes Series NS-99-1, 1999.

[73] Intel Corporation. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[74] H. Iwasaki and Z. Hu. A new parallel skeleton for general accumulative computations. *International Journal of Parallel Programming*, 32:389–414, 2004.

[75] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, mar 2011.

[76] K. Jérôme, S. Martina, H. Holger, and D. Daniel. The koblenz network collection. http://konect.uni-koblenz.de/.

[77] M. Kay. XSL Transformations (XSLT) Version 2.0, January 2007.

[78] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[79] M. Kiminori, E. Kento, and L. Yu. Parallelization of regular expression matching and its evaluation on hadoop. *IPSJ Transactions: Programming*, 4(4):1–11, sep 2011.

[80] A. Koster. Frequency assignment - models and algorithms, 1999.

[81] H. Kuchen. A skeleton library. In *Proceedings of Euro-Par 2002, Lecture Notes in Computer Science*, volume 2400, pages 620–629. Springer-Verlag, 2002.

[82] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. In *Proceedings of 3rd International Workshop on "Constructive Methods for Parallel Programming" (CMPP 2002)*, pages 3–16, 2002.

[83] R. Lämmel. Google's MapReduce programming model — Revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[84] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance. Cost-effective outbreak detection in networks. In P. Berkhin, R. Caruana, and X. Wu, editors, *KDD*, pages 420–429. ACM, 2007.

[85] Y. Liu, K. Emoto, and Z. Hu. A generate-test-aggregate parallel programming library: systematic parallel programming for mapreduce. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, number 11 in PMAM '13, pages 71–81, New York, NY, USA, 2013. ACM.

[86] Y. Liu, K. Emoto, and Z. Hu. A generate-test-aggregate parallel programming library for systematic parallel programming. *Parallel Comput.*, 40(2):116–135, Feb. 2014.

[87] 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.

[88] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[89] B. Lu and J. M. Mellor-Crummey. Compiler-optimization of implicit reductions for distributed memory multiprocessors. In *IPPS/SPDP*, pages 42–51, 1998.

[90] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[91] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. In *the International Workshop on "Constructive Methods for Parallel Programming" (CMPP'04)*, pages 109–123, 2004.

[92] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Proceedings of the Annual European Conference on Parallel Processing (EuroPar'03), LNCS 2790*, pages 789–798. Springer-Verlag, 2003.

[93] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. Technical Report METR 2003-21, Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2003.

[94] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 39–48, New York, NY, USA, 2006. ACM.

[95] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. 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.

[96] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653. Springer-Verlag, 2004.

[97] E. W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. In *SPAA*, pages 109–117, 1992.

[98] E. W. Mayr and R. Werchner. Optimal tree contraction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, 1997.

[99] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *FOCS*, pages 478–489. IEEE Computer Society, 1985.

[100] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489. IEEE Computer Society Press, 1985.

[101] A. Morihata. *Calculational Approach to Automatic Algorithm Construction*. PhD thesis, University of Tokyo, 2009.

[102] A. Morihata, K. Kakehi, Z. Hu, and M. Takeichi. Swapping arguments and results of recursive functions. In T. Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 379–396. Springer Berlin Heidelberg, 2006.

[103] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009*, pages 177–185. ACM, 2009.

[104] 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.

[105] MPI Forum. Message Passing Interface. http://www.mpi-forum.org/.

[106] 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.

[107] S. C. Mu and R. S. Bird. Theory and applications of inverting functions as folds. *Science of Computer Programming (Special Issue for Mathematics of Program Construction)*, 51:87–116, 2003.

[108] A. Nichterlein, R. Niedermeier, J. Uhlmann, and M. Weller. On tractable cases of target set selection. *Social Network Analysis and Mining*, 3(2):233–256, 2013.

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

[110] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, 1996.

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

[112] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, UK, UK, 2003.

[113] L. R. Rabiner. Readings in speech recognition. chapter A tutorial on hidden Markov models and selected applications in speech recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[114] N. Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[115] A. V. H. Röhrig, T. Hagerup, and H. Röhrig. Tree decomposition: A feasibility study. *Master's thesis, Max-Planck- . . .* , (September), 1998.

[116] 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.

[117] 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.

[118] S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726. IEEE, 2010.

[119] P. Shakarian, S. Eyre, and D. Paulo. A scalable heuristic for viral marketing under the tipping model. *Social Netw. Analys. Mining*, 3(4):1225–1248, 2013.

[120] 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.

[121] D. B. Skillicorn. The bird-meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[122] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[123] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

[124] D. B. Skillicorn. A parallel tree difference algorithm. *Information Processing Letters*, 60(5):231–235, 1996.

[125] G. L. Steele Jr. Parallel programming and parallel abstractions in Fortress. In M. Hagiya and P. 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.

[126] B. D. Sullivan, D. P. Weerapurage, and C. S. Groer. Parallel algorithms for graph optimization using tree decompositions. Technical Report ORNL/TM-2012/194, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2012.

[127] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, aug 2009.

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

[129] Q. Wang, M. Chen, Y. Liu, and Z. Hu. Towards systematic parallel programming of graph problems via tree decomposition and tree parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 25–36, New York, NY, USA, 2013. ACM.

[130] F. Wei. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 99–110, 2010.

[131] T. V. Wimer. *Linear Algorithms on k-Terminal Graphs*. PhD thesis, Clemson University, 1987. Report No. URI-030.

[132] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[133] 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.

[134] H. Zhenjiang, I. Hideya, T. Masato, and T. Akihiko. Tupling calculation eliminates multiple data traversals. In *In ACM SIGPLAN International Conference on Functional Programming*, pages 164–175. ACM Press, 1997.

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

International Journals

1. Kiminori Matsuzaki, Kento Emoto, Yu Liu, Parallelization of Regular Expression Matching and Its Evaluation on Hadoop. Information Processing Society of Japan Transactions on Programming (情報処理学会論文誌 トランザクション プログラミング), Vol.4 No.4, 2011.

2. Yu Liu, Kento Emoto, Zhenjiang Hu, A Generate-Test-Aggregate Parallel Programming Library for Systematic Parallel Programming, Parallel Computing, Volume 40, Issue 2, Feb 2014, pp 116-135, Elsevier.

3. Yu Liu, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Accumulative Computation on MapReduce. Information Processing Society of Japan Transactions on Programming (情報処理学会論文誌：プログラミング), Vol.7, No.1, pp 1-10, Jan 2014.

Refereed Papers (International Conferences and Workshops)

1. Yu Liu, Zhenjiang Hu, Kiminori Matsuzaki, Towards Systematic Parallel Programming over MapReduce, In proceedings of 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), Lecture Notes in Computer Science Vol. 6853, Part II, page:35-50, Springer, 2011

2. Yu Liu, Kento Emoto, Zhenjiang Hu, A Generate-Test-Aggregate Parallel Programming Library, In proceedings of The 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, Shenzhen, China, Feb 23, 2013.

3. Qi Wang, Meixian Chen, Yu Liu and Zhenjiang Hu. Towards Parallel Tree/Graph Computation with MapReduce. Accepted by 2nd Workshop on Functional High-Performance Computing (FHPC2013), 23rd, September 2013, Boston, Massachusetts, US.

Domestic Conference and Workshop Papers

1. Yu Liu, Zhenjiang Hu, A Homomorphism-based Framework for Systematic Parallel Programming with MapReduce , 第13回プログラミングおよびプログラミング言語ワークショップ(PPL2011) 論文集 pp116130, 日本ソフトウェア科学会 プログラミング論研究会, 2011.3

2. Yu Liu, Sebastian Fischer, Kento Emoto, Zhenjiang Hu, Generate-Test-and-Aggregate アルゴリズムのHadoop実装 (英語：Implementing Generate-Test-and-Aggregate Algorithms on Hadoop). 日本ソフトウェア科学会第28 回大会(2011年度) 講演論文集, 2011

3. Yu Liu, A Calculational Approach to Systematically Developing MapReduce Programs Master thesis, 2011.07

4. Tao Zan, Yu Liu, Zhenjiang Hu, Automatic Parallelization of Graph Queries with MapReduce, in proceedings of 29th 日本ソフトウェア科学会大会, Aug 13, 2012.